RESEARCH

Korbinian Molitorisz

Musterbasierte Parallelisierung sequenzieller Anwendungen

Konzept und Implementierung eines Verfahrens zur Softwaretransformation

Springer Vieweg

Musterbasierte Parallelisierung sequenzieller Anwendungen

Korbinian Molitorisz

Musterbasierte Parallelisierung sequenzieller Anwendungen

Konzept und Implementierung eines Verfahrens zur Softwaretransformation

Springer Vieweg

Dr.-Ing. Korbinian Molitorisz
Karlsruhe, Deutschland

ISBN 978-3-658-15094-5 ISBN 978-3-658-15095-2 (eBook)
DOI 10.1007/978-3-658-15095-2

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer Fachmedien Wiesbaden GmbH

Vorwort

„The free lunch is over“ – die Zeit kostenloser Beschleunigung von Software ist vorbei. Mit diesem Satz kommentierte Herb Sutter im Jahr 2005 den Umstand, dass die CPU-Taktfrequenz aufgrund physikalischer Limitierungen nicht weiter erhöht werden kann. Aus Sicht der Softwareentwicklung hat dies zur Folge, dass Software nunmehr in erster Linie durch manuelle Parallelprogrammierung beschleunigt werden muss. Diese Aufgabe ist aber nicht nur wissens- und zeitintensiv, sondern darüber hinaus auch fehlerträchtig.

Die vorliegende Arbeit stellt das Konzept *AutoPar* zur Parallelisierung von Anwendungssoftware vor, die nicht für die Ausführung auf Mehrkernprozessoren entwickelt wurde und somit auch nicht von der steigenden Rechenleistung profitieren kann. Dazu wird zunächst ein allgemeines Rahmenwerk zur Softwareparallelisierung vorgestellt. Auf dieser Basis wird das Verfahren *AutoPar* definiert, das aus drei Phasen besteht. *AutoPar* identifiziert drei Arten der Parallelverarbeitung in sequenzieller Software, transformiert diese in parallele Software und passt den Grad an Parallelität an das Zielsystem an.

In der Identifikations- und Transformationsphase stützt sich *AutoPar* auf einen vordefinierten Katalog an sequenziellen Ausgangs- und parallelen Zielmustern zur Erzeugung von Aufgaben-, Daten- und Fließbandparallelität. Für jedes dieser Musterpaare identifiziert *AutoPar* sogenannte *Tuning*-Parameter, deren Werte das Laufzeitverhalten der parallelen Software verändern. *AutoPar* gibt die Garantie nach korrekter Semantik der erzeugten parallelen Software zugunsten eines größeren Suchraums nach Parallelität auf. Es beinhaltet daher ein automatisches Verfahren zur effizienten Erkennung von Datenwettläufen durch erschöpfende Fadenverschränkung.

Zur Evaluierung der Identifikations- und Transformationsqualität wurden diese beiden Verfahrensteile separat analysiert. Die Identifikationsphase erzielte in einem *benchmark* aus sechs regulären und irregulären Anwendungen verschiedener Anwendungsdomänen mit mehr als 26.000 Quellcodezeilen eine Ausbeute von 99,0 % bei einer Präzision von 67,8 %. Diese Werte ergeben ein gewichtetes harmonisches Mittel von 75,4 %. Für die Transformationsqualität wurden die erzielten Beschleunigungswerte mit manuellen Parallelisierungen derselben *benchmark*-Programme verglichen. Nach Anwendung von *Auto Tuning* ergab sich auf dem Testsystem mit acht Kernen eine mittlere Beschleunigung von 2,72 (*AutoPar*) gegenüber

3,52 (manuell). Damit erreicht *AutoPar* in den ausgewählten Programmen zu 97,32 % die Leistung menschlicher Entwickler und benötigt dafür lediglich wenige Minuten.

In einer Entwicklerstudie mit drei Gruppen und zehn Teilnehmern zeigte sich, dass *AutoPar* in der Lage ist, mehr Parallelisierungspotenzial in kürzerer Zeit zutage zu fördern als das kommerzielle Parallelisierungswerkzeug INTEL PARALLEL STUDIO. Ferner zeigte die Auswertung der manuellen Kontrollgruppe, dass *AutoPar* die von Entwicklern als relevant erachteten Informationen zur Parallelisierung heranzieht und sie dem Entwickler dabei auch auf geeignete Weise visualisiert.

„Free lunches“ mögen vorbei sein, aber diese Arbeit zeigt, dass „free snacks“ definitiv noch erhältlich sind.

Korbinian Molitorisz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die vorliegende Dissertation befasst sich mit der Problemstellung, sequenzielle Software so zu transformieren, dass sie auf modernen Mehrkernprozessoren Beschleunigung erfährt. Dazu werden bestimmte Programmstrukturen der sequenziellen Software auf äquivalente parallele Strukturen abgebildet, falls die vorhandenen Abhängigkeiten eine Parallelausführung ermöglichen. Die transformierte Software ist zudem optimierbar und verifizierbar. Optimierbar insofern als sie über definierte Programmparameter verfügt, deren Werte direkten Einfluss auf das Laufzeitverhalten haben. Verifizierbar insofern als die parallelen Strukturen auf Datenwettläufe hin überprüft werden können.

Diese Dissertation erfasst relevante Arbeiten in dem geschilderten Problemkontext, vergleicht und bewertet sie. Aus dieser Diskussion werden konkrete Problemstellungen abgeleitet, konzeptionelle Lösungsansätze diskutiert und anschließend prototypisch implementiert. Die Evaluierung erfolgt erstens über eine Reihe von Fallstudien, die die Präzision und Ausbeute des Erkennungsverfahrens belegen, zweitens über die Beschleunigung, die das Transformationsverfahren erzielt, sowie drittens über den Zeitvorteil und Benutzerkomfort für Softwareentwickler bei der Verwendung des vorliegenden Verfahrens.

Zusammengenommen belegt die Evaluierung damit die Qualität des Verfahrens, seine vielseitige Verwendbarkeit und den Nutzen der prototypischen Implementierung für die Softwaretechnik im Allgemeinen.

1.1 Motivation

„*The free lunch is over.*“ – die Zeit der kostenlosen Beschleunigung von Software durch Erhöhung der CPU-Taktfrequenz ist vorbei. Mit diesem Satz kommentierte Herb Sutter im Jahr 2005 die beginnende *Multicore*-Ära [Sutt05]. Seit dieser Zeit wird nicht mehr nur im Nischenbereich des Hochleistungsrechnens tatsächlich parallel gerechnet, sondern in praktisch jedem Computersystem, vom Hochleistungsrechner bis hin zum tragbaren Mobilgerät. Parallele Hardware erfordert aber parallele Software, um voll ausgenutzt zu werden, und so erfährt Software seit Anbeginn dieser Ära Beschleunigung vorwiegend durch explizite Parallelität.

K. Molitorisz, *Musterbasierte Parallelisierung sequenzieller Anwendungen*,
DOI 10.1007/978-3-658-15095-2_1

Die *Multicore*-Ära führte somit auch zu einem neuen Verständnis von Softwareentwicklung: War es bisher die Aufgabe von Softwareentwicklern, Spezifikationsdokumente in Programmlogik zu überführen und dabei die beabsichtigte Semantik sicherzustellen, müssen sie nunmehr zusätzlich entscheiden, welche Teile ihrer Programmlogik zu welchem Zeitpunkt parallel zueinander ausgeführt werden können.

Diese als Parallelisierung bezeichnete Aufgabe besitzt mehrere Teilaspekte: Es muss geklärt werden, an welchen Stellen die Programmausführung aufgespaltet werden kann, nach welchen Strategien das jeweils geschehen soll, wie diese Strategien technisch implementiert werden müssen und ob dies schlussendlich zu einem schnelleren und weiterhin korrekten parallelen Programm führt. Wie L. Hochstein et al. im Artikel [HCSA05] aufzeigen, ist der Aufwand für den Entwurf und die Implementierung paralleler Programmlogik alleine fast 2,5 mal so hoch wie bei traditioneller sequenzieller Softwareentwicklung.

Die Parallelisierung ist momentan eine vorwiegend manuelle Aufgabe und aus den angeführten Gründen nicht nur wissens- und zeitintensiv, sondern darüber hinaus auch fehlerträchtig. Die *Multicore*-Ära stellt also höhere Anforderungen an Softwareentwickler. Wie J. Link in [Link12] feststellt, sind die Fähigkeiten zur Parallelisierung fast ein Jahrzehnt nach Beginn des *Multicore*-Zeitalters noch nicht breitflächig vorhanden. Softwareentwickler benötigen also umfassende Unterstützung bei der Entwicklung von paralleler Software. Wie L. Hochstein et al. ferner feststellen, parallelisieren erfahrene Entwickler absolut gesehen zwar schneller als unerfahrene, aber auch sie benötigen viel Zeit zur Lokalisierung, Leistungsoptimierung und Korrektheit. Eine Unterstützung ist also ungeachtet individueller Vorkenntnisse für alle Entwickler dringend notwendig.

1.2 Problemstellung

Die große Mehrheit an Software stammt aus einer Zeit, in der CPUs lediglich über einen einzigen Kern verfügten, und sich Softwareentwickler rein auf die Entwicklung der Programmlogik konzentrieren konnten. Aufgrund der gängigen Praxis, bei der Weiterentwicklung von Software bestehende Komponenten wiederzuverwenden, ist auch heute noch ein beträchtlicher Anteil sequenziell [BoKa07]. Dass dies ein Problem für rechenintensive Software darstellt, ist offensichtlich. Da aber seit der Einführung von Mehrkern-CPUs die Taktraten von Prozessorkernen sogar deutlich auf das Niveau von Prozessoren um die Jahrtausendwende gesunken sind, muss die Parallelisierung nicht mehr nur die Anforderung erfüllen, Software zu beschleunigen, sondern auch, die ursprüngliche Leistung sicherzustellen und zu erhalten. Dies macht die Notwendigkeit zur Parallelisierung bestehender Software umso deutlicher.

Die komplette Neuentwicklung von bestehender Software ist allerdings aufgrund der hohen finanziellen und personellen Aufwände, die dabei anfallen, nur in den wenigsten Fällen eine valide Option, zumal überdies technisch versierte Parallelexperten in großer Zahl benötigt werden, die – wie eingangs bereits geschildert – ja nicht vorhanden sind. Auch bei Neuentwicklungen, die nicht auf bestehender Software aufbauen, fehlen aktuell solche Experten. Wie V. Pankratius et al. in der Entwicklerstudie [PaJT09] zeigen, kann die Einarbeitungszeit in die Mehrkernprogrammierung und in die vorliegende Software beträchtlich sein. Dadurch wird deutlich, dass auch bei der Neuentwicklung Verfahren und Hilfestellungen zur Parallelisierungsunterstützung notwendig sind.

Heutzutage verfügen alle gängigen Entwicklungsumgebungen über eine Vielzahl an Möglichkeiten, um Softwareentwickler bei ihrer Arbeit zu unterstützen: Wiederverwendbare Klassenbibliotheken, Speicherbereinigung, Komponententests und Codevervollständigung sind nur einige prominente Beispiele. Die Unterstützungsleistung bei der Parallelprogrammierung beschränkt sich im Moment hingegen weitestgehend auf Spracherweiterungen oder Klassenbibliotheken, mit deren Hilfe die Erzeugung und die Synchronisierung paralleler Aktivitäten vereinfacht wird [Rola03]. Zu diesem Zeitpunkt muss die Frage, an welchen Stellen zu parallelisieren ist, und nach welcher Strategie dies zu geschehen hat, aber bereits beantwortet worden sein. Hierfür fehlt es bislang an umfassender Werkzeugunterstützung.

Zur Identifikation parallelisierbarer Stellen werden häufig sogenannte *hot spots* lokalisiert. Dabei handelt es sich um laufzeitintensive Teile der Software. Dies beruht auf der Annahme, dass Stellen mit hohem Laufzeitanteil auch zu besonders hohen Beschleunigungswerten durch Parallelisierung führen. Dies ist aber kein hinreichendes Kriterium, da zur Parallelisierung nicht nur die Frage gehört, welche Stellen hohes Parallelisierungspotenzial aufweisen, sondern auch, welche Coderegionen tatsächlich parallel zueinander ausgeführt werden können. An diese beiden Fragestellungen schließen sich dann sogar noch weitere an: Es muss beantwortet werden, wie eine solche Stelle effektiv zu parallelisieren ist. Letztlich aber sind die beiden Fragen, die über Erfolg oder Misserfolg einer Softwareparallelisierung entscheiden, ob die parallele Version die Ausführungszeit beschleunigt und dieselbe Semantik besitzt wie zuvor. Beides muss auf jeden Fall sichergestellt sein, da weder ein schnelleres aber inkorrektes Programm akzeptabel ist noch ein korrektes aber langsameres.

Seit dem kommerziellen Durchbruch der *Multicore*-Ära um die Jahrtausendwende nimmt die Anzahl an Kernen, die sich auf einem Prozessor befinden, stetig zu. Aus Sicht der Softwaretechnik führt dies zu dem Problem, dass parallele Software so entworfen werden muss, dass sie auf Architekturen mit unterschiedlichen Kernzahlen beschleunigt. Andernfalls kann die parallele Software bereits in naher Zukunft die zur Verfügung stehende Parallelität nicht mehr voll ausschöpfen. Um dies zu erreichen, ist es also nötig, laufzeitrelevante Parameter für Parallelsoftware zu definieren und deren Werte variabel zu halten.

Eines der gebräuchlichsten Programmierparadigmen in heutiger Anwendungssoftware ist die Objektorientierung. Sie bietet ein Programmiermodell, das sehr leicht verständlich und flexibel ist. Dynamische Datentypen, späte Typbindung und Vererbung sind drei prominente Eigenschaften dieses Modells. Wie M. Philippsen in [Phil00] aufzeigt, lassen sich Konzepte der Parallelprogrammierung sehr gut auf die Objektorientierung abbilden. Die Flexibilität objektorientierter Software erschwert dabei aber parallelisierende Programmanalysen erheblich, da gewisse Zusammenhänge erst zur Laufzeit als sicher angenommen werden können. Aus Sicht der Parallelisierung muss ein Verfahren diese Unwägbarkeiten in geeigneter Weise beachten, um Parallelisierungspotenzial ableiten zu können.

Die große Klasse irregulärer Anwendungen wird bestimmt von unvorhersehbaren Speicherzugriffen und referenzierten Datentypen, wie etwa verketteten Listen, Graphen oder nicht balancierten Bäumen. Das Zugriffsverhalten auf diese Datentypen kann derart komplex sein, dass es nicht als trivial gilt, sie auf Parallelisierbarkeit zu untersuchen.

Aus dieser einleitenden Diskussion ergeben sich die folgenden Probleme, die im Rahmen dieser Arbeit behandelt werden:

- Wie kann man Softwareentwickler sowohl bei der Transformierung bestehender sequenzieller, als auch bei der Entwicklung paralleler Software geeignet unterstützen? Wie muss eine solche Unterstützung aussehen?
- An welchen Stellen einer objektorientierten Software ist es möglich und sinnvoll zu parallelisieren?
- Welche Parallelisierungsstrategien bieten sich für Software an, die nicht für die Ausführung auf Mehrkern-CPUs vorgesehen ist?
- Welche laufzeitrelevanten Parameter gibt es? Wie lassen sie sich in sequenzieller Software identifizieren und so implementieren, dass sie systematisch und ohne Auswirkung auf den zugrundeliegenden parallelen Quellcode an die Zielplattform angepasst werden können?
- Wie stellt man sicher, dass es durch das Aufspalten des Kontrollflusses nicht zu Datenwettläufen kommt oder diese zumindest erkannt werden?

In unseren früheren Arbeiten [Moli13, MoMT15, SMJT13] konnten wir bereits zeigen, dass parallelisierbare Programmstrukturen innerhalb weniger Minuten mit einer Ausbeute von annähernd 100 % identifiziert werden können, und sich nach automatischer Anpassung der erkannten Laufzeitparameter eine Leistungssteigerung von bis zu 415 % gegenüber der sequenziellen Software erzielen lässt. Dies zeigt den Mehrwert der musterbasierten Parallelisierung und ihre Bedeutung für die Softwaretechnik als solche.

1.3 Gliederung der Arbeit

Auf der Grundlage der soeben eingeführten Problemstellung wird in Kapitel 2 zunächst eine Zielsetzung erörtert und der wissenschaftliche Beitrag zur Softwaretechnik erläutert. Anschließend werden die vier zentralen Arbeitshypothesen für diese Arbeit aufgestellt. Kapitel 3 stellt Grundbegriffe und Grundlagen aus den Forschungsbereichen vor, die für diese Arbeit relevant sind. Anschließend werden in Kapitel 4 relevante wissenschaftliche Arbeiten vorgestellt und zu dieser Arbeit in Bezug gesetzt. Kapitel 5 stellt das Konzept der musterbasierten Parallelisierung vor, dessen prototypische Implementierung in Kapitel 6 beschrieben wird. Die Evaluierung mittels verschiedener Fallstudien folgt in Kapitel 7, in dem auch die Erfüllung der Thesen überprüft wird. Kapitel 8 rundet diese Arbeit ab mit der Zusammenfassung der wesentlichen Erkenntnisse, nennt konkrete Anknüpfungspunkte und wirft Forschungsfragen auf, die in zukünftigen Arbeiten behandelt werden sollten.

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.

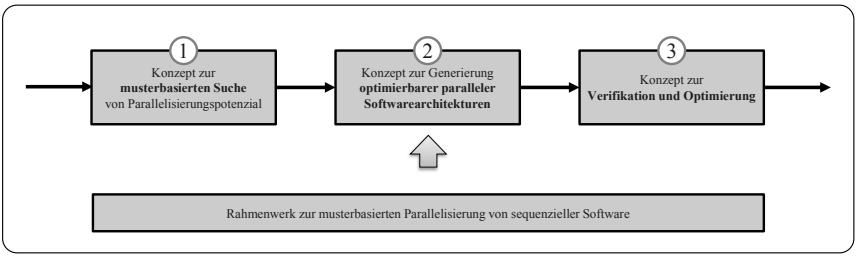

Abbildung 2.1: *AutoPar* - Musterbasierte Parallelisierung [Moli13]

K. Molitorisz, *Musterbasierte Parallelisierung sequenzieller Anwendungen*,
DOI 10.1007/978-3-658-15095-2_2

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-CPUs 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 „Musterausprägung“ 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 selbständig 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 Wettlauferkenner 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 Wettlauferkenner zeigen wir, dass man auch ohne Beachtung der Pfadüberdeckung annähernd alle vorhandenen parallelen Fehler lokalisieren kann. Während der beste Wettlauferkenner eine Erkennungsrate von etwa 50 % erreichte, konnten bis zu 92 % der tatsächlichen parallelen Fehler gefunden werden, indem diese Wettlauferkenner 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.