Kontrollflussbasierte Leistungsschätzung zur Parallelisierung

Studienarbeit von

Sergej Poinzew

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

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

Karlsruhe, den 28. Februar 2012

Sergej Pimzew
INHALTSVERZEICHNIS

1 Einleitung .................................. 1
1.1 Einführung in das Themengebiet .................................................. 1
1.2 In der Arbeit behandelte Fragestellungen ...................................... 2
1.3 Beschreibung des Evaluierungsszenarios ....................................... 2
1.4 Gliederung der Arbeit ................................................................. 2

2 Grundlagen .................................. 4
2.1 Performanz Evaluierung und Schätzung ......................................... 4
  2.1.1 Amdahlsches Gesetz ................................................................. 4
  2.1.2 Gustafsons Gesetz ................................................................. 5
2.2 Parallele Softwareentwicklung ...................................................... 6
  2.2.1 Parallelisierungsprozess ......................................................... 6
  2.2.2 Parallele Entwurfsmuster ....................................................... 7
2.3 Common Language Infrastructure (CLI) ....................................... 9
2.4 Common Compiler Infrastructure (CCI) ....................................... 11

3 Verwandte Arbeiten ................................ 12
3.1 Estimating Parallel Performance, A Skeleton-Based Approach [LL10] .................. 12
3.2 Function Level Parallelism Driven by Data Dependences [RV+07] .................... 14
3.3 Automated Experimental Parallel Performance Analysis [LD02] ...................... 15
3.4 Facilitating Performance Predictions Using Software Components [KK+07] .......... 16

4 Leistungsschätzung zur Parallelisierung ........................................ 18
4.1 Ziele und Anforderungen ............................................................ 18
4.2 Statische und dynamische Analyse sequenzieller Anwendungen ...................... 19
  4.2.1 Statische Analyse mit Common Compiler Infrastructure ..................... 20
  4.2.2 Dynamische Analyse mit dem Microsoft Visual Studio Profiler .............. 23
  4.2.3 Aufbau des Ausführungsgraphen ............................................. 25
  4.2.4 Analyse der Zielplattform ..................................................... 26
4.3 Das Schätzverfahren LoopEst ...................................................... 26
  4.3.1 Schätzung von Schleifen und Methoden ....................................... 27
  4.3.2 Leistungsschätzung des gesamten Programms ................................ 28
  4.3.3 Schätzung durch Skalierbarkeit .............................................. 29
4.4 Zusammenfassung ........................................................................ 30

5 Evaluierung ..................................... 32
5.1 Eigene Beispiele ............................................................................ 32
5.2 Matrizen- und Vektormultiplikationen .......................................... 34
5.3 Beispiele für parallele Programmierung mit .NET 4 .................................. 35
5.4 Zusammenfassung ........................................................................ 37

6 Zusammenfassung und Ausblick ................................................. 38

Anhänge ........................................ 39

Lehrstuhl Prof. Dr. Walter F. Tichy
1 EINLEITUNG

1.1 Einführung in das Themengebiet


Heute besitzen fast alle CPUs zwei, vier oder mehr Kerne auf einem Chip. Insbesondere für den Serverbereich ist es wichtig, mehrere unabhängige Recheneinheiten zu haben; aber auch eingebettete Systeme und Tablet PCs verfügen bereits heute über Mehrkernprozessoren. Im Jahr 2011 sind erstmals die Mobiltelefone mit Mehrkernprozessoren auf den Markt gekommen.

„Moore’s Law scaling should easily let us hit the 80-core mark in mainstream processors within the next ten years and quite possibly even sooner.“ (Justin Ratnerm, Leiter der Technologieabteilung bei der Firma Intel. Siehe [Rat10])

Ein derartiger Paradigmenwechsel in der Hardwarebranche reicht aber dennoch nicht aus, um die alte sequenzielle Software automatisch zu beschleunigen, so wie es früher mit steigenden Taktfrequenzen war. Moderne Betriebssysteme können durch ihren Ablaufplaner (engl. process scheduler) mehrere Prozessorkerne zeitgleich verwenden, allerdings sind sie nicht in der Lage, ein Programm, das nur aus einem Faden (engl. thread) besteht, auf mehrere Kerne zu verteilen. Daher sollte man bereits bei der Entwicklung einer Software beachten, dass das Programm von sich aus neue Fäden erzeugt und diese auch steuert.


Doch leider hat sich noch immer kein Weg für die Entwicklung paralleler Programme oder für die Parallelisierung der alten, sequenziellen Software gefunden, der absolut richtig und universell ist. Vor allem neue Fehlerarten – die als Folge der Parallelität entstehen – erschweren es, parallele Anwendungen zu entwickeln und zu testen. Auch trotz der großen Anzahl an unterstützenden Werkzeugen und Entwurfsmustern ist das parallele Programmieren im Vergleich zum sequenziellen Programmieren deutlich komplexer.

Fest steht, dass zum einen parallele Abläufe unbedingt kontrolliert verlaufen müssen. Zum anderen muss sowohl der ausführlichen Planung paralleler Programme als auch deren Überprüfung auf Korrektheit und Performance immer mehr Bedeutung zugesprochen werden.
1.2 In der Arbeit behandelte Fragestellungen

Das Hauptziel des Parallelisierens ist es, die Performanz zu maximieren und dabei immer noch korrekte Ergebnisse zu liefern. Doch was wird eigentlich unter Performanz verstanden? Ist es nur die Ausführungszeit, die möglichst klein sein soll, oder müssen auch weitere Kriterien in Betracht gezogen werden? Und welche Rolle spielt dabei die Effizienz, da vielleicht nicht immer die größte Leistung gefragt ist, sondern auch Faktoren wie Sparsamkeit und Energieeffizienz?

In dieser Arbeit sollen sequenzielle Programme untersucht und der möglicher Leistungszuwachs durch Parallelisierung abgeschätzt werden. Die Leistungsschätzung soll auch einen automatischen Parallelisierungsprozess unterstützen, insofern muss sie möglichst automatisch und vor dem Entwickler verborgen passieren. Wenn ein Programm parallelisiert werden soll, müssen zunächst potenzielle Stellen gefunden werden, wo dieses möglich ist. Auf der untersten Ebene sind dies die sequenziell mehrmals aufrufenden Methoden und die Schleifen. Dort wird am meisten und am längsten gerechnet, sodass sich die Parallelisierung an diesen Stellen am meisten lohnt. In dieser Arbeit wird ein Werkzeug entwickelt, das durch eine Analyse des Kontrollflusses eine Leistungsschätzung für diese Kandidaten abgibt und auf diese Weise die Kandidatenmenge reduzieren kann.

Umfangreiche Leistungstests sollen dabei helfen, Engpässe zu identifizieren und den Grad der Parallelität des Programms zu erhöhen. Auch der Vergleich zwischen sequenziell und parallelem Code soll es ermöglichen, ein besseres Auswahlverfahren für den automatischen Parallelisierungsprozess und für die Auswahl von Entwurfsmuster zu schaffen (siehe Kapitel 2.2.1 und 2.2.2).

Diese Arbeit wird im Rahmen der .NET-Multicore-Gruppe unter der Leitung von Korbinian Molitorisz am Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Prof. Dr. Walter F. Tichy geschrieben und bezieht sich komplett auf die .NET-Plattform.

1.3 Beschreibung des Evaluierungsszenarios

Es werden verschiedene Codebeispiele analysiert und eine Leistungsschätzung durchgeführt. Als Beispiele wird die von Microsoft Research zur Verfügung gestellte Sammlung von Algorithmen und kleinen Anwendungen verwendet, die in sequenzieller und manuell parallelisierter Form bereit stehen.


1.4 Gliederung der Arbeit

Im zweiten Kapitel werden die Grundlagen und Entwurfsmuster der parallelen Softwareentwicklung unter .NET sowie die Evaluierungsansätze beschrieben. Es werden die wichtigsten Aspekte einer Leistungsmetrik vorgestellt und analysiert.

Im darauffolgenden Kapitel 3 wird ein Überblick über verwandte Arbeiten, die sich mit der Leistungsschätzung und Parallelisierung beschäftigen, gegeben. Verschiedene Ansätze zur Leistungsschätzung werden beschrieben.
In Kapitel 4 werden die Ansätze aus dem Kapitel 3 kritisch analysiert und es werden Ziele und Anforderungen dieser Arbeit daraus hergeleitet. Es wird das Konzept und der Aufbau des Schätzungsverfahrens beschrieben sowie die dafür entwickelten Klassen und das Werkzeug. Es folgen Testbeispiele sowie deren Bewertung.

Abschließend werden im letzten Kapitel die Ergebnisse evaluiert und zusammengefasst. Im Anhang befinden sich die in dieser Arbeit verwendeten Abkürzungen, das Abbildungsverzeichnis und sowie das Literaturverzeichnis.
2 GRUNDLAGEN

2.1 Performanz Evaluierung und Schätzung
Die Steigerung der Effizienz ist eine der wichtigsten Gründe für das Parallelisieren von Software und Algorithmen. Allerdings bedeutet die Erhöhung der Anzahl der Prozessorkerne auf eine bestimmte Zahl \( N \) nicht, dass ein paralleler Algorithmus \( N \) mal schneller läuft als eine serielle Variante. Dies liegt daran, dass sich die parallel ablaufenden Fäden nicht ausschließlich mit der Berechnung der Ergebnisse beschäftigen können, sondern auch miteinander interagieren oder aufeinander warten müssen.

Die **Beschleunigung** (angl. *speedup*) durch Parallelisierung wird als \( S(N) \) bezeichnet. Außerdem werden die Ausführungszeit eines sequenziellen Programmes \( T(1) \) und die Ausführungszeit eines parallelen Programms auf \( N \) Prozessoren als \( T(N) \) bezeichnet.

\[
S(N) = \frac{T(1)}{T(N)}
\]

Formel 1: Beschleunigung durch Parallelisierung

Die **Effizienz** \( E \) (engl. *efficiency*) ist ein Wert, der typischerweise zwischen Null und Eins liegt und bestimmt, wie gut die Prozessoren bei der Lösung eines Problems ausgelastet sind. Der Effizienzwert zeigt auch, wie groß der Mehraufwand ist, der durch die Kommunikation und Synchronisation zwischen Prozessoren entsteht.

\[
E(N) = \frac{S(N)}{N}
\]

Formel 2: Effizienzwert bei der Parallelisierung

Beim Effizienzbegriff wird zwischen drei möglichen Beziehungen der Beschleunigung und der Anzahl an Prozessoren zueinander unterschieden [Shi96]:

- Ist \( S(N) < N \), spricht man von einer sublinearen Beschleunigung
- Ist \( S(N) = N \), ist die Beschleunigung linear
- Ist \( S(N) > N \), dann ist die Beschleunigung superlinear und \( E > 100\% \)

2.1.1 Amdahlsches Gesetz

**Amdahlsches Gesetz** [Amd67]: \( P \) der Laufzeitanteil der parallelisierbaren Teilstücke eines Programms. Dann ist \( (1-P) \) der sequentielle Anteil. Für die Gesamtaufzeit \( T \) ergibt sich bei der Ausführung auf einem einzigen Prozessorkern \( T = P + (1-P) = 1 \).
Sei $N$ die Anzahl der Prozessoren, die zur Berechnung eingesetzt werden und $S(N)$ der Synchronisations- und Kommunikationsaufwand, wenn $N$ Prozessorkerne an der Berechnung beteiligt sind. Dann berechnet sich die maximale Beschleunigung $S$ zu:

$$S = \frac{1}{(1 - P) + o(N) + \frac{P}{N}} < \frac{1}{(1 - P)}$$

Formel 3: Amdahlsches Gesetz

$S$ – Gesamtbeschleunigung  
$P$ – Anteil der Laufzeit der parallelen Teilstücke eines Programmes  
$N$ – Anzahl der Prozessoren  
$o(N)$ – Kommunikation- und Synchronisierungskosten, die mit $N$ zusammenhängen

Mit dem Gesetz von Amdahl werden nur pessimistische Aussagen getroffen [Shi96]. Einige positive Faktoren, wie beispielsweise eine mögliche Beschleunigung durch Verwendung größerer Cache werden dabei nicht berücksichtigt.

### 2.1.2 Gustafsons Gesetz

Fast 20 Jahre nach dem Amdahlschen Gesetz wurde von Gustafson ein weiteres Gesetz veröffentlicht, das sich mit der Leistungsschätzung zur Parallelisierung beschäftigt [Gus88]. Hier wird die zu erreichende Beschleunigung abgeschätzt, indem die Zeit festgesetzt und die Menge der Arbeit abgeschätzt wird, die in dieser Zeit gemacht werden kann, wenn die Aufgabe parallel bearbeitet wird. Dabei werden die gleichen Eigenschaften wie bei Amdahl verwendet: die Anzahl der Prozessoren $N$ und der parallele Anteil $P$.

$$S = (1 - P) + P \cdot N$$

Formel 4: Gustafsons Gesetz


Abbildung 1: Amdahl- und Gustafson-Gesetz im Vergleich
Beide Gesetze machen zwar sehr allgemeine Aussagen über die erzielbare Beschleunigung, sie können aber trotzdem als Grundlage für genauere Schätzungen verwendet werden. Dabei stellte sich heraus, dass Gustafsons Gesetz in der Regel optimistischere Schätzungen macht als das Gesetz von Amdahl [Shi96].

2.2 Parallele Softwareentwicklung


2.2.1 Parallelisierungsprozess

Wie [MSM04] zeigt, kann der Parallelisierungsprozess in die vier folgenden Phasen unterteilt werden:

- **Partitionierung**: Eine logische Teilung einer Aufgabe in parallel ausführbare Methoden oder Fäden.

- **Kommunikation**: Festlegung der Kommunikation zwischen den Fäden. Es wird auch entschieden wie einzelne Fäden auf gemeinsame Ressourcen zugreifen sollen und welche Schutzmechanismen dabei verwendet werden.

- **Agglomeration**: Zusammenfassung von Fäden zur Effizienzsteigerung. Mehr dazu siehe auch in dem Kapitel 3.2.

- **Prozessorzuordnung**: Zuordnung der Fäden zu den Prozessoren. Dieser Schritt wird meistens durch die Ausführungsumgebung oder das Betriebssystem realisiert.
Aufgabenparallelität

Es werden Aufgaben definiert, die parallel ausgeführt werden müssen und es wird versucht, diese möglichst voneinander unabhängig auszuführen. Auf diese Weise funktioniert zum Beispiel ein Webserver, der für verschiedene Nutzer einzelne threads startet. Die voneinander unabhängigen Aufgaben können problemlos parallel ausgeführt werden und sind somit sehr gute Kandidaten für eine Parallelisierung. Auch einzelne Schleifeniterationen, die nur mit lokalen Daten arbeiten, können leicht parallelisiert werden.

Datenparallelität

Abbildung 3: Sequenzdiagramme für die Veranschaulichung des Datenparallelismus am Beispiel einer Suchfunktion.

2.2.2 Parallele Entwurfsmuster
Einige klassische Entwurfsmuster lassen sich für die parallelen Anwendungen gut verwenden, es sind aber auch spezielle parallele Entwurfsmuster entstanden [LL10]. Hier werden einige von ihnen näher betrachtet, weil viele verwandte Arbeiten sie als Grundlage für die Leistungsschätzung verwenden.

Fork-Join-Muster

**Teile und herrsche (Divide-and Conquer)**

Fließband (Pipeline)

Abbildung 6: Verschiedene Fließbandarten aus [Pankratius11]

Map/Reduce
Weitere parallele Entwurfsmuster werden in [MSM04], [Huck10] und [RV+07] näher betrachtet.

2.3 Common Language Infrastructure (CLI)
Viele moderne Programmiersprachen und Umgebungen bieten spezielle Sprachkonstrukte, um die Entwicklung der parallelen Software zu erleichtern. Da diese Arbeit auf der Basis von .NET Framework der Firma Microsoft aufgebaut ist, werden in diesem Kapitel einige Klassen und Tools näher beschrieben.


Die Schlüsselkomponente des .NET Framework sind die Common Language Runtime (CLR) und ein Satz an Klassenbibliotheken, der auch als Base Class Library (BCL) bezeichnet wird. CLR ist eine virtuelle Maschine, in der die .NET-Programme ausgeführt werden. [Richter10]

Da alle .NET-Programmiersprachen die gleichen Bibliotheken verwenden, unterscheidet sich der Code eines C#-Programmes auf CLR-Ebene nicht von dem Code eines in einer anderen Sprache programmierten Programms. Das .NET Framework ist eine sehr dynamische und
schnell entwickelnde Plattform – mit der aktuellen Version 4.0 wurden viele Verbesserungen im Bereich der parallelen Verarbeitung erzielt [MSDN11a].

**Die Fäden in .NET**

Wie bereits in Kapitel 2.2 beschrieben wurde, entstehen im Programm beim Parallelisierungsprozess mehrere Fäden (engl. *threads*). Ein *thread* ist eine Sequenz von ausführenden Befehlen in einem Prozess, der auch aus mehreren *threads* bestehen kann. Ein *thread* im .NET entspricht einem *thread* unter Windows und wird durch das Betriebssystem automatisch auf den nächsten freien Prozessorkern ausgeführt.

Mithilfe von *threads* können mehrere parallele Entwurfs muster leicht implementiert werden (siehe Kapitel 2.2.2). So können in einem *master*-Faden mehrere *worker*-Fäden erzeugt werden. Auch das Fließband-Muster lässt sich damit realisieren, indem die Arbeiterfäden die Ergebnisse fertiger Fäden erhalten und weiter verwenden.

Ein *thread* ist viel leichtgewichtiger als ein Prozess und beim Kontextwechsel von *threads* werden nur Registerinhalte gewechselt, aber keine Adressräume. Trotzdem kann es für kurze Schleifen zum Beispiel ein *thread* sogar langsamer werden als sequenzielle Variante. Mehr dazu siehe in Kapitel 4.3.

![Diagramm Thread Pool in .NET 3.5](image)

Ein *Thread-Pool*-Mechanismus ermöglicht es zum einen, einzelne Fäden leichter zu verwenden und stellt zum anderen so viele Fäden zur Verfügung wie tatsächlich benötigt werden. Der *Thread Pool* arbeitet intern mit einer globalen Warteschlange (engl. *global Queue*). Hier landen alle erzeugten Fäden und werden über verfügbare Prozessorkerne verteilt. Die dabei erzeugten Kinder-Fäden werden ebenso in die globale Warteschlange aufgenommen, was jedoch Synchronisationsprobleme hervorbringen kann [MSM04].


Lehrstuhl Prof. Dr. Walter F. Tichy
2.4 Common Compiler Infrastructure (CCI)


Der Begriff .NET Metadaten bezeichnet die bestimmten Datenstrukturen, die dem CIL-Code hinzugefügt werden, um seine abstrakte Struktur beschreiben zu können. Metadaten beschreiben alle Klassen, Methoden und Attribute der Klassen, die bei der assembly bestimmt wurden, und ebenso die Klassen mit Eigenschaften.

Als assembly wird eine Menge von Daten bezeichnet, die aus .NET-Sprachen erzeugt werden. Eine assembly enthält folgende Daten:

- ein Manifest mit Metadaten, die die assembly beschreiben,
- eine Beschreibung aller in der assembly vorhandenen Datentypen,
- Microsoft IL-Code,
- weitere Ressourcen

Metadaten für eine Methode enthalten die komplette Beschreibung der Methode einschließlich seiner Klasse, seines zurückkehrenden Typs und alle Parameter dieser Methode. Leider können auf diese Ebene keine Schleifen beschrieben werden, da sie genauso wie bedingte Anweisungen aussehen.

Alle Typen, die in der CLI definiert sind, werden durch Metadaten beschrieben. Sämtliche Informationen über Typen und Methoden sind während der Laufzeit abrufbar, zudem kann auch die Struktur einer Anwendung in der Laufzeit untersucht werden.
3 VERWANDTE ARBEITEN


3.1 Estimating Parallel Performance, A Skeleton-Based Approach [LL10]

In dieser Arbeit werden Schätzungen für die Ausführungszeit der parallelen Entwurfsmuster (engl. skeleton) gemacht. Es wird der Geschwindigkeitszuwachs durch die Art des Entwurfsmusters sowie der Größe des parallelen Anteils eines Programms abgeschätzt. Dadurch sind gute Schätzungen für unbekannte Eingabegrößen oder eine unbekannte Anzahl an Prozessoren möglich.

Es werden $n$ als Eingabegröße und $p$ als Anzahl der Prozessoren festgesetzt. Die sequenzielle Ausführungszeit wird durch $T(n)$ bezeichnet. Die allgemeine Bezeichnung für die Ausführungszeit auf $p$ Prozessoren ist $T(n, p)$.

$\hat{A}(n,p)$ ist der Mehraufwand (engl. overhead), der bei einer parallelen Ausführung durch die Kommunikation zwischen einzelnen Prozessoren und anderen Faktoren entsteht.

$$ T(n, p) = \frac{T(n)}{p} + \hat{A}(n, p) $$

Formel 5: Ausführungszeit auf $p$ Prozessoren in [LL10]

Als Ziel der Arbeit nennt man die Möglichkeit gute Approximationswerte für $T(n)$ und $\hat{A}(n,p)$ zu finden. Dafür werden drei unterschiedlich parallele Entwurfsmuster separat betrachtet:

**Parallel Map**

Das „parallel map“-Muster stellt eine einfache Form des Datenparallelismus dar. Man hat zwei Listen – jedes Element einer Eingangsliste von Typ [a] wird parallel über eine Funktion (a → b) auf einem Element von Typ [b] abgebildet. Für jedes Element wird dabei ein neuer thread erzeugt. Die gesamte Ausführungszeit wird durch folgende Formel abgeschätzt:

$$ T_{parMap}(n, p) = \frac{n}{p} * T(1) + \hat{A}(n, p) $$

Formel 6: Ausführungszeit für Parallel Map

**Divide and Conquer**

Das „teile und herrsche“-Entwurfsmuster wird als typisches Beispiel für Task-Parallelismus genommen. Als Beispiel dafür nennen die Autoren den Mergesort-Sortieralgorithmus. In dieser Arbeit werden die Eingabedaten sequenziell bis zu einer bestimmten Tiefe aufgeteilt. Daraufhin werden unabhängige Worker-Prozesse erstellt, um die Sub-Bäume auf dieser Ebene zu bearbeiten.
Dadurch werden auf der $d$-ten Ebene $r^d$ Prozesse erzeugt, wobei ($r$) den Grad der Verzweigung darstellt, in Abbildung 8: Binäre Baum der Tiefe drei für „teile und herrsche“-Muster beträgt $r = 2$. Für die Eingabegröße $n$, werden $n/r^d$ Prozesse erzeugt. Hier kommt noch der Synchronisationsaufwand beim Aufteilen und Zusammenführen von Aufgaben hinzu und es könnte schwierig sein, zwischen dem Kommunikations- und dem Parallel-Zusatzaufwand zu unterscheiden. Außerdem muss $T(n)$ nicht unbedingt linear sein – $T(n) \neq l^*T(n/l)$ – aus diesem Grund wird eine neue Variable $O(n, k, p)$ eingeführt. Dies zeigt die Arbeit, die für die Aufteilung und Zusammenlegung der Aufgaben von der Größe $n$ zu der Größe $k$ gemacht werden muss.

So kann die sequenzielle Zeit auf der $r$-stelligen „teile und herrsche“ der Tiefe $d$ für die Eingabegröße $n$ wie folgt ausgedrückt werden:

$$T(n) = r^d T\left(\frac{n}{r^d}\right) + O\left(n, \frac{n}{r^d}, 1\right)$$

Formel 7: Ausführungszeit für Rekursion der Tiefe d

Für die allgemeine Ausführungszeit kommt die Summe der zusätzlichen Kosten $\tilde{A}(n,p)$ für Aufteilung hinzu und die Formel sieht dann wie folgt aus:

$$T_{\text{flat}(n,p)} = \sum_{i=0}^{d-1} r^i \tilde{A}\left(\frac{n}{r^i}, p\right) + \frac{r^d}{p} T\left(\frac{n}{r^d}\right) + O\left(n, \frac{n}{r^d}, 1\right)$$

Formel 8: Ausführungszeit für rekursives Problem

Mit dieser Formel können die parallelen Varianten von Mergesort ziemlich gut abgeschätzt werden.

**Iteration**

Als Iteration wird eine parallele *do-while*-Schleife bezeichnet. Die Ausführungszeit ist direkt abhängig von der Anzahl der Iterationen und der Ausführungsdauer einzelner Iterationen (man bezeichnet dies mit $s(n)$). Ebenso wird davon ausgegangen, dass die Arbeit gleichmäßig aufgeteilt ist und eine Schleife genau $k$ Iterationen hat. So wird die gesamte Zeit wie folgt abgeschätzt:

$$T_{\text{iter}}(n, p, k) = \frac{k}{p} s(n) + \tilde{A}(n, p)$$

Formel 9: Ausführungszeit einer Schleife

Die automatische Erkennung solcher rekursiven Muster wie „teile und herrsche“ – erweist sich als schwierig und nicht immer möglich. Demnach ist diese Strategie für die automatische
Parallelisierung weniger geeignet, solange die Muster weder sicher noch genau erkannt werden können.

Außerdem werden alle negativen sowie positiven Faktoren durch $\tilde{A}(n, p)$ ausgedrückt und sind somit nur von der Problembesgröße und der Anzahl der Prozessoren abhängig. Weitere Einflussfaktoren werden dabei nicht betrachtet.

3.2 Function Level Parallelism Driven by Data Dependencie [RV+07]


Erster Schritt – Analyse

Der erste Schritt in diese Richtung war die Parallelisierung auf Instruktionsebene (engl. instruction level parallelism - ILP) mithilfe geschickter Compiler-Strategien. Heute wird es in vollem Umfang ausgenutzt, so dass eine Weiterentwicklung dort übermäßig komplex ist und kein großes Potenzial mehr hat [RV+07].

Der weitere logische Schritt ist die Parallelisierung auf einer höheren Ebene – der Thread-Ebene (engl. thread level parallelism - TLP). Wichtig hierbei ist es, nicht nur Prozesse voneinander zu trennen, sondern auch den Datenfluss sowie existierende Datenabhängigkeiten zu beachten.


Abbildung 9: Beispiel für Call Graph aus [RV+07]

Zweiter Schritt – Parallelisierung
Wenn die vorherige Analyse geeignete Stellen für eine Parallelisierung gefunden hat, kann mit der Parallelisierung begonnen werden. Da die Datenabhängigkeiten bekannt sind und alle Funktionen nach ihren Rollen (Aufruber/Aufrufende) unterschieden werden können, wird ein weiterer Graph gebaut. Einzelne Funktionen werden in sogenannte Cluster gruppiert, sodass zwischen zwei Clustern möglichst wenig kommuniziert wird. Solche Cluster können dann parallel ausgeführt werden.

![Graph](image)

Abbildung 10: Beispiel für einen Datenflussgraphen aus [RV+07]

3.3 Automated Experimental Parallel Performance Analysis [LD02]
In dieser Arbeit wird eine automatische Leistungsanalyse vorgenommen, um alle notwendigen Parameter für die Leistungsschätzung, Lastverteilung und Optimierung zu gewinnen. Das Hauptziel des Parallelisierungsprozesses ist die Beschleunigung, sie sollte möglichst automatisch und einfach verlaufen. Dieser Ansatz fokussiert sich auf das Message-Passing-Paradigma in einem großen Rechnerbündel.

![Diagramm](image)

Abbildung 11: Parallelisierungsprozess aus [LD02]

Ein sequenzieller Algorithmus wird durch die Partitionierung und das Hinzufügen von Synchronisationsstellen parallelisiert. Die durch Parallelisierung erzielte Beschleunigung wird
analysiert, um einerseits die Leistung später vorhersagen zu können und andererseits eine effektive Optimierung von Engpässen sowie eine effiziente Lastverteilung zu ermöglichen.

### 3.4 Facilitating Performance Predictions Using Software Components [KK+07]

Die Leistungsschätzung spielt auch bei verteilten komponentenbasierten Systemen eine wichtige Rolle. Am Lehrstuhl für „Software-Design und Qualität“ (SDQ) des Karlsruher Instituts für Technologie – geleitet von Prof. Dr. Ralf H. Reussner – wird in diesem Bereich ebenfalls viel geforscht [HK+11], [KK+07].


Um diesem vorzubeugen, werden Leistungsmodelle gebaut und bewertet [KK+07].

![Abbildung 12: Leistungsabschätzung anhand von Modellen (Bild von PCM Homepage)](image)

Laut [HK+11] soll ein Leistungsmodell die Antworten auf folgende Fragen geben:

- Wie verhalten sich die Reaktionszeiten und der Durchsatz vom System unter erwarteter Arbeitsbelastung?
- Wie beeinflusst die Umsetzung von einzelnen Komponenten die gesamte Leistung?
- Wie stark wird die Leistung durch Zuteilung von Ressourcen beeinflusst?
- Wie verhält sich das System, wenn die Arbeitslast unerwartet steigt?

Diese Fragen lassen sich ebenso auf parallele Softwaresysteme übertragen. Hier spielen einzelne Methoden oder Statements die Rolle von Komponenten und besitzen dabei ähnliche Eigenschaften, wie zum Beispiel die Ausführungszeit und der Durchsatzes. Genauso wichtig ist das Identifizieren kritischer Pfade und Stellen, die die gesamte Performance beeinflussen und die durch Parallelisierung beschleunigt werden können.
4 LEISTUNGSSCHÄTZUNG ZUR PARALLELISIERUNG

In diesem Kapitel wird der Entwurf eines Werkzeugs zur Leistungsschätzung beschrieben. Es beginnt mit dessen Zielsetzung und der Definition von Anforderungen, um diese Ziele zu erreichen. Im Kapitel 4.2 werden die Methoden und Werkzeuge vorgestellt, mit deren Hilfe die definierten Ziele erreicht werden. In Kapitel 4.3 wird das Konzept des Schätzansatzes präsentiert, das exemplarisch in einem Werkzeug implementiert wurde. Abschließend fasst Kapitel 4.4 die wesentlichen Ergebnisse kurz zusammen.

4.1 Ziele und Anforderungen


In dieser Arbeit wird eine ähnliche Formel wie in [LL10] verwendet. Die Schleifen werden durch vorherige Analyse erkannt und die Anzahl der Iterationen wird bestimmt. Diese Schätzung geht aber anders als in der [LL10] vor: Dort wird die Anzahl der Iterationen k durch die Anzahl der Prozessoren p einfach geteilt und abgerundet. Es wird davon ausgegangen, dass, wenn eine parallelisierte Schleife sechs Iterationen besitzt, aber acht Prozessoren zur Verfügung stehen, diese Schleife nicht 0.75*sn durchlaufen wird, sondern 1*sn – also genau einen Zyklus. Dieser Sachverhalt ist in dieser Arbeit mitberücksichtigt. Mehr dazu in Kapitel [0].

Für die Analyse der Beschleunigung wird eine ähnliche Vorgehensweise wie in [LD02] verwendet. Wie in Kapitel 4.2 beschrieben wird, ist das Ziel der Analyse, Daten über Ausführungspfade und die Schleifen zu gewinnen, um die Engpässe zu identifizieren. Diese Daten werden auch für die Leistungsschätzung verwendet. Die Leistungsschätzung zur Parallelisierung basiert auf den Formeln von Amdahl und Gustafson, die durch diesen neuen Parameter erweitert werden. Somit liefert diese Schätzung gleiche oder bessere Ergebnisse für einzelne Programmteile (Methoden oder Schleifen) und ist immer präziser für das gesamte Programm.


Das Hauptziel dieser Arbeit ist es also, eine gute kontrollflussbasierte Leistungsschätzung zur Parallelisierung zu machen, um dem Entwickler bei der Parallelisierung zu helfen. Es werden folgende Ziele und Anforderungen dafür definiert:

Ziel 1: Präzise Leistungsschätzung

Wir wollen sequenzielle Programme untersuchen und den möglichen Leistungszuwuchs durch eine Parallelisierung möglichst genau abzuschätzen.

- **Anforderung 1.1**: Eine Erweiterung der Formeln von Amdahl und Gustafson durch Einführung neuer Parameter: Anzahl der Iterationen.
- **Anforderung 1.2**: Die Schätzung soll auf der Methodenebene passieren. Die Gesamtbeschleunigung wird als Summe der Teilbescheinigungen berechnet.
• **Anforderung 1.3:** Es soll eine Möglichkeit geben die Schätzergebnisse auf größere Datenmengen zu skalieren.

**Ziel 2: Werkzeugunterstützte Lösung**

Die Leistungsschätzung soll einen automatischen Parallelisierungsprozess unterstützen, insofern soll die Schätzung möglichst automatisch und vor dem Entwickler verborgen passieren. Um dies zu erreichen, werden folgende vier Anforderungen definiert:

- **Anforderung 2.1:** Die Codeanalyse und die Gewinnung der zur Schätzung verwendeten Daten soll vollautomatisch erfolgen.
- **Anforderung 2.2:** Die Zielplattform soll vollautomatisch ermittelt werden. Im Rahmen dieser Arbeit konzentrieren wir uns auf die Zahl der vorhandenen Prozessorkerne, da dies bei der präzisen Leistungsschätzung eine wesentliche Rolle spielt.
- **Anforderung 2.3:** Es soll einfach möglich sein die in dieser Arbeit verwendeten Werkzeuge für die Analyse und die Profilerstellung gegen andere auszutauschen.
- **Anforderung 2.4:** Trotz der Forderung, die Schätzung vor dem Entwickler zu verborgen soll es möglich sein, die automatisch ermittelten Werte wie Laufzeiten oder Anzahl an Schleifeniterationen manuell zu variieren, um die Ergebnisse skalieren zu können.

**4.2 Statische und dynamische Analyse sequenzieller Anwendungen**

Dieses Kapitel befasst sich mit der konzeptionellen Umsetzung der definierten Ziele. Dafür soll im Folgenden zunächst ein Analysebeispiel aus dem Anwendungsbereich der Bildbearbeitung eingeführt werden, anhand dessen die in diesem Kapitel definierten Analyse- und Schätzprozesse illustriert werden sollen.

In Abbildung 13 ist ein Grafikfilter *Filter()* dargestellt, der auf 10 Eingabebildern ausgeführt wird. Dieser Filter wendet stufenweise zwei Bildbearbeitungsverfahren auf ein Eingabebild an. Die *Farbkorrektur()* macht dabei die Änderung von Farbstichen, die *Invertierung()* anschließend bestimmt für jeden Bildpunkt die gegenteilige Farbe. Bereits bei diesem einfachen Beispiel werden zwei unterschiedliche Parallelisierungen erkannt, die von gängigen Ansätzen zur Leistungsschätzung nicht erkannt würden:

- Erstens könnten mehrere Bilder parallel bearbeitet werden. Im besten Fall könnte das 10-fach parallel laufen. Wenn man jedoch mehr als 10 Prozessoren besitzt, ist das gegebenenfalls nicht optimal, da die restlichen Kerne leer laufen.
- Die zweite Möglichkeit ist, die Korrekturen selbst parallel anzuwenden. Das Bild wird dabei in 1000 Teilstücke zerlegt und die einzelnen Bildbereiche werden dann parallel
bearbeitet. Die beiden Bildbearbeitungsverfahren zählen zu den Pixeloperationen und können auf jeden Bildpunkt ungeachtet seiner Nachbarn parallel angewandt.

Um das Ziel 1 einer präzisen Leistungsschätzung zu erreichen, benötigt man die Kenntnis über die Reihenfolge, die Anzahl und die Ausführungsduer von Schleifeniterationen. Außerdem müssen die Methoden, die sequentiell oder parallel ausgeführt werden und die Laufzeitverteilung zwischen einzelnen Methoden bekannt sein. Um Anforderung 1.2 zu erfüllen, muss festgestellt werden, wo die größte Laufzeit auftritt. Dies ist klassischerweise dort, wo Schleifen im Programm vorkommen.

Diese Arbeit versucht, mithilfe der Leistungsschätzung die Frage zu beantworten, an welchen Stellen sich die Parallelisierung am meisten lohnt. Eine Analyse vorhandener Datenabhängigkeiten wird nicht vorgenommen. Die resultierende Beschleunigung soll für verschiedene Parallelisierungsvarianten abgeschätzt werden können, um so dem Entwickler zu helfen, die richtige Entscheidung zu treffen. Damit kommen wir der Anforderung 2.4 nach.


Für die tatsächliche Leistungsschätzung wird der Ausführungsgraph des Programmes verwendet, der unabhängig von Werkzeugen und Sprachen ist und ebenso durch andere Analysewerkzeuge erstellt werden kann. Dies erfüllt die Anforderung 2.3. Im Folgenden werden die statischen und dynamischen Analysen genauer betrachtet.

4.2.1 Statische Analyse mit Common Compiler Infrastructure

Die statische Analyse ermöglicht, alle Pfade im Ausführungsgraphen zu analysieren, was einen Vorteil gegenüber der dynamischen Vorgehensweise darstellt. Gleichzeitig stellt dies aber auch einen Nachteil dar, da der korrekte Pfad durch die Anwendung statisch nicht exakt abgebildet werden kann. Dieser ergibt sich oftmals unmittelbar aus Eingabedaten, die aber erst zur Laufzeit feststehen. Ein weiterer Vorteil der statischen Analysen ist, dass sie vor der Ausführung der Anwendung geschieht und somit die Laufzeit der analysierten Anwendung nicht beeinflusst.

Viele Arbeiten, unter anderem [Tou2009], [Rul2007] und [HotPar11], verwenden die statische Analyse, um die Ausführungsgraphen das Programms abzubilden und die Kommunikation zwischen den einzelnen Methoden festzustellen. Um die Ergebnisse der statischen Analyse zu überprüfen und zu verfeinern, wird diese oft mit der dynamischen Analyse kombiniert.

Um Ziel 2 zu erreichen und Anforderung 2.1 zu erfüllen, verwendet diese Arbeit ebenfalls statische und dynamische Aspekte. Zur statischen Analyse und zur Ermittlung des Ausführungsgraphen wird CCI [2.4] verwendet. Dabei werden zuerst alle Klassen, Methoden
und Schleifen erfasst. Anschließend werden die dynamischen Aspekte Ausführungszeit und Anzahl der Schleifeniterationen durch die dynamische Analyse ergänzt.

Im Folgenden wird der Aufbau des Ausführungsgraphen erklärt und zwei Ansätze zur Schleifenerkennung diskutiert.

**Aufbau des Ausführungsgraphen**

Für die werkzeugunabhängige Implementierung des Ansatzes wurde die .NET-Bibliothek QuickGraph benutzt, die Datenstrukturen für Graphen und Algorithmen bereitstellt [QG11]. Ein Ausführungsgraph ist unabhängig von verwendeten Analyse- und Profilerstellungswerkzeugen und kann auch mit anderen Daten aufgebaut werden. Der Entwickler besitzt dabei auch die Möglichkeit manuelle Änderungen durchzuführen. Es wird ein Abstraktionsniveau erreicht und die Anforderung 2.3 wird erfüllt.

![Abbildung 14: Statischer Ausführungsgraph für Beispiel 1](image)

Um mit der Analyse einer *assembly* zu beginnen, wird der entsprechende Pfad angegeben. Mithilfe der Methode `IAssembly.GetAllTypes()` werden zunächst alle in der *assembly* definierten Klassen zurückgeliefert. Danach bekommt man alle Methoden der Klasse, die als eine Kollektion definiert sind.


**Schleifeninstrumentierung als anonyme Methoden**

Da die Erkennung von Schleifen ein wichtiger Punkt in dieser Arbeit ist, wurde zunächst versucht, die Schleifen für den Profiler sichtbar zu machen, indem die Schleifenkörper in eine anonyme Methode verpackt wurden. Vorher sieht man nur die Methode selbst und keine Schleifen innendrin:

![Abbildung 15: Normale Schleife im Profiler](image)
Der Schleifenkörper kann auch in eine anonyme Methode verpacken und in der Schleife dieser Methode aufrufen. Die Abbildung 16 zeigt die Implementierung davon:

```csharp
static int methode()
{
    int sum = 0;
    for (int i = 0; i < MAX; i++)
    {
        sum += i * i;
    }
    return sum;
}
```

```csharp
delegate void sumMethode(int i);
static int methode()
{
    int sum = 0;
    sumMethode x = delegate(int i)
    {
        sum += i * i;
    };
    for (int i = 0; i < MAX; i++)
    {
        x(i);
    }
    return sum;
}
```

Abbildung 16: Schleifenkörper als anonyme Methode


So können indirekt die Schleifen erkannt und die Anzahl der Iterationen (NumCalls) ablesen werden:

Abbildung 17: Schleifenkörper in Methode enthalten

Die Methoden, und somit die Schleifen, werden zwar erkannt, aber wie später, durch dynamische Analyse festgestellt wurde, verändern sich die Laufzeiten und das Programm wird generell viel langsamer. Diese Nachteile führten dazu, dass eine andere Vorgehensweise gewählt wurde.

**Attributbasiertes Erkennen von Schleifen**

Da eine generische Schleifenerkennung im Rahmen dieser Studienarbeit nicht möglich ist und die Schleifeninstrumentierung als anonyme Methoden einige Nachteile gezeigt hat, hat man entschieden die Methoden-Attribute für die Erkennung der Schleifen in C# verwendet.

Folgende Klasse definiert ein spezielles Methoden-Attribut, um Methoden mit Schleifen zu markieren:

```csharp
[AttributeUsage(AttributeTargets.Method, Inherited = true, AllowMultiple = true)]
public class LoopAttribute : System.Attribute
{
    private string name;
    private int startLine;
    private int endLine;
    private int numCalls;

    public LoopAttribute(string _name, int startLine, int endLine, int _numCalls)
    {
        this.name = _name;
        this.startLine = startLine;
        this.endLine = endLine;
        this.numCalls = _numCalls;
    }
}
```

**Abbildung 18: Definition einer Attributklasse**

Der Ausdruck, der innerhalb der eckigen Klammern steht, ist nichts anderes als der Aufruf eines Konstruktors dieser Klasse. Eine Methode kann auch mehrere Schleifen enthalten. In diesem Fall kann ein Attribut auch mehrmals vorkommen (AllowMultiple=true).

Die Verwendung des Attributes sieht folgendermaßen aus:

```csharp
[LoopAttribute("for", 37, 41, 1000)]
public static void master(int count)
{
    for (int i = 0; i < count; i++)
    {
        worker1();
        worker2();
    }
}
```

**Abbildung 19: Anwendung von Attributen**

In der hier vorgeschlagenen Implementierung muss der Entwickler die Methoden, die Schleifen enthalten, entsprechend markieren. In diesem Beispiel muss auch die Schleifenvariable `count` im Attribut angegeben werden. Das ist nicht immer möglich und soll eigentlich durch die dynamische Analyse ersetzt werden. Auch andere Schleifentypen wie `while` – bei dem die Abbruchbedienung durch eine boolesche Variable definiert werden kann – lassen sich auf diese Weise nur schlecht analysieren, weil sie oft nur zur Laufzeit bekannt werden.

Es empfiehlt sich, ein spezielles Werkzeug für die Erkennung und Analyse der Schleifen zu verwenden. Im Rahmen einer Studienarbeit von Evgeny Selyansky in dieser Forschungsgruppe wurde ein solches Werkzeug zur Identifikation von Schleifen entwickelt.

### 4.2.2 Dynamische Analyse mit dem Microsoft Visual Studio Profiler

Nun soll die tatsächliche Ausführungsreihenfolge, sowie die Laufzeit und Anzahl der Schleifeniterationen bestimmen werden. Dafür wird das ausgewählte Programm (*assembly*) durch Microsoft Visual Studio Profiler gestartet und ausgeführt. Dabei werden alle Ereignisse wie zum Beispiel die Methodenaufrufe, und die Laufzeit mitgezählt und in eine Profildatei
gespeichert. Aus dieser Profildatei, die mehrere Megabyte groß sein kann, werden spezielle Profile erstellt.

Ein Profil enthält folgende Laufzeitdaten:

- *ModuleName* – Name des Programms oder Bibliothek
- *FunctionName* – Methodenname
- *LineNumber* – Codezeile, wo diese Methode beginnt
- *NumCalls* – Anzahl der Aufrufen
- *InclusiveElapsedTimePercent* – Zeigt wie lange (in %) hat diese Methode selber gearbeitet
- *ExclusiveElapsedTimePercent* – Zeigt wie lange (in %) haben andere, von dieser Methode aufgerufene Methoden gearbeitet

Die drei folgenden Eigenschaften können dafür verwendet werden, die Verteilung und damit verbundene Fehlerwahrscheinlichkeit zu bestimmen:

- *MinInclusiveElapsedTime* – minimale Arbeitszeit
- *AvgInclusiveElapsedTime* – mittlere Arbeitszeit
- *MaxInclusiveElapsedTime* – maximale Arbeitszeit

Wenn eine Methode bei mehreren Aufrufen immer die gleiche Laufzeit gezeigt hat, so kann davon ausgegangen werden, dass die Varianz ziemlich klein ist und die Schätzung ziemlich präzise wird. Die Testläufe haben gezeigt, dass die kleinen Methoden, die wenig Zeit arbeiten, aber oft aufgerufen werden, die Varianz größer ist als bei der großen Methoden. Hier spielt der Prozess-Scheduler auch eine Rolle, sowie andere laufende Prozesse. Es empfiehlt sich somit für die Testläufe alle anderen Programmen abzuschalten.

Solch eine dynamische Analyse kann ziemlich lange dauern (um Faktor 10-100 langsamer, als der normale Programmablauf) und ist ziemlich ressourcenaufwendig. Es empfiehlt sich daher, eine kleine Datennenge dafür zu verwenden. Die Ergebnisse können dann auf größere Datennmenge skaliert werden. Mehr dazu in Kapitel 4.3.3.

**Caller-Callee und CallTree-Profile**

Für unser Beispiel liefert *Caller-Callee*-Profile die Daten darüber, welche Methode der Aufrufer ist und welche aufgerufen werden? Auch die Anzahl der Aufrufe (*NumCalls*) ist hier zu sehen. Mit dieser Information kann der Aufrufgraph vervollständigt werden. Außerdem erkennt man, welche Pfade inaktiv bleiben und welche durch die Eingabevariable beeinflusst werden.

Die Master-Methode (hier als Root bezeichnet) ruft weitere Methoden auf:

```
<CallerCallee CallerCalleeType="Root" RootFunctionName="TestForProfiler.Program.master"
FunctionName="TestForProfiler.Program.master" ... />
```

```
<CallerCallee CallerCalleeType="Callee"
RootFunctionName="TestForProfiler.Program.master"
FunctionName="TestForProfiler.Program.worker1" NumCalls="1.000" ... />
```

```
<CallerCallee CallerCalleeType="Callee"
RootFunctionName="TestForProfiler.Program.master"
FunctionName="TestForProfiler.Program.worker2" NumCalls="20" ... />
```

Abbildung 20: CallerCallee-Profil
Eine weitere Möglichkeit für die Anordnung von Methoden und der Aufbau des Kontrollflusses ist die Verwendung von CallTree-Profil. Bei einem CallTree-Profil werden alle Methoden in einer Liste abgespeichert und anhand eines Level-Attributs kann die Tiefe erkannt werden. Die NumCalls werden in diesem Fall auch entsprechend summiert.

Beide Profile enthalten zudem prozentuelle Laufzeitinformationen: zum einen, wie lange diese Methode selbst aktiv war, und zum anderen, wie lange von ihr aufgerufene weitere Methoden aktiv waren. Diese Zeitangaben bilden die Grundlage für die weitere Schätzung und spielen dabei eine große Rolle. Da bekannt ist, welche Methode wie lange gearbeitet hat und wie oft sie aufgerufen wurde, kann nun die Leistungsschätzung zur Parallelisierung von Methoden erfolgen.

In Beispiel 1 können jetzt auch die Methoden erkannt werden, die nicht ausgeführt werden, wie zum Beispiel Kantenhervorhebung() in der Abbildung 22: Dynamischer Ausführungsgraph mit Schleifen.

### 4.2.3 Aufbau des Ausführungsgraphen

Die durch Attribute markierten Schleifen werden als selbstständige Knoten im Graphen dargestellt und besitzen die gleichen Eigenschaften wie normale Methoden. Es werden die Methoden aus dem Graphen entfernt, die im Code vorhanden sind, aber nicht ausgeführt werden.

Wir setzen außerdem eine untere Schranke und entfernen die Methoden, die weniger als 1% der Laufzeit aktiv sind. Diese Größe kann variabel oder abhängig von anderen Faktoren wie zum Beispiel Anzahl der Kernen \( N \) gemacht werden. Denn, wenn man viele Kerne hat, können auch die kleinen Methoden attraktive Parallelisierungskandidaten sein, wenn man zum Beispiel nicht nur äußere Schleifen, sondern auch die Inneren parallelisiert werden.

![Abbildung 22: Dynamischer Ausführungsgraph mit Schleifen](image_url)
Die durch dynamische Analyse gemessenen Werte sind auch in einer Tabelle zusammengefasst und lassen sich durch den Entwickler manuell ändern. So kann man die Eingaben schnell variieren um andere Schätzungen zu machen.

4.2.4 Analyse der Zielplattform

Im Rahmen dieser Arbeit wird eine automatische Analyse der Zielplattform durchgeführt um die Anzahl der Prozessorkerne zu bestimmen und somit die Anforderung 2.2 zu erfüllen.

Die einfachste Möglichkeit Informationen über Hardwareumgebung im .NET Framework zu bekommen ist die Klasse Environment. Diese Klasse enthält statische Methoden und Eigenschaften, die Informationen über die aktuelle Hardware-Umgebung und der Software-Plattform liefern.

Die Eigenschaft ProcessorCount liefert die Anzahl der Prozessoren zurück, die für das System sichtbar sind. Allerdings wird hier zwischen echten und logischen Prozessoren nicht unterschieden. Für die Prozessoren, die Hyper-Threading Technology verwenden, werden somit doppelt so viele Kerne angezeigt, als sie tatsächlich haben. Da die Beschleunigung für solche Systeme ziemlich schwer vorhersehbar ist und von vielen anderen Faktoren abhängt, soll nur die tatsächliche Anzahl der Prozessoren berücksichtigt werden.

Um die tatsächliche Anzahl der Prozessoren zu erfahren, wird in dieser Arbeit das Windows Management Instrumentation (WMI) Rahmenwerk verwendet. Die WMI-Klasse Win32_Processor liefert ausführliche Information über den im System verwendeten Prozessor. Eine Instanz dieser Klasse enthält folgende Eigenschaften:

- NumberOfCores – Anzahl der Prozessorkerne
- NumberOfLogicalProcessors – Anzahl der logischen Prozessoren

Somit kann zwischen logischen und physischen Prozessoren unterschieden und der Anforderung 2.2 nachgekommen werden.

4.3 Das Schätzverfahren LoopEst


Um die in Kapitel 4.1 definierten Ziele zu erreichen und eine präzise Leistungsschätzung zu machen, werden die Formeln von Amdahl und Gustafson durch einen neuer Parameter: Anzahl der Iterationen erweitert. Es werden drei mögliche Fälle betrachtet und unterschiedlich abgeschätzt. Somit wird die möglich Beschleunigung präziser abgeschätzt werden und der Anforderung 1.1 nachgekommen. Außerdem werden in dieser Arbeit die einzelnen Methoden und Statements [0] separat abgeschätzt und der gesamte Leistungszuwachs durch Parallelisierung wird als eine partielle Summe berechnet [4.3.2]. Dadurch werden Anforderungen 1.1 und 1.2 erfüllt.

Um der Anforderung 1.3 nachzukommen, wird in dieser Arbeit ein Ansatz vorgestellt, der die Leistungsschätzung durch Skalierung auf größere Datenmengen ermöglicht. Es wird ein Skalierungsbeispiel eingeführt, das zeigt, wie sich sequenzielle und parallele Programmanteile dabei verändern, und welchen Einfluss diese Veränderung auf die Beschleunigung hat.

Im Folgenden werden das LoopEst und die neue Formeln näher betrachtet.
4.3.1 Schätzung von Schleifen und Methoden


Die Testläufe haben allerdings gezeigt, dass bei der Parallelisierung der Schleifen ein zusätzlicher Parameter die Beschleunigung stark beeinflusst: die Anzahl der Iterationen (NumCalls) und sein Verhältnis zu N. Wenn alle Pfade gleich lang sind und keine dynamische Verteilung von Aufgaben stattfindet, können drei Fälle unterschieden und jeder für sich separat abgeschätzt werden.

1. Fall: \( k < n \)

Ist die Anzahl der Iterationen kleiner als die Anzahl der Prozessoren, so wird die Schleife in einem „Takt“ abgearbeitet und man kann \( N = \text{NumCalls} \) setzen.

\[
S = (1 - P) + P \cdot (\text{NumCalls})
\]

Formel 10: Leistungsschätzung mit NumCalls

2. Fall: \( k > n \) und \( k \% n = 0 \)

In diesem Fall wird die Schleife in NumCalls/N Iterationen abgearbeitet und für die Abschätzung der Leistung kann das klassische Gustafsons-Gesetz angewendet werden:

\[
S = (1 - P) + P \cdot N
\]

Formel 11: Leistungsschätzung für den Fall 2

3. Fall: \( k > n \) und \( k \% n \neq 0 \)

Falls NumCalls nicht restlos durch N teilbar ist, bleiben noch X Iterationen (\( 1 \leq X < N \)) übrig. Diese werden auch auf einmal abgearbeitet, wobei die Beschleunigung für \( X_1 \neq X_2 \) unterschiedlich ist.

![Abbildung 23: Schleifenschema](image)

Die Ausführungszeit für zehn Iterationen entspricht der Ausführungszeit für zwölf Iterationen, aber die Menge der geleisteten Arbeit und damit die Beschleunigung sind unterschiedlich. Es wird definieren:

\[
D = \text{NumCalls} \mod N
\]

\[
P_{\text{normal}} = P \cdot \left(1 - \frac{D}{\text{NumCalls}}\right); \quad P_{\text{delta}} = P - P_{\text{normal}}
\]

\[
S = (1 - P) + (P_{\text{normal}} \cdot N) + (P_{\text{delta}} \cdot D)
\]

Abbildung 24: Herleitung der Formel
\( P_{\text{normal}} \) ist der Zeitanteil mit „vollen“ Iterationsblöcken und \( P_{\text{delta}} \) ist kleiner Rest. Wie erkennbar ist, wird \( P_{\text{delta}} \) mit steigenden \textit{NumCalls} immer kleiner und man nähert sich dem Fall 2.

Abbildung 25: Laufzeiten für sequenzielle und parallele Variante mit \( N=4 \)

Die Unterschiede zwischen klassischen Formeln und diesem Ansatz sind insbesondere groß für kleine \textit{NumCalls}-Werte und große \( P \). Dieses Verfahren ist besser als klassische Formeln bei hohem parallelen Anteil und geringem Anzahl der Iterationen einer Schleife. Damit kann die Leistungsschätzung nicht nur für Schleifen, die oft viel mehr Iterationen als Anzahl an Prozessoren haben, aber auch für die einzelnen Tasks oder Worker-Threads gemacht werden.

4.3.2 Leistungsschätzung des gesamten Programms

Eine Leistungsschätzung für das gesamte Programm anhand von \( P \) und \( N \) ist zwar möglich, dies zeigt aber nur grobe Unter- und Obergrenzen für einen möglichen Leistungszuwachs durch die Parallelisierung. Die Gesetze von Amdahl und Gustafson zeigen keinen Unterschied zwischen zwei Programmen mit jeweils vier und fünf Arbeiterfäden auf einem Vier-Kern-Prozessor, obwohl ein großer Beschleunigungsunterschied vorliegt.

Um Ziel 1 zu erfüllen, wird in dieser Arbeit anders vorgegangen: Die Parallelisierungskandidaten, die vorher durch die Analyse bestimmt wurden, werden separat abgeschätzt und die einzelne Beschleunigungsraten anschließend gewichtet und zusammenaddiert.

Folgende Formel illustriert den Prozess:

\[
S_{\text{gesamt}} = \sum S_m * G_m
\]

Abbildung 26: Gesamtbescheinigung als die Summe

\( S_{\text{gesamt}} \) – Gesamte Beschleunigung  
\( S_m \) – Beschleunigung einzelner Methoden  
\( G_m \) – Gewichtsfaktor einer Methode

Die Gewichtung einzelner Methoden kann durch die dynamische Analyse bestimmt werden. Wir verwenden die Laufzeiten einzelner Methoden als Gewichtsfaktor. Dabei ist es wichtig zu beachten, dass für andere Problemgrößen die Verhältnisse konstant bleiben. Es empfiehlt sich mehrere kleine Beispiele zu analysieren, um die Skalierungsfaktoren zu bestimmen.
4.3.3 Schätzung durch Skalierbarkeit

Die Skalierbarkeit ist ein weiterer Punkt. Da man oft zuerst kleinere Probleme analysiert und dadurch versucht, die Schätzung auf größere Probleme zu übertragen. Es ist sehr wichtig das richtige Verhältnis beizubehalten.


Mit wachsendem N nimmt gleichzeitig die Beschleunigung zu und wächst linear. Wie jedoch bekannt ist, kann der parallele Teil nicht auf beliebig kleine Abschnitte zerlegt werden und ist zumindest auf Instruktions-Ebene immer noch deterministisch.

Da \( N \) – aus unserer Sicht – gleich der Anzahl der Prozessoren im Rechner ist, bleibt diese Größe unabhängig von der Eingabegröße und demnach konstant. Wir haben die Formel mit \( NumCalls \) erweitert, da \( NumCalls \) eine Größe ist, die sich sehr wohl skalieren lässt. Werden im Programm zwei Matrizen oder Vektoren miteinander multipliziert, ist \( NumCall \) gleich der Größe dieser Objekte oder deren Anzahl. Ändert sich \( NumCall \), so wird der parallele Teil des Programmes (P) auch größer und der sequentielle (1-P) Teil verkleinert sich dementsprechend.

Ein Skalierungsbeispiel


Es wird angenommen, dass für die Filmgröße X beide Teilaufgaben gleich sind, also \( P = (1-P) = 50\% \). Bei einem doppelt so langem Film entspricht das Verhältnis: \( P = 66\% \) und \( (1-P) = 33\% \).

Wir berechnen \( P_{\text{neu}} \) folgendermaßen:

\[
P_{\text{neu}} = \frac{P_{\text{alt}} \times \text{NumCallsFactor}}{(1 - P_{\text{alt}}) + p \times \text{NumCallsFactor}}
\]

Abbildung 27: Leistungsschätzung durch Skalierung

Dabei steht der \( \text{NumCallsFactor} \), ein Umrechnungsfaktor, für die alte und neue Anzahl der Iterationen. Somit kann die Anforderung 1.3 erfüllt werden.

Diese Tabelle zeigt wie sich die Verhältnisse dabei ändern:

<table>
<thead>
<tr>
<th>NumCalls</th>
<th>P</th>
<th>NumCalls</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50%</td>
<td>11</td>
<td>92%</td>
</tr>
<tr>
<td>2</td>
<td>67%</td>
<td>12</td>
<td>92%</td>
</tr>
<tr>
<td>3</td>
<td>75%</td>
<td>13</td>
<td>93%</td>
</tr>
<tr>
<td>4</td>
<td>80%</td>
<td>14</td>
<td>93%</td>
</tr>
<tr>
<td>5</td>
<td>83%</td>
<td>15</td>
<td>94%</td>
</tr>
<tr>
<td>6</td>
<td>86%</td>
<td>16</td>
<td>94%</td>
</tr>
<tr>
<td>7</td>
<td>88%</td>
<td>17</td>
<td>94%</td>
</tr>
<tr>
<td>8</td>
<td>89%</td>
<td>18</td>
<td>95%</td>
</tr>
<tr>
<td>9</td>
<td>90%</td>
<td>19</td>
<td>95%</td>
</tr>
<tr>
<td>10</td>
<td>91%</td>
<td>20</td>
<td>95%</td>
</tr>
</tbody>
</table>

Tabelle 1: Änderung von P durch Skalierung
Im Kapitel 5.1 wird diese empirisch bestimmte Formel anhand eines Beispiels evaluiert und es wird gezeigt, dass die Ergebnisse realistisch sind.

### 4.4 Zusammenfassung

In diesem Kapitel wurden die Instrumentierung, die Analyse sowie der Schätzungsansatz beschrieben. Anschließend wurde eine Formel für die Leistungsschätzung durch Skalierung vorgestellt und anhand eines Beispiels verdeutlicht.


Die folgende Tabelle fasst die Gemeinsamkeiten und die Unterschiede zwischen dieser Arbeit und den anderen Arbeiten zusammen:

<table>
<thead>
<tr>
<th>Arbeit / Ansatz</th>
<th>Gemeinsamkeiten</th>
<th>Unterschiede</th>
</tr>
</thead>
</table>
| [Amd67], [Gus88] | Gleiche Basisformel | 1. Erweiterung der Formel durch zusätzliche Parameter $NumCalls$  
2. Anwendung auf Methoden-Ebene und nicht auf das gesamte Programm |
| [LL10]          | Ähnliche Formel für die Schätzung von Schleifen | 1. Es werden keine Entwurfsmuster betrachtet, sondern der Kontrollfluss allgemein  
2. Die Formel berücksichtigt $NumCalls$ |
| [RV+07], [LD02] | 1. Statische und dynamische Analyse für die Ermittlung des Kontrollflusses und der Aufbau des Ausführungsgraphen  
2. Arbeit auf der Methodenebene | 1. Die Datenabhängigkeiten werden nicht berücksichtigt  
2. Es findet keine Parallelisierung statt, sondern es werden nur die besten Kandidaten (Schleifen oder Methoden) dafür bestimmt |

Tabelle 2: Ähnlichkeiten und Unterschiede mit anderen Arbeiten

Die Schätzungsverfahren in dieser Arbeit beruhen auf einer Reihe von Annahmen, die sich auch kritisch hinterfragen lassen. Zum einen wird eine vorherige Instrumentalisierung durch den Entwickler benötigt, um die Schleifen zu markieren. Das bedeutet, dass die Anforderung 2.1 nur
teilweise erfüllt werden kann. Dabei sollen die Iterationszahlen im Vorfeld bereits bekannt sein – was nicht immer der Fall ist.

Zum anderen wird der vorhandene Code unter der Annahme abgeschätzt, dass die Schleifen parallelisiert werden können und dabei keine Datenabhängigkeiten existieren. In dem Fall, dass die Datenabhängigkeiten vorkommen, kann die Schätzung trotzdem durchgeführt werden, wenn eine Reduktionsoperation möglich ist.
5 EVALUIERUNG

Für die Entwicklung unseres Ansatzes haben wir verschiedene kleine Anwendungen geschrieben, getestet und deren Verhalten analysiert. In diesem Kapitel werden die Evaluationsergebnisse von diesen Anwendungen vorgestellt und diskutiert.


Zuerst wird eine kurze Beschreibung jeder zu analysierenden Anwendung gegeben und anschließend das Ergebnis der Leistungsschätzung, das vom Werkzeug geliefert wurde, präsentiert. Die Ergebnisse werden schließlich manuell geprüft und bewertet. Als Testumgebung wird ein Computer mit vier Kernen (mit Quad-Core Q6600 Intel-Prozessor) und 2x2Gb Dual-Channel DDR2 Speicher verwendet.

5.1 Eigene Beispiele

Im ersten Beispiel wird ein Bildfilter auf zehn verschiedenen Bildern angewendet. Der Filter besteht aus zwei Stufen, die in sich weitere Schleifen besitzen.

Für die Evaluierung wurden alle drei Parallelisierungsmöglichkeiten betrachtet und die Laufzeiten der sequenziellen und parallelten Varianten gemessen. Jede Variante wurde 30-mal gestartet und danach ein arithmetischer Mittelwert daraus berechnet.

Folgende Tabelle zeigt die Ergebnisse. Die dritte Spalte \( S(\text{methode}) \) zeigt die Beschleunigung für die ausgewählte Methode. Die \( S(\text{gesamt}) \)-Spalte zeigt die Beschleunigung für das Gesamtprogramm. \( S(\text{real}) \) ist die tatsächlich gemessene Beschleunigung und Gus88 die Schätzung durch das Gustafson-Gesetz.

<table>
<thead>
<tr>
<th>Methode</th>
<th>( P )</th>
<th>NumCalls</th>
<th>( S(\text{methode}) )</th>
<th>( S(\text{gesamt}) )</th>
<th>( S(\text{real}) )</th>
<th>Gus88</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter(Bild)</td>
<td>98%</td>
<td>10</td>
<td>295%</td>
<td>291%</td>
<td>290%</td>
<td>394%</td>
</tr>
<tr>
<td>Farbkorrektur</td>
<td>59%</td>
<td>1000</td>
<td>228%</td>
<td>175%</td>
<td>131%</td>
<td>277%</td>
</tr>
<tr>
<td>Invertierung</td>
<td>36%</td>
<td>1000</td>
<td>170%</td>
<td>125%</td>
<td>105%</td>
<td>208%</td>
</tr>
</tbody>
</table>

Je kleiner die Methode ist, die parallelisiert wird, umso größer sind die Schätzungsfehler (siehe Abbildung 28). Trotzdem sind die Ergebnisse besser als eine allgemeine Schätzung durch Gustafson-Gesetz.

\[ \text{Abbildung 28: Evaluierung von Beispiel 1} \]
Skalierungsbeispiel

Im Kapitel 4.3.3 wurde eine Formel präsentiert, die Leistungsschätzung durch Skalierung ermöglicht. Wenn man eine Leistungsschätzung für größere Datenmengen abgeben möchte, muss dies unbedingt mitberücksichtigt werden. Es wurde ein kleines Beispiel geschrieben, das eine ähnliche Verhalten simuliert und mit unserem Werkzeug evaluiert.

Das Programm hatte \( P = (1 - P) = 50\% \) und eine relativ kurze Laufzeit. Danach wurde das Beispiel stufenweise bis auf Faktor 20 skaliert. Hierdurch sieht man den Vergleich zwischen dem tatsächlichen Leistungszuwachs \( S \) und dem geschätzten \( S_{\text{geschätzt}} \). Der Test wurde auf einer Quad-Core Intel Q6600 Maschine durchgeführt.

<table>
<thead>
<tr>
<th>NumCalls</th>
<th>( T(1) ), ms</th>
<th>( T(N) ), ms</th>
<th>( S )</th>
<th>( P )</th>
<th>( S_{\text{geschätzt}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>202</td>
<td>236</td>
<td>86%</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>305</td>
<td>238</td>
<td>128%</td>
<td>67%</td>
<td>158%</td>
</tr>
<tr>
<td>3</td>
<td>398</td>
<td>239</td>
<td>167%</td>
<td>75%</td>
<td>225%</td>
</tr>
<tr>
<td>4</td>
<td>495</td>
<td>234</td>
<td>212%</td>
<td>80%</td>
<td>295%</td>
</tr>
<tr>
<td>5</td>
<td>590</td>
<td>295</td>
<td>200%</td>
<td>83%</td>
<td>250%</td>
</tr>
<tr>
<td>6</td>
<td>693</td>
<td>302</td>
<td>229%</td>
<td>86%</td>
<td>267%</td>
</tr>
<tr>
<td>7</td>
<td>785</td>
<td>315</td>
<td>249%</td>
<td>88%</td>
<td>296%</td>
</tr>
<tr>
<td>8</td>
<td>888</td>
<td>332</td>
<td>267%</td>
<td>89%</td>
<td>333%</td>
</tr>
<tr>
<td>9</td>
<td>996</td>
<td>391</td>
<td>255%</td>
<td>90%</td>
<td>295%</td>
</tr>
<tr>
<td>10</td>
<td>1081</td>
<td>407</td>
<td>266%</td>
<td>91%</td>
<td>306%</td>
</tr>
<tr>
<td>11</td>
<td>1188</td>
<td>420</td>
<td>283%</td>
<td>92%</td>
<td>325%</td>
</tr>
<tr>
<td>12</td>
<td>1290</td>
<td>437</td>
<td>295%</td>
<td>92%</td>
<td>351%</td>
</tr>
<tr>
<td>13</td>
<td>1399</td>
<td>527</td>
<td>265%</td>
<td>93%</td>
<td>319%</td>
</tr>
<tr>
<td>14</td>
<td>1507</td>
<td>548</td>
<td>275%</td>
<td>93%</td>
<td>327%</td>
</tr>
<tr>
<td>15</td>
<td>1554</td>
<td>565</td>
<td>275%</td>
<td>94%</td>
<td>341%</td>
</tr>
<tr>
<td>16</td>
<td>1651</td>
<td>574</td>
<td>288%</td>
<td>94%</td>
<td>361%</td>
</tr>
<tr>
<td>17</td>
<td>1746</td>
<td>682</td>
<td>256%</td>
<td>94%</td>
<td>333%</td>
</tr>
<tr>
<td>18</td>
<td>1854</td>
<td>697</td>
<td>266%</td>
<td>95%</td>
<td>340%</td>
</tr>
<tr>
<td>19</td>
<td>1946</td>
<td>691</td>
<td>282%</td>
<td>95%</td>
<td>352%</td>
</tr>
<tr>
<td>20</td>
<td>2038</td>
<td>708</td>
<td>288%</td>
<td>95%</td>
<td>368%</td>
</tr>
</tbody>
</table>

Tabelle 3: Leistungsschätzung durch Skalierung

Folgende Grafik illustriert die geschätzte und reelle Beschleunigung in Abhängigkeit von der Problemgröße:
Wie aus der Abbildung erkennbar, ist die Schätzung zu optimistisch, da hier die Cache-Effekte und weitere Hardware-Faktoren nicht berücksichtigt werden. Es ist aber auch deutlich zu sehen, dass die beiden unteren Kurven etwa gleiche Charakteristika haben und es sich um konstante Unterschiede handelt. Das zeigt, dass die Annahme und die Skalierungsformeln in diesem Fall korrekt sind und bessere Schätzwerte als [Gus88] liefern.

5.2 Matrizen- und Vektormultiplikationen


Es wurde eine Klasse `Matrix` wie folgt implementiert:

```java
class Matrix
{
    private double[,] m_Array;
    private void Alloc(int rows, int cols)
    {
        Rows = rows;
       Cols = cols;
m_Array = null;
        if (Rows * Cols != 0)
            m_Array = new double[Rows, Cols];
    }
    public Matrix(int rows, int cols)
    {
        Alloc(rows, cols);
    }
    public double this[int i, int j]
    {
        get { return m_Array[i, j]; }  
        set { m_Array[i, j] = value; }
    }
}
```

Der Multiplikationsoperator für diese Klasse wurde überladen, sodass zwischen sequenziellen und parallelen Multiplikationsvarianten ausgewählt werden kann. Zunächst wurden die Laufzeiten für die sequenziellen Varianten für verschiedene Matrixgrößen gemessen. Schließlich wurde eine Schätzung für die Laufzeiten der parallelen Varianten gemacht. Die erste Spalte T(1) zeigt die sequenzielle Laufzeit. Die Spalten zwei und drei die geschätzte und die tatsächliche Laufzeit für die parallele Variante. Die parallele Laufzeitanteil P(geschätzt) wurde durch die in Kapitel 4.3.3 vorgestellte Formel berechnet.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>T(1)</th>
<th>T(4) geschätzt</th>
<th>T(4) real</th>
<th>P (geschätzt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250x250</td>
<td>375 ms</td>
<td>120 ms</td>
<td>128 ms</td>
<td>95%</td>
</tr>
<tr>
<td>500x500</td>
<td>3020 ms</td>
<td>840 ms</td>
<td>913 ms</td>
<td>97%</td>
</tr>
<tr>
<td>1000x1000</td>
<td>24050 ms</td>
<td>6725 ms</td>
<td>7185 ms</td>
<td>98%</td>
</tr>
</tbody>
</table>

**Tabelle 4: Leistungsschätzung durch Skalierung**

Lehrstuhl Prof. Dr. Walter F. Tichy
Wie in Abbildung 30: Schätzung durch Skalierung ersichtlich, ist die Schätzung durch Skalierung ebenfalls ziemlich präzise und kann erfolgreich angewendet werden.

Abbildung 30: Schätzung durch Skalierung

5.3 Beispiele für parallele Programmierung mit .NET 4

Compute PI
Die Zahl π lässt sich durch partielle Integration wie folgt berechnen:

\[
\int_0^1 \frac{4}{1 + x^2} \, dx = \pi
\]

Formel 12: Compute PI

Dabei kann das Intervall zwischen 0 und 1 in beliebig viele Teilintervalle zerlegt und jedes für sich separat berechnet werden – je mehr solcher Intervalle, desto genauer wird das Ergebnis. Da die Berechnungen voneinander unabhängig sind, lassen sich die Teilintervalle parallel berechnen.

Die sequenzielle Variante der Berechnung sieht folgendermaßen aus:

```csharp
static double SerialPi()
{
    double sum = 0.0;
    double step = 1.0 / (double)num_steps;
    for (int i = 0; i < num_steps; i++)
    {
        double x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    return step * sum;
}
```

Die parallele Berechnung unterscheidet sich nur dadurch, dass die for-Schleife durch ihre parallele Variante ersetzt wird.

Die Variable `num_steps` (Anzahl der Schritte) wird definiert gleich 100.000.000 und es wird zunächst die sequentielle Variante berechnet. Die Laufzeit beträgt dabei 2535 Millisekunden. Es handelt sich um einen perfekten Parallelisierungskandidaten. Wenn man nun naiv vorgeht, kann man die Laufzeit für die parallele Variante auf ein Viertel davon abschätzen (also ca. 633 ms).
Wird aber zunächst eine dynamische Analyse durchgeführt, so wird festgestellt, dass der Laufzeitanteil für die Schleife nur 71,5% beträgt. Dies bedeutet, dass deutlich weniger als eine vierfache Beschleunigung erreicht wird.

Abbildung 31: Copute PI, Dynamische Analyse und Laufzeitverteilung

Wie in Abbildung 31: Copute PI, Dynamische Analyse und Laufzeitverteilung zu sehen ist, wird die Schleife ihre Laufzeit mittels einer Parallelisierung von 1812 ms. auf 840 ms. reduzieren (geschätzt vom Werkzeug). Die gesamte Laufzeit des Programms verbessert sich von ca. 972 (1812 - 840) ms. auf 1563 ms.

Die tatsächlich gemessene Laufzeit beträgt 1540 ms. In diesem Fall hat unser Werkzeug eine sehr gute Schätzung gemacht. Gleichzeitig bestätigt dieses Beispiel die These, dass ein Programm nie vollständig parallelisiert werden kann und dies bei der Abschätzung der Laufzeit unbedingt beachtet werden muss.

BabyNames

In diesem Beispiel wird zunächst eine große Liste (3x10⁶) mit Kindernamen zufällig generiert. Danach wird mithilfe von LINQ bzw. PLINQ für jedes Jahr in einem Zeitabschnitt ein bestimmter Name gesucht und gezählt, wie oft dieser Name in diesem Jahr vorkommt. Zum Schluss erstellt das Programm ein Balkendiagramm anhand der Ergebnisse. Die Laufzeiten der Abfragen sowie die erreichte Beschleunigung berechnet das Programm selbst.

Abbildung 32: BabyNames Beispiel
Die Programmanalyse ist in diesem Beispiel deutlich komplizierter, da es sich um eine grafische Anwendung handelt mit deutlich mehr Methoden und Quellcode. Außerdem ist unbekannt, wie schnell PLINQ im Vergleich zu LINQ ist.

Da hier keine Schleifen sind, kann die Schätzung nur anhand der Laufzeitverteilung der einzelnen Methoden gemacht werden. Mithilfe unseres Werkzeugs lässt sich herausfinden, dass die Methode `System.Linq.Enumerable.ToList()` etwa 72% der Laufzeit gearbeitet hat. Wenn man also davon ausgeht, dass eine Parallelisierung ideal ist und durch PLINQ eine vierfache Beschleunigung erreicht wird, dann soll die gesamte Beschleunigung gleich:

\[ S_{ges} = 0,28 + 4 \times 0,72 = 3,16 = 316\% \]

Dass die tatsächliche Beschleunigung etwas größer war, lässt sich unter anderem dadurch erklären, dass PLINQ durch eine größere Cache-Hit Rate eine superlineare Skalierung haben kann.

5.4 Zusammenfassung

In diesem Kapitel wurde das LoopEst-Konzept an einer Reihe von reellen Anwendungen getestet und evaluiert. Es wurde eine normale Leistungsschätzung (siehe Kapitel 5.1 und 5.3) und die Schätzung durch Skalierung (Kapitel 5.2) durchgeführt. Die geschätzten Werte wurden abschließen mit den Werten nach [Gus88] und den tatsächlichen Werten verglichen.

Die Testläufe haben gezeigt, dass LoopEst immer leicht zu optimistisch ist. Dies liegt daran, dass durch Parallelisierung entstehender Mehraufwand in LoopEst nicht berücksichtigt wurde. Die Speicherzugriffe und dabei entstehende Cache-Effekte wurden in dieser Arbeit ebenfalls nicht betrachtet.

6 ZUSAMMENFASSUNG UND AUSBLICK

Im Rahmen dieser Arbeit wurde ein Konzept LoopEst für die kontrollflussbasierte Leistungsschätzung zur Parallelisierung entwickelt und prototypisch implementiert. Das Werkzeug führt eine statische und dynamische Analyse eines Programmes mithilfe von weiteren Werkzeugen durch. Als Ergebnis dieser Analyse wurde ein Ausführungsgraph des Programms erstellt und versucht, durch verschiedene Parameter den Leistungszuwachs, der durch die Parallelisierung von Schleifen oder Methodenaufrufen entsteht, abzuschätzen.


Es besteht auch ein Weiterentwicklungsbedarf, damit die Analyse einerseits flexibler und andererseits generisch wird. Momentan ist eine vorherige Instrumentierung von Schleifen durch den Entwickler nötig. Es wird auch davon ausgegangen, dass die Schleifen problemlos parallelisiert werden können. Es wird nicht überprüft, ob dabei Datenabhängigkeiten entstehen. Das LoopEst-Werkzeug sagt also nur, wie groß der mögliche Leistungszuwachs durch Parallelisierung sein kann, liefert aber keine Aussagen darüber, wie zu parallelisieren ist.


ANHÄNGE

A. Abkürzungsverzeichnis

Abkürzung | Langbezeichnung und/oder Begriffserklärung
--- | ---
.NET | .NET ist eine Implementierung des Common Language Infrastructure-Standards (CLI) für Windows durch den Softwarehersteller Microsoft. Sie besteht aus einer Laufzeitumgebung und Klassenbibliotheken, die gemeinsam eine Basis für Softwareentwicklung bieten.
LINQ | Language Integrated Quer y. LINQ ist eine Komponente in .NET Framework zur Abfrage von beliebigen Datenstrukturen und Quellen.
CLI | Common Language Infrastructure - ist ein ISO/IEC/ECMA Standard, der eine virtuelle Maschine beschreibt, bestehend u.a. aus einem Typsystem, einem Instruktionssatz und einem Laufzeitsystem.
CCI | Common Compiler Interface. Siehe mehr unter [CCI09]
NumCalls | Anzahl der Iterationen in eine Schleife oder Anzahl der Aufrufe einer Methode, die durch dynamische Analyse erkannt werden.

B. Abbildungsverzeichnis

Abbildung 1: Amdahl- und Gustafson-Gesetz im Vergleich .......................................................... 5
Abbildung 2: Sequenzielle und parallele Programmabschnitte .................................................. 6
Abbildung 3: Sequenzdiagramme für die Veranschaulichung des Datenparallelismus .................. 7
Abbildung 5: „teile und herrsche“-Ansatz aus [Pankratius11] ......................................................... 8
Abbildung 7: Thread Pool in .NET 3.5 ............................................................................................ 10
Abbildung 8: Binäre Baum der Tiefe drei für „teile und herrsche“-Muster .................................. 13
Abbildung 9: Beispiel für Call Graph aus [RV+07] ...................................................................... 14
Abbildung 10: Beispiel für einen Datenflussgraphen aus [RV+07] .............................................. 15
Abbildung 11: Parallelisierungsprozess aus [LD02] ................................................................. 15
Abbildung 12: Leistungsabschätzung anhand von Modellen (Bild von PCM Homepage) .......... 16
Abbildung 13: Beispiel „Bilderverarbeitung“ ............................................................................. 19
Abbildung 14: Statischer Ausführungsgraph für Beispiel 1 ......................................................... 21
Abbildung 15: Normale Schleife im Profiler .............................................................................. 21
Abbildung 16: Schleifenkörper als anonyme Methode ................................................................. 22
Abbildung 17: Schleifenkörper in Methode enthalten ................................................................. 22
Abbildung 18: Definition einer Attributklasse ............................................................................ 23
Abbildung 19: Anwendung von Attributen ............................................................................... 23
Abbildung 20: CallerCallee-Profil .............................................................................................. 24
Abbildung 21: CallTree-Profil .................................................................................................... 25
Abbildung 22: Dynamischer Ausführungsgraph mit Schleifen .................................................. 25
Abbildung 23: Schleifenschema ................................................................................................. 27
Abbildung 24: Herleitung der Formel ....................................................................................... 27
Abbildung 25: Laufzeiten für sequenzielle und parallele Variante mit N=4 .............................. 28
Abbildung 26: Gesamtbescheinigung als die Summe ................................................................. 28
Abbildung 27: Leistungsschätzung durch Skalierung ................................................................. 29
Abbildung 28: Evaluierung von Beispiel 1 ............................................................................... 32

Lehrstuhl Prof. Dr. Walter F. Tichy
C. Formelverzeichnis
Formel 1: Beschleunigung durch Parallelisierung ............................................... 4
Formel 2: Effizienzwert bei der Parallelisierung ..................................................... 4
Formel 3: Amdahlsches Gesetz ................................................................................. 5
Formel 4: Gustafsons Gesetz .................................................................................. 5
Formel 5: Ausführungszeit auf p Prozessoren in [LL10] ............................................. 12
Formel 6: Ausführungszeit für Parallel Map .............................................................. 12
Formel 7: Ausführungszeit für Rekursion der Tiefe d ................................................. 13
Formel 8: Ausführungszeit für rekursives Problem .................................................. 13
Formel 9: Ausführungszeit einer Schleife ................................................................. 13
Formel 10: Leistungsschätzung mit NumCalls .......................................................... 27
Formel 11: Leistungsschätzung für den Fall 2 .......................................................... 27
Formel 12: Compute Pi ........................................................................................... 35

D. Tabellenverzeichnis
Tabelle 1: Änderung von P durch Skalierung ............................................................ 29
Tabelle 2: Ähnlichkeiten und Unterschiede mit anderen Arbeiten ............................ 30
Tabelle 3: Leistungsschätzung durch Skalierung ...................................................... 33
Tabelle 4: Leistungsschätzung durch Skalierung ...................................................... 34

E. Literaturverzeichnis
[Rat10] Justin Ratnerm, Leiter der Technologieabteilung bei der Firma Intel

[MSM04] T. Mattson, B. Sanders; B. Massingill: Patterns for Parallel Programming, Addison-Wesley, 2004


[Shi96] Yuan Shi: Reevaluating Amdahl's Law and Gustafson's Law, 1996


[LD02] Jan LEMEIRE, Erik Dirkx: Automated Experimental Parallel Performance
Anhänge

Analysis, 2002


[LINQ05] The LINQ Project http://msdn.microsoft.com/de-de/library/aa479865%28en-us%29.aspx