Karlsruher Institut für Technologie (KIT)

Institut für Anthropomatik, ISL

Karlsruher Institut für Technologie

# Diplomarbeit

# Parallelisierung für statistische maschinelle Übersetzung

Christian Wening

Karlsruhe

Matr.Nr.: 857021

Betreuer:

Prof. Dr. Alexander Waibel

Dr. Sebastian Stüker

Betreuender Mitarbeiter:

Dr. Ing. Jan Niehues

## Ehrenwörtliche Versicherung

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

….....

Datum, Unterschrift Christian Wening

# Inhaltsverzeichnis

# Abbildungsverzeichnis

## Kurzzusammenfassung

In der vorliegenden Arbeit soll untersucht werden, inwiefern die Parallelisierung des KIT-Übersetzungs-Toolkits für statistische maschinelle Übersetzung (SMT) möglich und sinnvoll ist. Dazu werden die grundsätzlichen Probleme und Randbedingungen der Parallelisierung beleuchtet, anschließend wird der vorhandene Programm-Code einer statischen und einer dynamischen Analyse unterzogen.

Die aus der Analyse gewonnenen Erkenntnisse werden benutzt, um eine geeignete Granularität der Parallelität festzulegen. Anschließend wird der vorliegende Code hinsichtlich geeigneter Strukturen für eine Parallelisierung mit der gewünschten Granularität durchleuchtet und die betreffenden Abschnitte werden auf Abhängigkeiten untersucht. Bei der Implementierung werden diese Abhängigkeiten bereinigt und die Mehrfädigkeit integriert.

Zum Abschluss werden die erzielten Ergebnisse geprüft und mit dem theoretischen Ansatz verglichen.

# 1. Einführung

In einer zunehmend dichter vernetzten Welt spielt verbale Kommunikation eine immer bedeutendere Rolle. Um so wichtiger ist es, Sprachbarrieren als Hemmnis zu überwinden.

Damit das Bedürfnis nach Kommunikation nicht zu Gleichschaltung und damit der Verödung der Sprachvielfalt führt, ist es notwendig zuverlässig funktionierende und einfach zu bedienende Übersetzungssysteme für den Alltag bereitzustellen. Douglas Adams hat dieses Bedürfnis 1979, wenn auch in einem etwas weiträumigeren Kontext [A1979], mit seinem Babelfisch – einem in den Gehörgang einzubringenden Simultanübersetzer - vorweggenommen.

Mit dem stetig zunehmenden Leistungspotenzial aktueller Computersysteme und neuen Erkenntnissen aus der Wissenschaft ist es heute immer besser möglich, gute Übersetzungen durch Maschinen anfertigen zu lassen.

Die Ansätze der maschinellen Übersetzung (Machine Translation, MT) folgen dabei unterschiedlichen Wegen:

- Syntaktisch regelbasierte Ansätze versuchen ein Sprachsystem möglichst vollständig durch komplexe grammatische Regeln zu beschrieben und schaffen anschließend Transformationen von der Quell- zur Zielsprache.
- Interlingua erzeugt eine unabhängige Metasprache und nutzt diese als Zwischenschritt für Übersetzungen. Dabei werden für jede Sprache nur die Regeln von und zu dieser Sprache nach Interlingua definiert.
- Die statistische Maschinen-Übersetzung (Statistical Machine Translation, SMT) 'erlernt' aus bereits übersetzten Texten eines Sprachpaars automatisch Übersetzungsregeln und wendet diese auf neue, noch nicht übersetzte Texte an.

Die Funktionsweise von statistischer Maschinen-Übersetzung kann in zwei Kernkomponenten eingeteilt werden:

- In der ersten Komponente, dem Übersetzungsmodell (Translation Model, TM), werden aus von Menschen übersetzten, mehrsprachigen Paralleltexten, wie z. B. Reden im europäischen Parlament erlernte Phrasenpaare aus einem oder mehreren Wörtern nach deren Häufigkeit im Kontext der jeweiligen Sprachdomäne ausgewählt.
- Die zweite Kernkomponente, das Sprachmodell (Language Model, LM) bestimmt anhand umfangreicher Schriften der Zielsprache eine Metrik für die Wohlgeformtheit dieser Phrasen. Ein möglicher Indikator dafür ist ein häufiges Auftreten einer Phrase in als korrekt anerkannten Texten der Zielsprache.

Der Übersetzer versucht nun die wahrscheinlichste Übersetzung zu finden, um weitere, ihm bis dahin unbekannte Texte, zu übersetzen.

Die dafür entwickelte Software "KIT Toolkit" [VHKW2004] wurde programmiert, um in Textform vorliegende Sätze einer Quellsprache in eine Zielsprache zu übersetzen.

Im Training extrahiert das Toolkit dafür mit sogenannten parallelen Korpora (ca. 40 Millionen Phrasenpaare aus Quell- und Zielsprache) und großen Mengen stilistisch korrekter Texte (mehrere Milliarden Wörter), statistische Kennzahlen des Sprachpaares und der jeweiligen Zielsprache.

Der Übersetzer analysiert unbekannte Texte Satz für Satz und findet innerhalb dieser, die im ersten Schritt erlernten Phrasen. Diese werden anschließend übersetzt und ggf. umgeordnet, um dem Satzbau der Zielsprache zu genügen.

Die ursprüngliche Architektur des Übersetzers wurde optimiert auf ein Einprozessorsystem mit, aus heutiger Sicht, geringem Speichervolumen.

Ziel dieser Diplomarbeit ist die Parallelisierung des vorliegenden Programms, also die Umstrukturierung des Quellcodes derart, dass bestimmte Programmteile gleichzeitig auf mehreren Prozessoren ausgeführt werden, ohne Veränderung des ursprünglichen Algorithmus'.

## 1.1. Aufbau der Arbeit

Im Kapitel nach dieser Einführung wird die motivierende Idee für diese Arbeit beschrieben und die Aufgabenstellung umrissen. Darüber hinaus wird die grundsätzliche Vorgehensweise erklärt. In Kapitel 3 werden verwandte Arbeiten und deren Parallelisierungsansatz vorgestellt.

Grundlegende Probleme, relevante Gesetze und verschiedene Techniken von Parallelverarbeitung werden neben der Funktionsweise des KIT-Übersetzers in Kapitel 4 (Grundlagen) dargestellt.

In Kapitel 5 (Parallelisierungsansatz) wird die Lösungsidee und drei Schritte zu deren Realisierung beschrieben.

Den ersten Schritt stellt dabei die Laufzeitanalyse mit Hilfe von Instrumentierung der vorhandenen Code-Basis dar (Kapitel 6 Dynamische Laufzeitanalyse des sequenziellen Codes).

Mit der Implementierung der Parallelisierung befasst sich Kapitel 7 (Parallelisierung) bevor in Kapitel 8 (Validierung der Ergebnisse und Messungen des parallelen Codes) die Ergebnisse der parallelen Ausführung gegen die des sequenziellen Ablaufs geprüft und die Laufzeitunterschiede in Abhängigkeit der Prozessanzahl gemessen werden. Abschließend werden die Ergebnisse mit dem in den Grundlagen vorgestellten Amdahlschen Gesetz verglichen.

Das letzte Kapitel (9 - Zusammenfassung und Ausblick) fasst die Ergebnisse der Arbeit zusammen und zeigt mögliche Wege für zukünftige Verbesserungen auf.

# 2. Motivation

Da seit mehreren Jahren keine signifikanten Zuwächse in der reinen Taktgeschwindigkeit von Mikroprozessoren und somit in der sequenziellen Befehlsverarbeitung mehr zu verzeichnen sind, wohl aber in der Integrationsdichte der Halbleiter, ist eine Verkürzung der Laufzeiten von Programmen und Algorithmen meist nur noch durch Parallelverarbeitung zu erreichen. [S2005]

Dies zeigt auch die Entwicklung heutiger Prozessoren, deren allgemeine Taktfrequenz seit mehreren Jahren bei ca. 3GHz stagniert, während die Anzahl der Kerne stetig steigt. Der gleiche Trend ist auch bei mobilen Devices (Smartphones, Tablets, Phablets, u.vgl.) zu verzeichnen.

## *2.1. Aufgabenstellung*

Ziel dieser Arbeit ist es demnach, das vorliegende Programm zur statistischen Maschinenübersetzung (KIT Übersetzungssystem) aus der rein sequenziellen in eine parallele Ablaufstruktur zu überführen.

Die erste Idee hierfür war, das Übersetzungssystem nach der Initialisierung, vollständige Sätze auf verschiedenen Rechenknoten parallel übersetzen zu lassen. Da jedoch das Toolkit auch für Simultanübersetzungen, wie verbale Konversation oder die Übersetzung von Untertiteln, eingesetzt werden soll, ist eine möglichst geringe Latenz der Ausgabe erwünscht. Da bei diesen Anwendungsszenarien die Sätze dem System nur nacheinander zur Verfügung gestellt werden könnten, wäre eine parallele Bearbeitung nicht möglich. Der deshalb notwendige niedrigere Grad der Granularität schließt diese Variante deshalb aus.

Insofern ist eine der wichtigsten Erkenntnisse am Beginn dieser Arbeit, die Identifikation des maximalen Grades der Parallelverarbeitung der gegebenen Algorithmen im Rahmen der in Kapitel 4 (Grundlagen) näher beschriebenen Gesetze.

## 2.2. Vorgehensweise

Zur Realisierung der Aufgabe muss grundsätzlich geklärt werden, ob und in welchen Programmteilen ein entsprechendes Potenzial für eine Parallelisierung steckt, welche Parallelisierungsverfahren infrage kommen und welches am geeignetsten erscheint. Dies geschieht in Kapitel 4.4 (Verschiedene Techniken der Parallelität).

Zur Bestimmung des Ablaufverhaltens wird wie in Kapitel Fehler: Referenz nicht gefunden (Fehler: Referenz nicht gefunden) erläutert, ein Zeitprofil erstellt, das aufzeigt, wie viel Rechenzeit in welchem Teil des vorhandenen Programms verbraucht wird. Anhand dieses Profils wird der Fokus für das weitere Vorgehen festgelegt.

Anschließend werden mit Hilfe verschiedener Datensätze Testläufe mit unterschiedlicher Anzahl von Prozessorkernen durchgeführt und die Laufzeiten untersucht.

# 3. Verwandte Arbeiten

### 3.1.1. Ren & Shi

Bereits 2000 entwickelten Ren & Shi [RS2001] einen Lösungsansatz, der die teilweise parallele Ausführung eines SMT-Systems zum Ziel hatte. Dabei wurden Sätze parallel von unterschiedlichen Algorithmen übersetzt und die Ausgaben verglichen. Waren die Ergebnisse der unterschiedlichen Systeme identisch, erhöhte dies den Qualitätsindikator dieser Übersetzung.

Konnte keiner der Algorithmen ein Ergebnis erzielen, wurde der Quellsatz in Phrasen zerteilt und diese unabhängig übersetzt und anschließend rekombiniert. Auch hierbei wurden die Phrasen gleichzeitig von den Übersetzungsalgorithmen bearbeitet.

### 3.1.2. Moses Toolkit

Das Moses Toolkit [KetAl2007] verfolgt einen ähnlichen Ansatz wie das KIT Übersetzungssystem. Die Entwicklung begann 2005 und seit 2006 wird es von der EU im Rahmen verschiedener Programme gefördert.

Es beherrscht seit 2013 das parallele Training und die satzweise parallele Übersetzung.

### 3.1.3. Jane

„Jane“ von der RWTH Aachen [VSHN2010] ist ein SMT-System, das in C++ entwickelt wird und zuerst 2010 vorgestellt wurde.

Es erstellt die Phrasentabelle z.T. parallel, indem der Korpus in sogenannte Chunks unterteilt und die folgenden Schritte (count collection, marginal computation und count normalization) dann auf unabhängigen Knoten eines Clusters ausgeführt werden.

Die parallele Übersetzung („Generation“) findet satzweise statt.

### 3.1.4. Joshua

Joshua ist eine Portierung von David Chiangs Pyhton-Implementierung „Hiero“ und wurde von Zhifei Li an der Johns Hopkins Universität, Baltimore in Java entwickelt. Erstmals wurde die Arbeit 2009 in [L.etAl2009] beschrieben.

Ein Parallelisierungsansatz dieser Arbeit geht von einer satzweisen Verteilung des vollständig vorliegenden Quelltexts auf mehrere Prozessoren eines Rechnersystems aus. Alternativ können auch mehrere Sprachmodelle in verschiedene Systeme eines Clusters geladen und dort parallel bearbeitet werden. [LK2008]

### 3.1.5. cdec

Cdec versteht sich als Software-Plattform für die wissenschaftliche Erforschung von Übersetzungsmodellen und entsprechenden Algorithmen. Ihr Design soll sowohl in eingeschränkten Umgebungen (Einzelprozessor, begrenzter Speicher) als auch in großen Clustern effizient sein.

Das cdec-Framework ist für den Stapel-Ablauf („batch“) vorgesehen, nicht für Simultanübersetzungen. Es beherrscht paralleles Training, das via MPI auf verschiedene Rechner eines Clusters verteilt wird.

cdec wurde von Chris Dyer  et. al an der Carnegie Mellon University entwickelt und zuerst 2010 vorgestellt. [D.etAl2010]

### 3.1.6. Parallelisierung von neuronalen Netzen

Ein gutes Beispiel für die Parallelisierung generell sind neuronale Netze: Leicht ersichtlich kann jedes Neuron, dessen Eingaben aus darüber liegenden Schichten anstehen, unabhängig von benachbarten Neuronen bearbeitet werden. Nachfolgende Schichten jedoch können die Bearbeitung jedoch erst beginnen, wenn die Ausgaben aller angebundenen darüber liegenden Neuronen vorliegen. Hierzu muss eine Synchronisationsbarriere integriert werden. [G2012]

# 4. Grundlagen

In diesem Kapitel wird auf Sachverhalte grundlegender Natur eingegangen, die das Verständnis der folgenden Kapitel unterstützen können.

Die Parallelisierung sequenzieller Programme erweist sich oft als aufwendig und teils sogar als fehleranfällig. Um parallelisierbare Teile einer Anwendung identifizieren zu können, ist einerseits grundlegende Kenntnis genereller Parallelisierungsproblematiken, andererseits ein tief greifendes Verständnis des Quellcodes und der Algorithmen des zugrunde liegenden Programms notwendig.

Darüber hinaus existieren Grenzen für den zu erwartenden Geschwindigkeitsgewinn, die in Arbeiten von Amdahl und Gustafson untersucht wurden.

## *4.1. Grundlegende Probleme der Parallelverarbeitung*

Die Parallelverarbeitung von Programmen birgt gegenüber der sequenziellen Befehlsverarbeitung Konflikte, die es zu identifizieren und aufzulösen gilt. Gemäß [BU2002] sind diese die Folgenden:

### 4.1.1. Kontrollflussabhängigkeiten

Eine Anweisung A1, die darüber entscheidet, ob eine folgende Anweisung A2 ausgeführt wird oder nicht, muss auch bei paralleler Ausführung zwingend vor der bedingt auszuführenden abgeschlossen sein.

### 4.1.2. Datenabhängigkeiten

Bei schreibenden Zugriffen auf eine Speicherzelle S ist die Reihenfolge der lesenden und schreibenden Zugriffe entscheidend. Dabei werden grundsätzlich drei Typen von Abhängigkeiten unterschieden:

#### *4.1.2.1.Echte Datenabhängigkeit - Read After Write (RAW)*

Falls die Anweisung A2 die Daten lesen soll, die von Anweisung A1 geschrieben werden, darf A2 erst ausgeführt werden, nachdem A1 abgeschlossen ist.

#### 4.1.2.2. Gegenabhängigkeit - Write After Read (WAR)

Falls die Anweisung A2 Daten überschreibt, die eine vorhergehende Anweisung A1 lesen muss, muss sichergestellt werden, dass dieses Überschreiben erst stattfindet nachdem A1 erfolgreich gelesen hat.

#### 4.1.2.3. Ausgabeabhängigkeit - Write After Write (WAW)

Das Zurückschreiben von Daten der Anweisungen A1 und A2 muss in der richtigen Reihenfolge geschehen, da ansonsten nach Abschluss der Operationen der falsche Wert im Speicher stehen bleibt.

### 4.1.3. Verklemmung (Dead Lock)

Sofern die o.a. Konflikte durch Ausschlüsse (Mutex, Semaphore, Monitor, Scoped Lock, Readers/Writers Lock, etc.) aufgelöst wurden, kann durch den konkurrierenden Zugriff mehrerer Fäden dennoch eine Verklemmung entstehen, wenn beide auf Ressourcen warten, die der jeweils andere Faden sperrt.

## 4.2. Amdahlsches Gesetz

Gene Amdahl hat in seiner Arbeit [A1967] bereits festgestellt, dass ein Programm nicht vollständig parallel ausführbar ist. So lässt sich jedes Programm (Ausführungszeit $T_{(1)}$) in einen vollständig sequenziell *(S = 1 – P)* und einen vollständig parallel *(P)* ausführbaren Teil unterteilen, wobei gilt:

$$T_{(1)} = (T_{(1)} - T_{(P)}) + T_{(P)}$$

Der Geschwindigkeitsgewinn *S* (engl. Speedup) ist dabei abhängig von der Anzahl *N* der verwendbaren Prozessoren und wird dargestellt als:

$$S_{(N)} = \frac{T_{(1)}}{(T_{(1)} - T_{(P)}) + \frac{T_{(P)}}{N}} \leq \frac{T_{(1)}}{T_{(1-P)}} \quad ,$$

Dabei ist *S* als Faktor der rein sequenziellen Ausführungszeit der Anwendung zu verstehen.

Der Gewinn ist, wie unschwer ablesbar ist, durch den sequenziellen Anteil des Programms limitiert: Bei Programmen mit hohem sequenziellen Anteil geht er gegen 1, bei Programmen mit hohem parallelen Anteil und großer Anzahl verwendbarer Rechenwerke kann er nahezu beliebig klein werden und strebt gegen 0.

Fügt man nun noch den Organisationsaufwand $o_{(N)}$, bspw. für die Initialisierung der parallelen Fäden, hinzu erhält man:

$$S_{(N)} = \frac{T_{(1)}}{T_{(1-P)} + T\left(o_{(N)}\right) + \frac{T_{(P)}}{N}} \leq \frac{T_{(1)}}{T_{(1-P)}}$$

Somit konvergiert die daraus resultierende Kurve nicht mehr gegen *1/1-P,* sondern erreicht ein Maximum, um danach wieder abzufallen.

## 4.3. Gustafsons Gesetz

Nach Gustafsons Gesetz [G1988] ist der Geschwindigkeitsgewinn S gegeben durch:

$$S_{(N)} = T_{(1-P)} + N * T_{(P)} \quad ,$$

zu verstehen als Maß einer lösbaren Problemgröße innerhalb eines festen Zeitfensters.

Wie leicht ersichtlich ist, wäre der Speedup bei vollständig unabhängigen Programmteilen mit entsprechender Größe und hoher Anzahl zur Verfügung stehender Prozessoren nahezu unbeschränkt.

## 4.4. Verschiedene Techniken der Parallelität

Unter Parallelverarbeitung versteht man die gleichzeitige Bearbeitung von Programmschritten unter der Voraussetzung derer Unabhängigkeit.

Es existieren aktuell viele unterschiedliche Methoden zur parallelen Verarbeitung von Programmen und Algorithmen. Im Wesentlichen unterscheiden sich die Verfahren im Grad der Granularität der parallelisierten Einheiten: Während auf der einen Seite aktuelle Mikroprozessoren kleine Maschinenbefehle in ihren Pipelines in noch kleinere Mikrooperationen zerteilen und diese über Wartestationen den mehrfach vorgehaltenen Ausführungseinheiten parallel zuführen, stehen am anderen Ende der Skala komplexe, vollständige Applikationen, die auf unabhängigen Rechnersystemen, teils mit unterschiedlichen Hardware-, Software- oder Betriebssystemarchitekturen, die weltweit verteilt auf höchster Systemebene gemeinsame Aufgaben erledigen. [S1997]

Da im vorliegenden Fall eine bereits vorhandene Software parallelisiert werden soll, stehen nicht alle Freiheitsgrade uneingeschränkt zur Verfügung. Aus diesem Grund wurden drei gängige Techniken (eine davon in zwei Ausprägungen) einer näheren Betrachtung unterzogen.

### 4.4.1. Message Passing Interface (MPI)

Message-Passing-Interface ist die Definition einer Schnittstelle, die dazu genutzt wird, Operationen und ihre Semantik zwischen Prozessen auszutauschen. [MPI1994] MPI schreibt kein Protokoll oder eine konkrete Implementierung vor. Dadurch kann MPI über Rechnergrenzen hinweg eingesetzt werden, wobei jedes System über seinen eigenen Speicher und sein eigenes Betriebssystem verfügt.

Die Implementierung erfolgt explizit und sollte bereits beim Entwurf berücksichtigt werden.

Für gute Ergebnisse ist eine schnelle Verbindung zwischen den beteiligten Rechnersystemen wichtig. In einem beispielhaften, idealisierten Szenario werden umfangreiche Aufgaben auf verhältnismäßig kleinen Datenpaketen ausgeführt, die zu Beginn an die beteiligten Knoten ausgeteilt werden. Nach erfolgter Bearbeitung durch einen eigenen, dort ausgeführten Programmteil senden die Knoten die Ergebnisse zurück und signalisieren ggf. ihre Bereitschaft für weitere Aufgaben.

### 4.4.2. General Purpose computing on Graphics Processing Units (GPGPU) / OpenCL

Die Entwicklung hochperformanter Grafikbeschleuniger für Computerspiele hat auch vorteilhafte Folgen für die wissenschaftliche Arbeit. Ein großer Teil der Berechnungen moderner 3D-Spiele besteht aus mathematischen Operationen, wie Matrizentransformationen oder einfachen Fallunterscheidungen. Im Rahmen dieser Befehle weisen aktuelle Grafikkartenprozessoren ein um eine Größenordnung höheres Leistungspotenzial auf, wie vergleichbar teure Mikroprozessoren. Um dieses Potenzial nutzen zu können, haben mehrere Hersteller (nVidia: CUDA; AMD/ATI: Stream) proprietäre Programmiersprachen entwickelt, die auf ihre jeweilige Hardware optimiert wurden. [GPGPU2002], [OCL2009]

Erfreulicherweise existiert mit OpenCL auch eine herstellerübergreifende parallele Standardsprache, die auf unterschiedlichen Plattformen gleichermaßen funktioniert.

Die Implementierung erfolgt dabei explizit in einer eigenen Syntax, die Befehlsbearbeitung geschieht gleichzeitig auf den Shader-Prozessoren der GPU und in dem dort angeschlossenen Speicher. Operationen werden, wie die zugehörigen parallelen Programme, in den Speicher der Grafikkarte geladen, Ergebnisse von dort wieder ausgelesen.

Zur Bearbeitung können auch größere Datenmengen aus dem Speicher des Rechners in den Speicher der GPU transportiert werden. Der Datenaustausch mit anderen Knoten (GPUs) ist implizit aktuell nicht möglich und bedürfte einer gesonderten Behandlung durch bspw. MPI.

### 4.4.3. OpenMP

Die herstellerunabhängige Programmierschnittstelle „Open Multi-Processing“ [OMPI2004] erreicht ihre Parallelität auf einer sehr niedrigen Ebene innerhalb des Programmcodes: So werden häufig durchlaufene Programmteile, wie z. B. Schleifen, in mehreren Fäden innerhalb eines Systems gleichzeitig ausgeführt.

Dazu werden in den Programm-Code Compiler-Direktiven eingefügt, die von einem Präprozessor in kompilierbaren Code übersetzt werden; dadurch bleibt der Quellcode grundsätzlich für nicht OpenMP-fähige Compiler übersetzbar; die OpenMP-Direktiven werden dann ignoriert.

Auf dieser niedrigen Ebene ist OpenMP nicht in der Lage Fehler automatisch zu detektieren; so müssen Synchronisationsbarrieren manuell in den Code eingefügt werden, ansonsten könnten kritische Wettlaufsituationen falsche Ergebnisse erzeugen. Auch Semaphoren für den Zugriff auf dann gemeinsam genutzte Ressourcen müssen explizit implementiert werden.

OpenMP unterstützt nur Shared-Memory-Systeme und kann nicht über Rechnergrenzen hinweg kommunizieren. Diese Einschränkungen versucht die im Folgenden beschriebene Erweiterung Cluster OpenMP aufzuheben.

### 4.4.4. Cluster OpenMP

Um die OpenMP innewohnende Begrenzung des Speicherzugriffs auf den lokalen Speicher des jeweiligen Rechners aufzuheben, erweitert das Speichermodell von Cluster OpenMP dieses um sogenannte „Global Arrays“: sie stellen einen virtuellen Speicherbereich dar, der die teilnehmenden Cluster-Knoten überspannt. Es ist somit nicht notwendig eigene Programmteile für die rechnerübergreifende Kommunikation zu implementieren.

Bestehender Programm-Code muss dennoch an dieses Speichermodell angepasst werden. Da aber ein Speicherzugriff auf einen entfernten Knoten zwischen 100 und 1000 Mal langsamer ist, als ein Zugriff auf den lokalen Speicher, sind solche Zugriffe mit Umsicht zu verwenden.

Nach [COMP2005] bleiben Cluster-OpenMP-Implementierungen bei gleicher Anzahl von Prozessoren in ihrer Leistungssteigerung meist hinter Shared-Memory-OpenMP-Implementierungen zurück.

### 4.4.5. Gegenüberstellung Parallelisierungs-Techniken

| Technologie / Merkmal | MPI | GPGPU | OpenMP | Cluster OpenMP/ DSM OpenMP |
|---|---|---|---|---|
| **Implementierung** | OpenMPI mit Wrapper-Compiler | OpenCL/Cuda | gcc 4.2: OpenMP 2.5<br>gcc 4.4: OpenMP 3.0 | Intel's Cluster OpenMP (eingestellt) |
| **Art der Parallelität** | explizit | explizit | implizit | implizit mit expliziter Speicherverwaltung |
| **Speichermodell** | Distributed Memory Multiprocessor (DMM) | eigener Speicher der GPU (~SMM) | Shared Memory Multiprocessor (SMM) | Distributed Shared Memory (DSM) |
| **Vorteile** | kann auf Cluster ausgebreitet werden | sehr performant auf vergleichsweise günstiger Hardware | Code bleibt für Nicht-openmp-Compiler weiterhin kompilierbar, da Realisation über Compiler-Direktiven | Parallelität in lokalen Knoten einfach integrierbar in bestehende Projekte |
| | ideal für grobkörnige Parallelisierung mit wenig Datenverkehr und viel Rechenaufwand | | leicht nachträglich in bestehende Programme integrierbar | auch für Cluster geeignet |
| | | | jeder parallele Thread kann sowohl private als auch globale Variablen benutzen | |
| **Nachteile** | nur sehr aufwendig in bestehende Architektur integrierbar (Speichermodell) | nur sehr aufwendig in bestehende Architektur integrierbar | keine automatische Detektion von Fehlern (z.B. Raceconditions) | bestehendes Projekt muss auf komplett andere Speicherarchitektur aufgesetzt werden |
| | wenig Performance-Gewinn bei feinkörniger Parallelität | Performance-Gewinn nur bei hohem Parallelitäts-Grad und verhältnismäßig einfachen Kommandos | nicht einsetzbar auf DM-Rechnern (Cluster) | Die meisten openMP-DSM Projekte sind im pre-alpha Stadium oder eingestellt |
| **Anwendungsbereich** | ideal für schnell vernetzte Server mit gleicher Architektur | ideal für Workstations mit potenter GPU | ideal für Rechner mit mehreren Kernen und gemeinsamem Speicher | ideal für Cluster und/oder Rechner mit vielen Kernen |

## 4.5. Funktionsweise des KIT-Übersetzers

Zunächst wird der KIT-Übersetzer einer genaueren Untersuchung unterzogen und seine Funktionsweise schematisch dargestellt. Dies dient als Grundlage für die weitere Analyse.

### 4.5.1. Darstellung des Programmablaufs

Der Programmablauf des KIT-Decoders kann vereinfacht folgendermaßen dargestellt werden: (Abbildung 1: Ablaufdiagramm des KIT-Decoders, Seite 18)

Zuerst alloziert das Hauptprogramm benötigte Ressourcen und lädt Konfigurationen, die Phrasentabelle sowie die Sprachmodelle in den Hauptspeicher des Rechners.

Danach beginnt die satzweise Übersetzung in einer äußeren großen Schleife. Hierin wird ein einfaches Lattice mit möglichen Übersetzungen erzeugt. Dazu wird im Übersetzungsmodell nach passenden Phrasen gesucht und an die Hypothesenexpansion übergeben, wo einzelne Phrasen der zuerst leeren Hypothese hinzugefügt und bewertet werden. Anschließend werden noch nicht übersetzte Wörter hinzugefügt. Ist eine neue Hypothese besser als die bisherigen, wird sie gespeichert. Falls weitere mögliche Hypothesen existieren, wird die Hypothesenexpansion erneut ausgeführt; im anderen Fall wird die Übersetzung des gerade bearbeiteten Satzes ausgegeben und es wird mit dem nächsten Satz, falls vorhanden, fortgefahren.

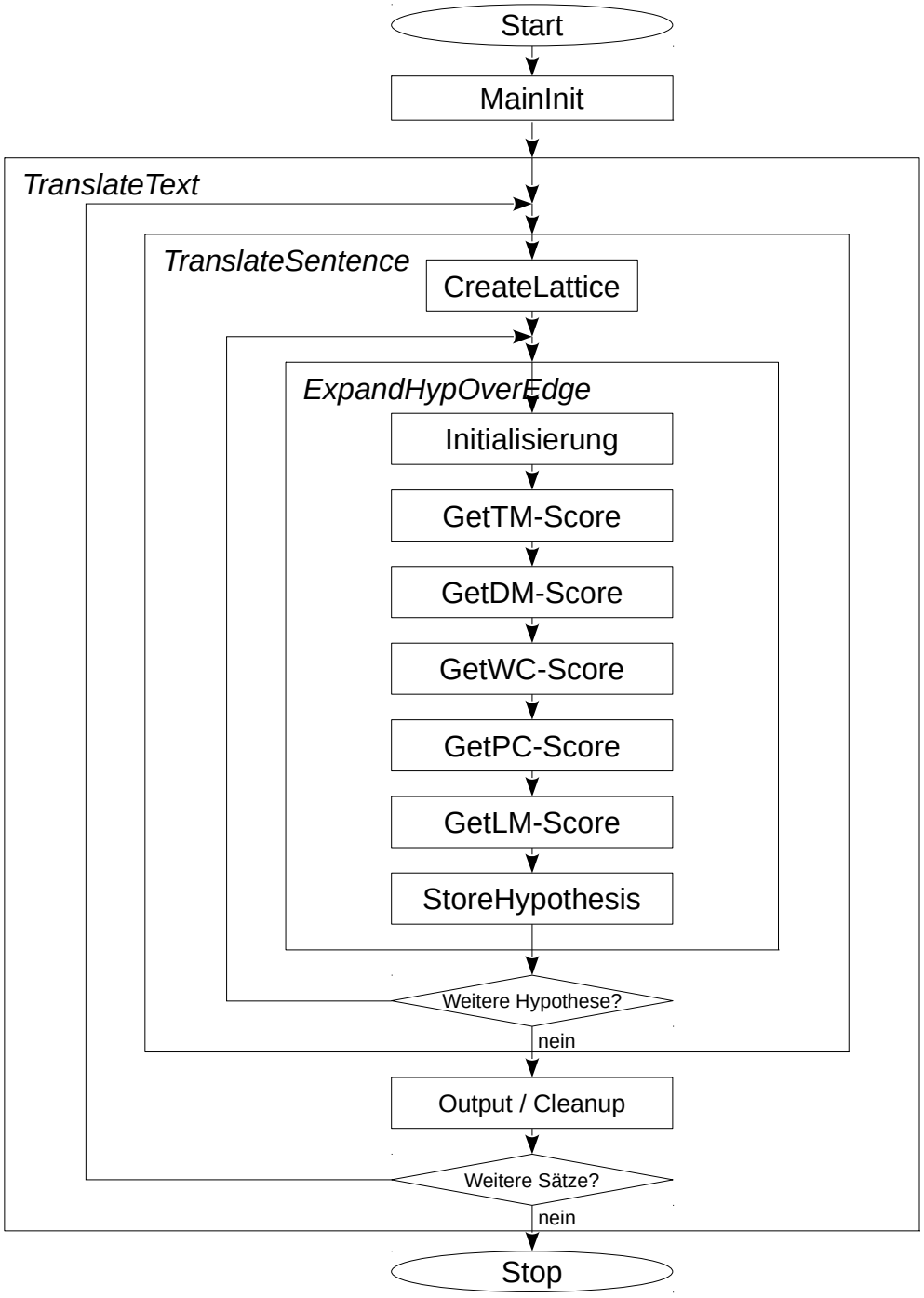


*Abbildung 1: Ablaufdiagramm des KIT-Decoders*

### 4.5.2. Aufbau des Lattice

Der aktuelle Decoder des KIT-Übersetzers stellt alle möglichen Übersetzungen als einen Netz-Graphen, den sogenannten Lattice (Abbildung 2: Beispiel eines Lattice, © 2013 Institut für Anthropomatik) dar.

Dazu erweitert er, ausgehend von einer leeren Hypothese (dargestellt als Knoten im Graph) diese um eine mögliche Übersetzung einer Phrase (ein oder mehrere Wörter) des Quellsatzes. Jeder Übersetzung wird dabei eine partielle Bewertung zugeordnet.

Die neu erzeugte Hypothese wird anschließend einem Container gemäß ihrer Abdeckung („coverage“, ein Vektor der die bereits übersetzen Wörter des Quellsatzes identifiziert) zugeordnet, sowie die Hypothese anhand mehrerer Modelle (Übersetzung, Umordnung, Wortanzahl, Phrasenanzahl, Sprachmodell(e)) bewertet. Sollte sich dabei herausstellen, dass sich in dem jeweiligen Container bereits mehrere Hypothesen mit besseren Bewertungen befinden, wird die soeben bearbeitete Hypothese verworfen.

Falls eine neu erstellte Hypothese einer bereits vorhandenen entspricht, wird nur die mit der besseren Bewertung dem Lattice hinzugefügt und die ggf. entstehende Lücke durch einen Zeiger überbrückt.

Durch das sogenannte Reordering kann eine Phrase auch mehrere, nicht aufeinanderfolgende Wörter des Quellsatzes abdecken.

Im weiteren wird jede beibehaltene Hypothese so lange erweitert, bis die Übersetzung alle Wörter des Quellsatzes abdeckt (full coverage).

Somit bewirkt eine hohe Auftrittswahrscheinlichkeit im Sprach-Modell der Zielsprache eine positive Bewertung einer bestimmten Phrase. Daneben stellen auch die Häufigkeit von Umordnungen (Reordering), die Anzahl der Worte und die Anzahl der Phrasen Qualitätskriterien für eine Übersetzung dar.

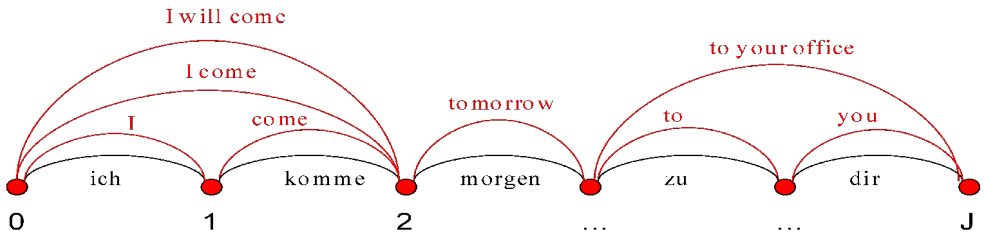


*Abbildung 2: Beispiel eines Lattice, © 2013 Institut für Anthropomatik*

# 5. Parallelisierungsansatz

## 5.1. Wahl der Parallelisierungstechnik

Der Übersetzer des KIT Toolkits muss, um seine Arbeit effizient verrichten zu können, die Phrasentabelle in möglichst kurzer Zugriffszeit auslesen können. Zugriffe erfolgen dabei zufällig (random read). Deshalb werden diese Tabellen vollständig in den Hauptspeicher des jeweiligen Rechners geladen.

Aus diesem Grund brächten Systeme mit verteiltem Hauptspeicher (Distributed Shared Memory, DSM) enormen Zuschlag mit sich, da alle Tabellen jedem Knoten zur Verfügung gestellt werden müssten. Dazu müssten sie auf alle beteiligten Knoten kopiert werden, was auch bei großer Bandbreite einige Zeit in Anspruch nehmen würde. Außerdem würde der Korpus auf jedem Knoten einen entsprechend großen Teil des Hauptspeichers belegen.

GPGPU ist optimiert auf das gleichförmige Verarbeiten großer Datenmengen, wie z. B. umfangreiche Matrizenmultiplikationen oder einfaches Verknüpfen von großen Speicherbereichen. Verzweigungen und Fallunterscheidungen wirken sich bei GPGPU stark leistungsmindernd aus. Sollten darüber hinaus mehrere Systeme an der Bearbeitung beteiligt sein, treten zusätzlich noch die o.a. Probleme von DSM-Systemen auf. Weiterhin müsste der vorliegende Code weitgehend neu implementiert werden. Diese Eigenschaften der GPGPU-Architektur sprechen gegen den Einsatz als Parallelisierungstechnik für den KIT-Übersetzer.

Bei OpenMP würde nicht nur die Code-Basis als solche erhalten bleiben, sie wäre sogar weiterhin auf Single-Core-Maschinen kompilier- und ausführbar. Die umfangreichen Korpus-Tabellen werden nur ein Mal in den Hauptspeicher des Shared-Memory-Rechners geladen, die Kommunikation zwischen den einzelnen Befehlsfäden hätte minimalen Overhead.

Bei Bedarf wäre zu einem anderen Zeitpunkt noch die Ausdehnung qua Cluster-OpenMP auf unabhängige Rechner möglich.

Aus den o.a. Erwägungen wurde die Entscheidung gefällt, den vorhandenen Code, hinsichtlich seines Parallelisierungspotenzials zu untersuchen, die beschriebenen Konflikte aufzudecken und ihn dort, wo dies notwendig ist, umzuarbeiten und um OpenMP-Direktiven zu erweitern.

## 5.2. Schritte zur Parallelisierung

Zur Parallelisierung eignen sich grundsätzlich Methoden, die häufig, beispielsweise in Schleifen, aufgerufen werden. Diese dürfen eine bestimmte Größe nicht unterschreiten, da ihre parallele Initialisierung einen festen Aufwand pro Befehlsfaden verursacht. Weiterhin müssen sie unabhängig ausführbar sein oder unter Beibehaltung ihres Algorithmus' derart umgearbeitet werden, dass sie unabhängig ausführbar werden.

Am Ende der parallelen Ausführung sind oft Synchronisationsschranken einzufügen, an denen die in den unterschiedlichen Befehlsfäden bearbeiteten Ergebnisse zusammengeführt werden. Insofern ist es von Vorteil, wenn der Umfang der parallel ausgeführten Methoden nicht stark abweicht.

Schließlich sollte die Parallelisierung innerhalb der Übersetzung einzelner Sätze stattfinden um Simultanübersetzungen zu ermöglichen.

Somit können die Kriterien, die eine Methode für die Parallelisierung qualifizieren, folgendermaßen zusammengefasst werden:

- häufige Wiederholungen
- hinreichende Mindestgröße
- weitgehende Unabhängigkeit hinsichtlich der genutzten Daten
- ähnliche Laufzeit
- Granularität unterhalb satzweiser Parallelität

### 5.2.1. Analyse

Im ersten Schritt (Kapitel Fehler: Referenz nicht gefunden - Fehler: Referenz nicht gefunden) wird eine dynamische Laufzeitanalyse des Codes durchgeführt um die Abschnitte zu identifizieren, für die eine Parallelisierung gemäß den o.a. Kriterien erfolgversprechend erscheinen.

Dazu wird der Sourcecode mit Zeitmarken instrumentiert und verschiedene Programmdurchläufe durchgeführt, die danach ausgewertet werden (Kapitel 6.1 - Einsatz von Messpunkten).

Mit den Ergebnissen der Auswertung aus Kapitel 6.4 (Auswertung und Auswahl der Granularität) werden die gewählten Programmteile einer statischen Analyse unterzogen werden. Dabei wird der Code en détail auf potenziell geeignete Bestandteile für die Parallelisierung, wie Schleifen oder ähnlich gleichförmige Bestandteile untersucht. Im Weiteren werden diese Programmteile hinsichtlich der in Kapitel Fehler: Referenz nicht gefunden (Fehler: Referenz nicht gefunden) beschriebenen Konflikte untersucht.

### 5.2.2. Parallelisierung

Wo Hemmnisse und Konflikte auftreten können, werden verschiedene Ausschlüsse oder Erweiterungen eingeführt. Dazu kommen folgende Techniken in Frage:

- Mutex

  Ein Mutex beschreibt den wechselseitigen Ausschluss von Befehlsfäden aus kritischen Abschnitten. Ein solcher Abschnitt ist beispielsweise eine globale Tabelle, auf die zumindest von einem Faden schreibend zugegriffen werden soll.

  Mutexe können unter ungünstigen Umständen zum Anhalten aller Fäden führen (Deadlock) oder bei Befehlsfäden mit unterschiedlichen Prioritäten zur Umkehrung dieser führen (Prioritätsinversion). Weiterhin kann das Verlassen des kritischen Bereichs aufgrund einer Ausnahmebehandlung die Freigabe der Ressource verhindern.

- Scoped Lock

  Beim Scoped Lock wird eine Wächterklasse definiert, deren Konstruktor eine Sperre akquiriert und deren Destruktor diese wieder freigibt. So kann für jeden Block der kritischen Ressource eine eigene Sperre realisiert werden, die auch beim Verlassen eines kritischen Bereichs bspw. durch eine Ausnahmebehandlung, die korrekte Freigabe der Ressource bewirkt.

- Readers/Writers Lock

  Das Readers/Writers Lock unterscheidet bei parallelen Zugriffen, ob diese lesend oder schreibend erfolgen sollen. Liegen nur lesende Anfragen vor, können diese parallel ablaufen, bei schreibenden Zugriffen werden alle weiteren Anfragen blockiert, bis der schreibende Befehlsfaden den kritischen Bereich verlassen hat.

- Anpassung von Parameterübergaben bei Methodenaufrufen

  Oft werden bei Methodenaufrufen nicht nur Parameter, sondern aus Effizienzgründen nur Zeiger auf Datenstrukturen übergeben. Bei mehrfädiger Ausführung könnte dies dazu führen, dass unterschiedliche Fäden auf denselben Strukturen Operationen ausführen, was zu inkonsistenten Zuständen dieser Strukturen führen würde. Dies kann bereinigt werden, indem beim Methodenaufruf die komplette Datenstruktur übergeben wird.

- Erweiterung von Datenstrukturen

  Globale Datenstrukturen können für jeden Befehlsfaden gesondert erzeugt und vorgehalten werden, wobei sichergestellt werden muss, dass alle Exemplare vor der Weiterverarbeitung in geeigneter Weise zusammengeführt werden, und die aggregierten Ergebnisse den der sequenziellen Ausführung entsprechen.

Die Einführung dieser Techniken wird in Kapitel Fehler: Referenz nicht gefunden (Fehler: Referenz nicht gefunden) beschrieben.

### 5.2.3. Validierung der Ergebnisse und Messung

Um die Ergebnisse zu überprüfen, werden Vergleichsmessungen mit unterschiedlicher Anzahl von Programmfäden auf unterschiedlichen Testsets durchgeführt (Kapitel 8.2.1 Messung mit Sprachmodell-Toolkit SriLM und Kapitel 8.2.2 Messungen mit Sprachmodell-Toolkit KenLM).

Zum Abschluss werden die Protokolle dieser Messungen verglichen und ausgewertet (Kapitel 8.3 Vergleich der Ausführungszeit mit Amdahlschem Gesetz). Damit kann der effektive Zeitgewinn (Speedup) der Parallelisierung für diese Software ermittelt werden.

Im Anhang Fehler: Referenz nicht gefunden finden sich Details zur Implementierung, die die Anpassung des vorhandenen Codes und der Umgebung dokumentieren; sie sollen dazu dienen die Mehrfädigkeit auch in andere Äste der Implementierung des Decoders zu integrieren.

# 6. Dynamische Laufzeitanalyse des sequenziellen Codes

Zur genauen Messung der verbrauchten Rechenzeit der einzelnen Programmteile wurden Zeitmarken und Zählmarken in den Code eingeführt, die eine Analyse erlauben, wie oft ein Prozessor einen bestimmten Programmteil durchläuft und wie viel Arbeit er dort verrichtet. [H2013]

Dazu musste dem Projekt der High-Precision-Timer zugefügt werden, da dieser in der Lage ist, verstrichene Zeit in Mikrosekunden zu erfassen:

Grundsätzlich wären auch Taktzyklen (CPU-Clocks) dazu geeignet aber bei entsprechenden Tests hat sich herausgestellt, dass deren Werte durch Präemption beim Wechsel eines Fadens von einem Kern auf einen anderen fundamental falsche Ergebnisse geliefert hätten.

Aus diesem Grund wurde der High-Precision-Timer eingeführt und damit die Ausführungszeit in Echtzeit gemessen.

Bei der eingesetzten Realtime-Clock ist zu beachten, dass bei häufiger Verdrängung des gemessenen Fadens auf einem Kern ebenfalls leicht verfälschte Ergebnisse entstehen können. Deshalb ist bei Referenz-Messungen darauf zu achten, dass jeder Faden möglichst lange und ungestört auf einem Kern arbeiten kann.

Das Testset besteht aus einer Menge von 2000 zu übersetzenden Sätzen, wobei die Quellsprache Deutsch und die Zielsprache Englisch ist. Die Übersetzungstabelle umfasst ca. 830.000 Phrasenpaare, das Sprachmodell ist von der Ordnung 4 und hat einen Umfang von ca. 12.000 Einträgen.

Bei den Messungen werden die jeweiligen Hauptmethoden in absteigender Reihenfolge, beginnend mit der `main()`-Methode, betrachtet. Beansprucht eine Methode einen überwiegenden Anteil (>90%) der Ausführungszeit, wird nur diese weiter verfolgt, ansonsten werden alle Methoden betrachtet, die mehr als 10 % Anteil an der Ausführungszeit der aufrufenden Methode beanspruchen.

Methodennamen werden durch einen `Schriftsatz fester Laufweite` und ein abschließendes Paar aus einer öffnenden und einer schließenden Klammer gekennzeichnet.

## 6.1. Einsatz von Messpunkten

In den Code wurden Messpunkte integriert, die am Beginn und am Ende einer Methode eine Zeitmarke ausgeben. Diese Zeitmarken wurden anschließend ausgewertet. Im Weiteren werden Code-Elemente grau unterlegt dargestellt. Sofern dies nicht explizit anders angegeben wird, handelt es sich dabei um C++.

Hier ein Beispiel für die Erfassung der Laufzeit der `main()`-Methode:

```
#ifdef COMPILE_TIMEMEASUREMENT
 clock_gettime(CLOCK_REALTIME, &TimeStampMainExit);
cout<<"cw:Main;"<<(TimeStampMainExit.tv_sec-
TimeStampMainEnter.tv_sec)<<";"<<(TimeStampMainExit.tv_nsec-
TimeStampMainEnter.tv_nsec)<<endl;
#endif
```

Da die Rückgabewerte des High-Precision-Timers für Sekunden und Nanosekunden in zwei gesonderten Variablen zurückgegeben werden, müsste ein Fallunterscheidung integriert werden.

Beispiel:

Sei $t_n$: 1000 [s] 500 [ns] und $t_{n+1}$: 1010 [s] 400 [ns]

Werden nun die Variablen für Sekunden und die für Nanosekunden subtrahiert ($t_{n+1}$-$t_n$) ergibt sich

$\Delta_t$:10 [s] -100 [ns].

Bei einer einzigen Messung wäre dies ein zwar prinzipiell richtiges aber schlecht zu interpretierendes Ergebnis. Da im vorliegenden Fall Hunderte bis Tausende von $\Delta_t$-Werten aufsummiert werden, bei denen von einer statistischen Gleichverteilung ausgegangen werden kann, wird dieser Fehler vernachlässigbar klein.

Um die Größe der häufig aufgerufenen Laufzeitmessung gering zu halten, wurde deshalb auf die Fallunterscheidung bei negativen Nanosekunden-Werten verzichtet.

## 6.2. Darstellung der Messwerte und Erläuterung

Die in diesem Kapitel dargestellten Kuchendiagramme repräsentieren die gesamte Laufzeit der gemessenen Methode, die im jeweiligen Titel beschrieben wird.

In der Diagrammfläche oder an dessen Rand finden sich die Namen der aufgerufenen Methode nebst dem prozentualen Anteil der Gesamtausführungszeit der aufrufenden Methode, gerundet auf eine Nachkommastelle.

In der dazu gehörenden Tabelle sind die jeweiligen Werte nochmals in numerischer Form aufgeführt. Dabei werden auch Aufrufe berücksichtigt, die im Diagramm aufgrund ihrer Größe nicht dargestellt werden können.

Die Akronyme in Versalien sind die konkatenierten Anfangsbuchstaben der aufrufenden und der aufgerufenen Methode.

### 6.2.1. Laufzeitanalyse Gesamtdurchlauf

Für einen vollständigen exemplarischen Durchlauf (ein Faden, PC208) ergibt sich folgendes Bild:

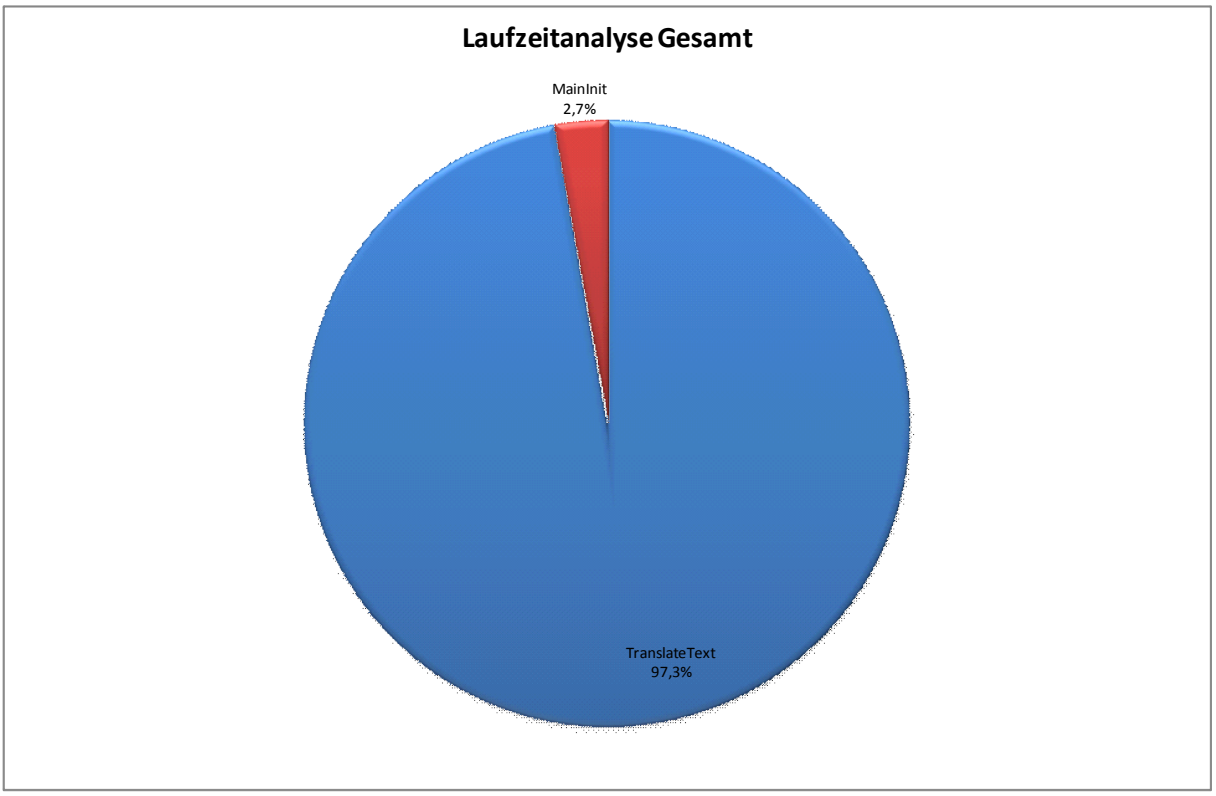


*Abbildung 3: Laufzeitanalyse Gesamtdurchlauf*

Der Programmdurchlauf beginnt mit der Initialisierung des Decoders, die hauptsächlich aus dem Laden der Phrasentabelle und des Sprachmodells besteht.

Anschließend wird für 2000 Sätze der jeweilige Lattice erstellt, und dessen Erweiterungen mit Hilfe der geladenen Modelle bewertet. Die 20 bestbewerteten Übersetzungen werden nach der Bearbeitung jedes Satzes ausgegeben und mit dem nächsten fortgefahren.

Der gesamte Durchlauf nahm 2936 Sekunden in Anspruch, also ca. 49 Minuten. Davon entfiel auf die Initialisierung (Programmstart, Speicherallokierung, Ladevorgang der Modelle und der Tabellen) knapp drei Prozent (79 s) der Gesamtlaufzeit, der Rest auf eine Schleife in der Methode `TranslateText()`, die die gesamte Übersetzung des Texts durchführt. Im Folgenden wird nur diese betrachtet.

### 6.2.2. Laufzeitanalyse TranslateText

Die Funktion `TranslateText()` führt die Übersetzung des Textes, Satz für Satz durch (Abbildung 4: Laufzeitanalyse TranslateText(), Seite 28). Die Sprachmodelle und die Phrasentabelle stehen dabei bereits im Hauptspeicher des Rechners zur Verfügung.

Grundsätzlich wären diese Schleifendurchläufe hinsichtlich der meisten Parallelisierungskriterien geeignet; da die Granularität der Parallelisierung unterhalb der Satzgrenze liegen soll, wird der mögliche Ansatz an dieser Stelle übergangen.

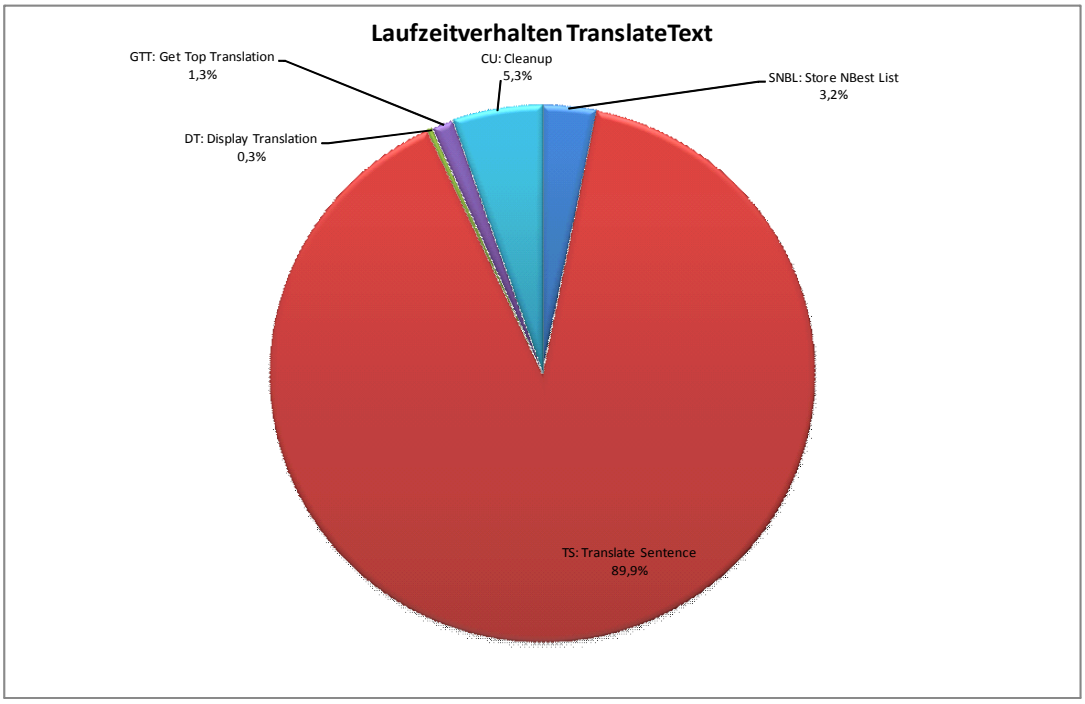


*Abbildung 4: Laufzeitanalyse TranslateText()*

| TracePointer | Zeit [s] | Anteil | Beschreibung **TranslateText** |
|---|---|---|---|
| cw:TS | 2643,298 | 90,61% | TS: Translate Sentence, führt eine Schleife über alle Sätze des Quelltextes aus. |
| cw:CU | 144,431 | 4,95% | CU: Cleanup, Löschen des Lattice, Freigabe von Ressourcen |
| cw:TLT | 2917,242 | 100,00% | TLT: Total Loop Time, Gesamtzeit der Methode TranslateText |

Wenig überraschend verbringt die `TranslateText()`-Methode den größten Teil ihrer Ausführungszeit (90,6%) in der Methode `TranslateSentence()`, weshalb im Weiteren nur diese einer genaueren Betrachtung unterzogen wird.

### 6.2.3. Laufzeitanalyse TranslateSentence, Apply Language Model, FindMHPCBest

Die Methode `TranslateSentence()` erstellt durch `CreateLattice()` die initiale Datenstruktur des leeren Lattices. Der Aufbau erfolgt anschließend in der Methode `ApplyLanguageModel()`, welche 99,4% der Ausführungszeit der untersuchten Methode in Anspruch nimmt.

Im Wesentlichen expandiert die Methode `ApplyLanguageModel()` neue Hypothesen an Knoten des Lattices mit `ExpandHypotheses()`, die ihrerseits nur eine einzige weitere Methode in Abhängigkeit der Abdeckung (Coverage) aufruft: `ExpandAllHypotheses-WithCoverage(Coverage)`. Aus diesem Grund wurde auf eine grafische Darstellung des Zwischenschritts verzichtet.

Diese Funktion verbringt zu über 96% ihrer Laufzeit mit der Methode `FindMHPCBest()`. Analog wird weiterhin die Methode `ExpandHypOverEdge()` untersucht, die `FindMHP-CBest()` nahezu vollständig ausfüllt.

### 6.2.4. Laufzeitanalyse Expand Hyp Over Edge

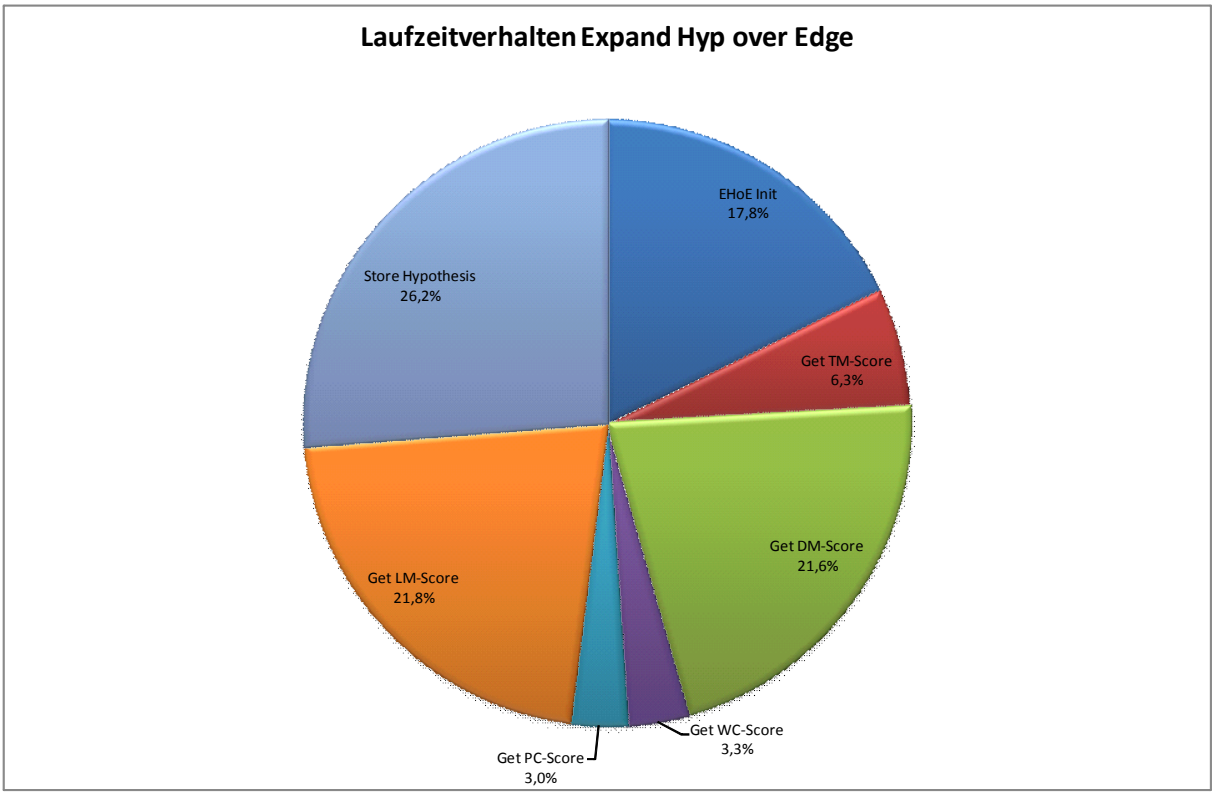


*Abbildung 5: Laufzeitanalyse ExpandHypOverEdge*

| TracePointer | Zeit [s] | Anteil | Beschreibung |
|---|---|---|---|
| | | | **EHOE: Expand Hyps Over Edge** |
| cw:EHOEI | 351,202 | 17,8% | Init; Initialisierung |
| cw:EHOEGTMS | 121,141 | 6,3% | Get TM-Score; Ermittlung des Scores des Übersetzungsmodells |
| cw:EHOEGDMS | 416,194 | 21,6% | Get DM-Score; Ermittlung des Scores des Umordnungssmodells |
| cw:EHOEGWCS | 64,050 | 3,3% | Get WC-Score; Ermittlung des Scores des Wortanzahlmodells |
| cw:EHOEGPCS | 59,745 | 3% | Get PC-Score; Ermittlung des Scores des Phrasenanzahlmodells |
| cw:EHOEGLMS | 422,702 | 21,8% | Get LM-Score; Ermittlung des Scores des Sprachmodells |
| cw:EHOESH | 730,493 | 26,2% | Store Hypothesis; Speichern der aktuell bearbeiteten Hypothese |
| cw:EHOETL | 2293,243 | 100,00% | Total Loop Time |

Auf dieser Ebene (Abbildung 5: Laufzeitanalyse ExpandHypOverEdge, Seite 30) erkennt man auf den ersten Blick unabhängige Programmteile, die ggf. parallel ausgeführt werden können.

Innerhalb der Sprachmodelle ist kaum nennenswertes Potenzial für eine feingranulare Parallelisierung zu finden, da die Hauptarbeit von der Funktion `LogProb()` erledigt wird, die an dieser Stelle nicht parallelisierbar ist.

`StoreHypothesis()` muss nach Abschluss aller anderen Programmteile synchronisiert ausgeführt werden. Deshalb wird explizit untersucht, ob diese Methode beschleunigt werden kann.

### 6.2.5. Laufzeitanalyse StoreHypothesis

| TracePointer | Zeit [s] | Anteil | Beschreibung |
|---|---|---|---|
| | | | **StoreHypothesis** |
| cw:SHInit | 168,551 | 27,55% | Init |
| cw:SHIFHFP | 76,000 | 12,42% | Iter Find Hyp For Pruning |
| cw:SHT | 611,895 | 100,00% | Total |

Wie am Namen erkennbar ist, speichert die Methode `StoreHypothesis` die Hypothese im Lattice, nachdem diese expandiert und bewertet wurden.

Sie enthält keine Schleifen, die Potenzial für die Parallelisierung erkennen lassen; ein Großteil der Prozedur ist zwingend sequenziell mit zahlreichen Daten- und Kontrollabhängigkeiten. Weitere Teile rufen Systemmethoden (`Iter … find`) auf, die hier nicht veränderbar sind.

### 6.2.6. Laufzeitanalyse Gesamt / Expand Hyp over Edge

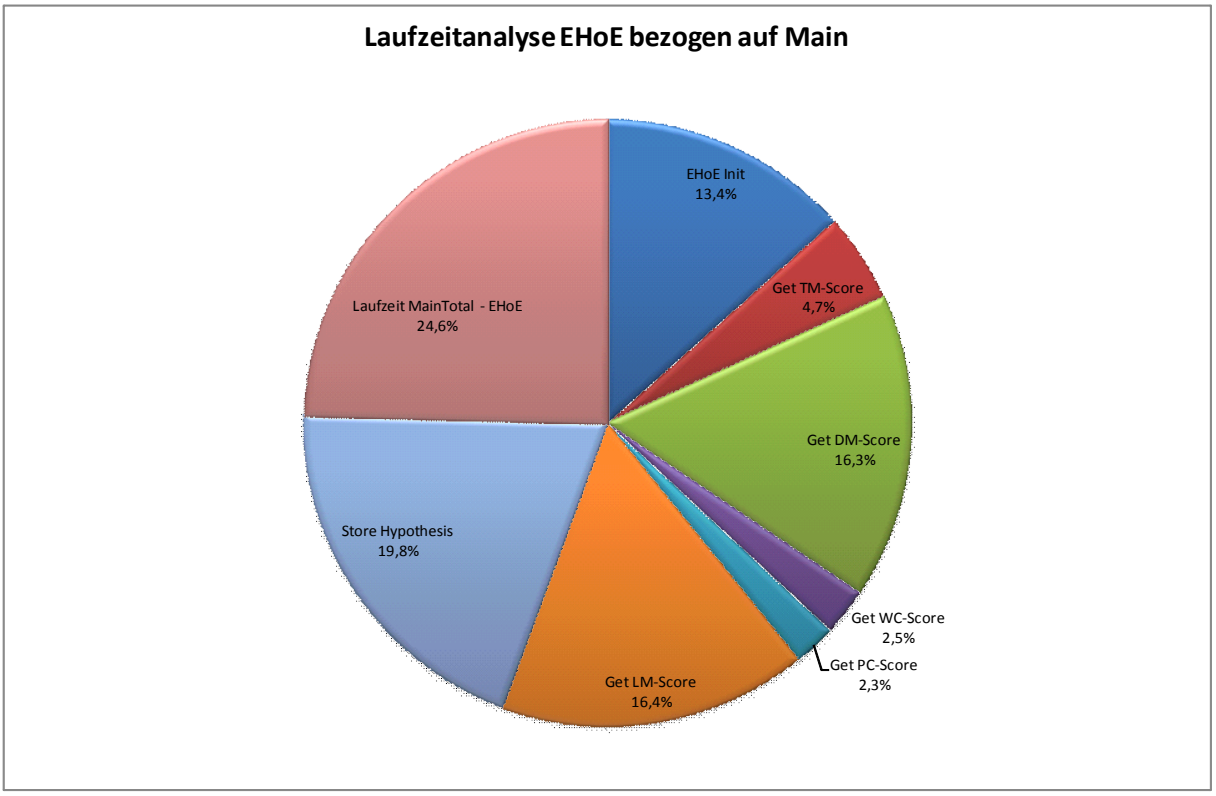


*Abbildung 6: Laufzeitanalyse Gesamt / ExpandHypOverEdge*

Die Methode `ExpandHypOverEdge()` erweitert den bestehenden Lattice um eine weitere Übersetzungshypothese in Form einer Kante.

Im obigen Schaubild (Abbildung 6: Laufzeitanalyse Gesamt / ExpandHypOverEdge) werden die laufzeitrelevanten Teile der Methode `ExpandHypOverEdge()` ins Verhältnis zur gesamten Ausführungszeit des Programms gesetzt. Der Anteil im linken oberen Quadraten `(MainTotal – ExpandHypOverEdge() )` entspricht dabei der Ausführungszeit aller vorangehenden Programmteile. Dadurch wird sichtbar, wie viel Rechenzeit für die Initialisierung und den Verwaltungsaufwand (24,6%), im Verhältnis zur Zeit für den eigentlichen Algorithmus (75,4%) benötigt wird.

Die Parallelisierung an diesem Punkt erscheint Erfolg versprechend, da diese Methode viel der verbrauchten Laufzeit beansprucht und häufig aufgerufen wird.

### 6.2.7. Analyse der einzelnen Aufrufe Expand Hyp

Insgesamt wurde die Methode `ExpandHyp()` 14.712 Mal aufgerufen. Die durchschnittliche Laufzeit dieser Aufrufe beträgt 166 Millisekunden, der kürzeste Durchlauf benötigt ca. 0,3 Millisekunden, der längste ca. 677 Millisekunden.

In der folgenden Grafik (Abbildung 7: Laufzeitverteilung ExpandHyp-Aufrufe (aufsteigend sortiert)) sind diese Aufrufe nach ihrer Ausführungszeit in Nanosekunden angetragen. Man erkennt eine recht gleichmäßige Verteilung der Ausführungsdauern mit Ausnahme der Randbereiche: Circa 2.000 Aufrufe terminieren kurzfristig, während nur ca. 1.000 Durchläufe länger als 400 Millisekunden für ihre Ausführung benötigen.

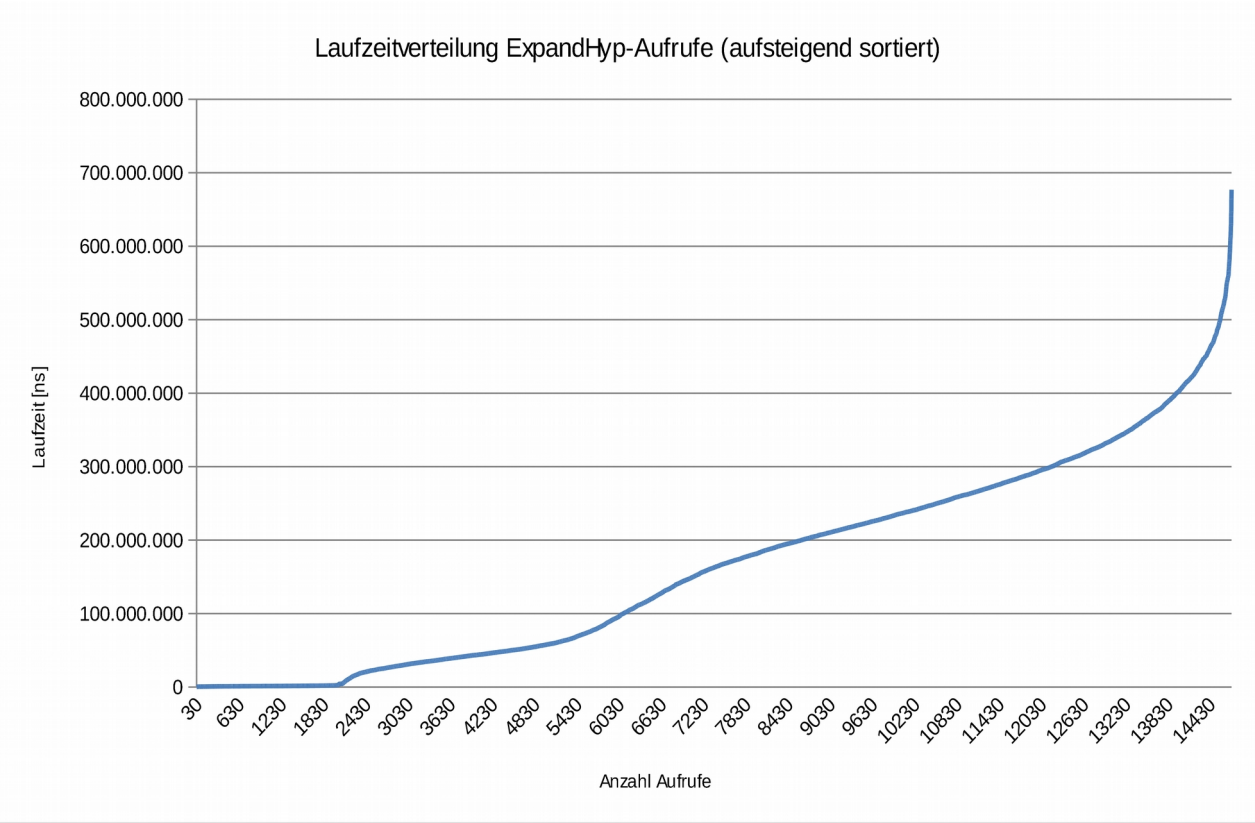


*Abbildung 7: Laufzeitverteilung ExpandHyp-Aufrufe (aufsteigend sortiert)*

Das folgende Schaubild (Abbildung 8: Laufzeiten ExpandHyp-Aufrufe (reihenfolgetreu)) zeigt die Aufrufdauern in der tatsächlichen Reihenfolge ihres Eintritts:

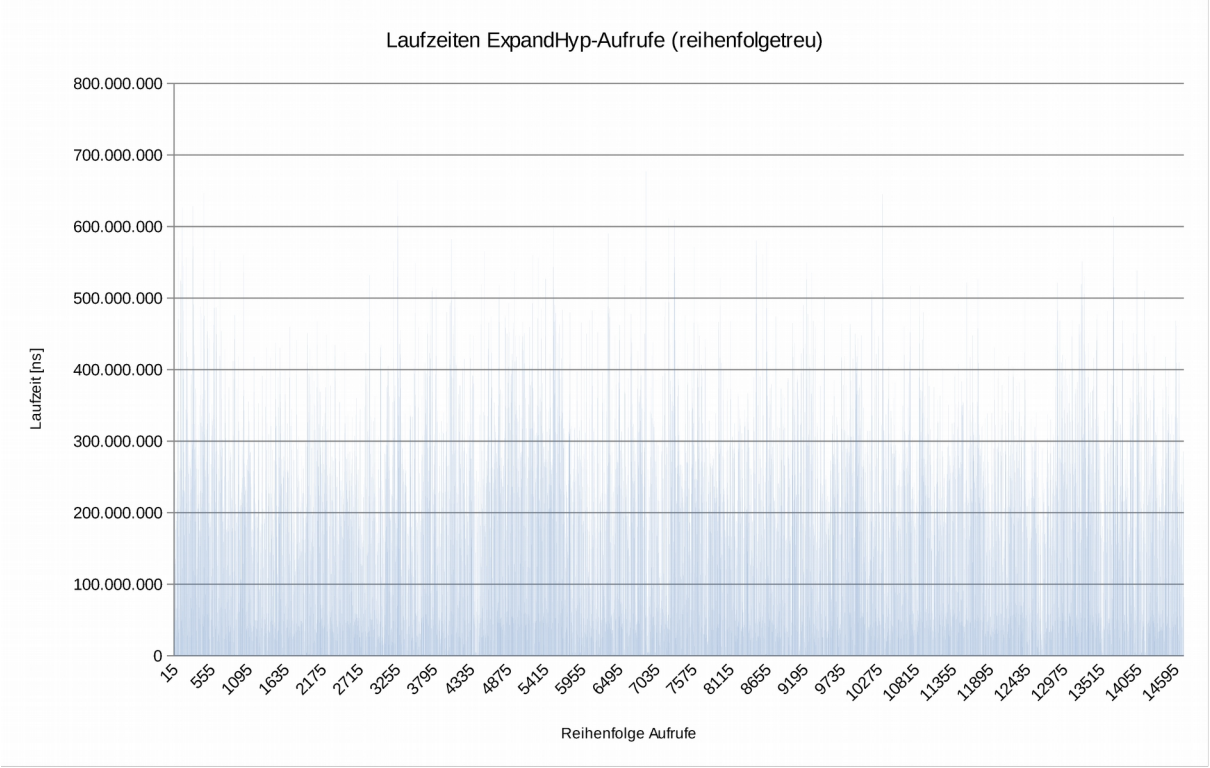


*Abbildung 8: Laufzeiten ExpandHyp-Aufrufe (reihenfolgetreu)*

Da für eine effiziente Parallelisierung die Laufzeiten der nebenläufigen Befehlsfäden, die im Anschluss wieder synchronisiert werden müssen, nicht stark schwanken sollten, ist es notwendig, die Laufzeiten dieser Methode zu vergleichen.

Wie zu erkennen ist, sind diese weitgehend gleichmäßig verteilt und unterliegen keinen auffälligen Einflüssen oder Tendenzen, die eine gesonderte Behandlung erforderlich machen würde.

Insofern ist die Anforderung, dass die Aufrufdauern, der zu parallelisierenden Methode nicht zu stark variieren dürfen, erfüllt.

## 6.3. Implementierung der Zeitmessung

Um die Arbeitsdauer der einzelnen Schritte der Programmdurchläufe in Abhängigkeit der beteiligten Prozessoren messen und vergleichen zu können, wurden an mehreren Stellen Operationen zur Zeitmessung eingefügt. Da die Zeitmessung lediglich für das Benchmarking benutzt werden soll, für die spätere Benutzung des Programms aber nicht benötigt wird, wird die Messung so implementiert, dass sie durch einen Präprozessor nur bei Bedarf aktiviert werden kann. Somit ist es möglich, zur Compile-Zeit zu entscheiden, ob das Programm in der Lage sein soll, Laufzeiten zu erfassen oder nicht.

Zur zentralen Steuerung der Parallelität und der Aggregation der Messergebnisse wurde die Klasse `MultiThreading` eingeführt.

### 6.3.1. Zentraler Schalter für die Zeitmessung

Die Definition `COMPILE_TIMEMEASUREMENT` dient dazu alle Elemente zur Zeitmessung auf Compiler-Ebene zu deaktivieren. Wird sie auskommentiert (`#`) findet die Zeitmessung keinen Eingang in den ausführbaren Programm-Code und beeinflusst somit die Laufzeiten des Programms nicht:

multithreading.hh Zeile 3:

```
...
#define COMPILE_TIMEMEASUREMENT
...
```

### 6.3.2. Zählung der Aufrufanzahl bestimmter Methoden

Um beurteilen zu können, ob eine Funktion oder eine Methode für die Parallelisierung geeignet ist, wird die Anzahl der Durchläufe der zu untersuchenden Funktion gezählt. Um dies zu erreichen, werden in der Klasse `multithreading.hh` zentrale Variablen eingeführt und die entsprechenden Funktionen instrumentiert.

Exemplarisch wird hier die Variablendeklaration der Funktionen `ExpandHyp()` und `ExpandHypOverEdge()` gezeigt:

multithreading.hh Zeile 7:

```
...
extern unsigned long     NumOfExpandHypCalls;
extern unsigned long     AvgDurationExpandHypCall;
extern unsigned long     ShortestExpandHypCall;
extern unsigned long     LongestExpandHypCall;

extern unsigned long     NumOfExpandHypOverEdgeCalls;
extern unsigned long     AvgDurationExpandHypOverEdgeCall;
extern unsigned long     ShortestExpandHypOverEdgeCall;
extern unsigned long     LongestExpandHypOverEdgeCall;
...
```

### 6.3.3. Variablen zur Zeitmessung

Um weiterhin erkennbar zu machen, wie lange eine Funktion für die summierten Durchläufe benötigt, werden externe Variablen und Messpunkte eingeführt, die eine Bestimmung der von einem Prozessor in einer Methode verbrauchten Zeit erlauben. Auch dies dient der Beurteilung, ob eine Methode zur Parallelisierung geeignet ist.

Zuerst wird ein eigener Typ für die Zeitmessung definiert um diesen gegebenenfalls einfach global ändern zu können. Da der High-Precission-Timer des Betriebssystems den Rückgabewert des Zeitstempels in zwei Teilen, nämlich einem `long`-Wert für Nanosekunden und einem für Sekunden liefert, wurden zwei `long`-Variablen definiert.

Die Definitionen erfolgen für das Hauptprogramm `translate09`, die Klasse `Decoder-BLM()`, `ExpandHypOverEdge()` und `StoreHypothesis()`.

Die Implementierung wird im Anhang IV (Implementierung der Zeitmessung) dargestellt.

## 6.4. Auswertung und Auswahl der Granularität

Als mögliche Ansatzpunkte für die Parallelisierung bieten sich nun drei Stellen an, die sich maßgeblich im Grad ihrer Granularität unterscheiden. Eine genaue Beschreibung der Punkte folgt nach deren Aufzählung:

1. die Schleife, die über die zu übersetzenden Sätze iteriert, mit der Hauptmethode `TranslateSentence()`
2. die Anwendung der Sprachmodelle und die Methode `StoreHypothesis()` innerhalb `ExpandHypOverEdge()`
3. die Schleife, die die Hypothesen expandiert, `ExpandHypOverEdge()`

Die parallele Verarbeitung ganzer Sätze erscheint auf den ersten Blick als der am meisten Erfolg versprechende Ansatzpunkt für eine effiziente mehrfädige Verarbeitung: Die Phrasentabelle und die Sprachmodelle können im gleichen Speicherbereich verbleiben, da die Zugriffe auf sie nur lesender Natur sind; für jeden Satz kann ein unabhängiger Lattice aufgebaut werden. Jeder Kern würde dann parallel einen Satz übersetzen.

Da dies die Übersetzungsgeschwindigkeit innerhalb eines einzelnen Satzes nicht erhöhen würde, würde dieser Ansatz der Anforderung, das Toolkit für Simultanübersetzungen zu nutzen, nicht genügen.

Einen sehr viel feineren Grad an Granularität verspricht der zweite Ansatzpunkt, die Anwendung der Sprachmodelle und die Speicherung der Hypothesen im Lattice: Die Modelle könnten parallel angewendet werden, lediglich die Methode `StoreHypothesis()` dürfte erst nach dem Abschluss des letzten Modells starten.

Man könnte also die fünf Modelle (Übersetzungsmodell TM, Umordnungsmodell DM, Wortanzahl-Modell WC, Phrasenanzahl-Modell PC, Sprachmodell(e) LM) parallel ausführen und im Anschluss die Hypothese speichern. Wenn man dabei die Zahlen aus Kapitel 6.2.6 (Laufzeitanalyse Gesamt / Expand Hyp over Edge) zugrunde legt, und das Programm (1) in den parallel ausführbaren Teil P und den sequenziell auszuführenden (1-P) unterteilt ergäbe sich nach Amdahl folgender Geschwindigkeitsvorteil:

Sei N= 5, für den parallelen Anteil P gilt:

| **Methode** | **Anteil Gesamtlaufzeit** |
|---|---|
| `Get TM-Score()` | 4,7% |
| `Get DM-Score()` | 16,3% |
| `Get WC-Score()` | 2,5% |
| `Get PC-Score()` | 2,3% |
| `Get LM-Score()` | 16,4% |
| | |
| **Σ** | **42,2%** |

für den sequentiellen Programmanteil 1-P gilt dann:

| **Methode** | **Anteil Gesamtlaufzeit** |
|---|---|
| Übrige Laufzeit (Initialisierung, Ausgabe, ...) | 24,6% |
| Initialisierung `ExpandHypOverEdge` | 13,4% |
| `StoreHypothesis()` | 19,8% |
| | |
| **Σ** | **57,8%** |

Setzt man diese Werte unter Vernachlässigung der Initialisierungszeit der Threads in die Amdahlsche Formel aus Kapitel 4.2 (Amdahlsches Gesetz) ein, ergibt sich der maximale Geschwindigkeitsgewinn S von ca. 51%:

$$S = \frac{1}{57{,}8\,\% + \frac{42{,}2\,\%}{5}} = 1{,}51$$

Da aber noch zu berücksichtigen ist, dass nicht alle parallel ausführbaren Programmteile (P) die gleiche Ausführungszeit benötigen und nach deren Abschluss eine Synchronisationsbarriere eingeführt werden muss ( `StoreHypothesis()` ), ist die hypothetische Grenzfunktion P* durch eine untere Schranke begrenzt auf:

$$P^* = \max \begin{cases} \max p_n \ (\forall n \in N) \\ \\ \frac{P}{N} \end{cases}$$

Daraus ergäbe sich der maximal zu erwartende Gewinn:

$$S = \frac{1}{57{,}8\,\% + 16{,}4\,\%} = 1{,}35$$

Von diesen ca. 35% müssten weiterhin der Initialisierungsaufwand der parallelen Befehlsfäden sowie die Wartezeiten an Synchronisationsbarrieren abgezogen werden.

Angesichts dieser überschaubaren Erfolgssaugsichten wurde dieser Ansatz ebenfalls verworfen.

Die Funktion `ExpandHypOverEdge()` erweitert alle vorliegenden Hypothesen um weitere Worte oder Phrasen. Dabei wird jede bestehende Hypothese traversiert und um neue Kanten erweitert. Dadurch entstehen neue Hypothesen.

In Kapitel 5.2 (Schritte zur Parallelisierung) wurden folgende Anforderungen für einen Erfolg versprechenden Parallelisierungsansatz aufgestellt:

- häufige Wiederholungen
- hinreichende Mindestgröße
- weitgehende Unabhängigkeit hinsichtlich der genutzten Daten
- ähnliche Laufzeit
- Granularität unterhalb satzweiser Parallelität

Die untersuchte Funktion wird bei der Übersetzung des Testsets mit 2.000 Sätzen insgesamt 132.949.240 aufgerufen, pro Satz also durchschnittlich 66.474 Mal.

Die Laufzeit beträgt durchschnittlich ca. 166 ms, was einer hinreichenden Größe entspricht.

Kontrollfluss- und Datenabhängigkeiten sind vorhanden aber lösbar.

Die maximale Laufzeit eines Durchlaufs liegt bei 677 ms, die minimale bei 0,3 ms. Der Durchschnitt liegt bei 166 ms, der Median bei 163 ms. Die Verteilung der Laufzeiten stellt sich gleichmäßig dar, ohne Ausreißer oder Häufungen. Die Laufzeiten der Methodenaufrufe schwanken also nur moderat.

Deshalb genügt diese Methode allen gestellten Anforderungen an einen geeigneten Parallelisierungsansatz.

Der gewählte Ansatz setzt darum dort ein und erzeugt neue Hypothesen parallel in unabhängigen Befehlsfäden.

Um Konflikte zu minimieren und damit die Wartezeit an Sperren möglichst gering zu halten, werden weiterhin Hypothesen mit unterschiedlicher Abdeckung des Quellsatzes (Coverage) unabhängig gehandhabt und parallel bearbeitet.

# 7. Parallelisierung

Im folgenden Kapitel werden die Motivation und die Kernelemente der Implementierung beispielhaft dargestellt. Um die Verständlichkeit zu verbessern, wird an einigen Stellen detaillierter auf die Änderungen eingegangen. Weitere wichtige aber dem Verständnis des Vorgehens weniger zuträgliche Änderungen des Codes werden im Anhang C (Details der Implementierung) aufgeführt.

## *7.1. Parallelisierung der Hypothesenexpansion*

Wie in Kapitel 6.4 (Auswertung und Auswahl der Granularität) wurde die Funktion `ExpandHypOverEdge()` ausgewählt und soll parallelisiert werden. Dazu wird bei der Hypothesenexpansion der Befehlsfaden in mehrere Entitäten aufgeteilt und auf verfügbare Prozessoren verteilt. Da zu diesem Zeitpunkt nicht erkennbar ist, welcher Faden welchem Prozessor zugeordnet wird und wann dieser terminiert, ist am Ende des parallelen Abschnitts eine Synchronisationsbarriere notwendig, die von OpenMP automatisch eingefügt wird.

Der in Kapitel 4.5.1 (Darstellung des Programmablaufs) beschriebene Programmablauf soll dabei so umgestellt werden, dass die Hypothesen-Expansion nicht mehr in einer Schleife, sondern parallel erfolgt. (Abbildung 9: Ablaufdiagramm parallelisierter KIT-Decoder, Seite 42)

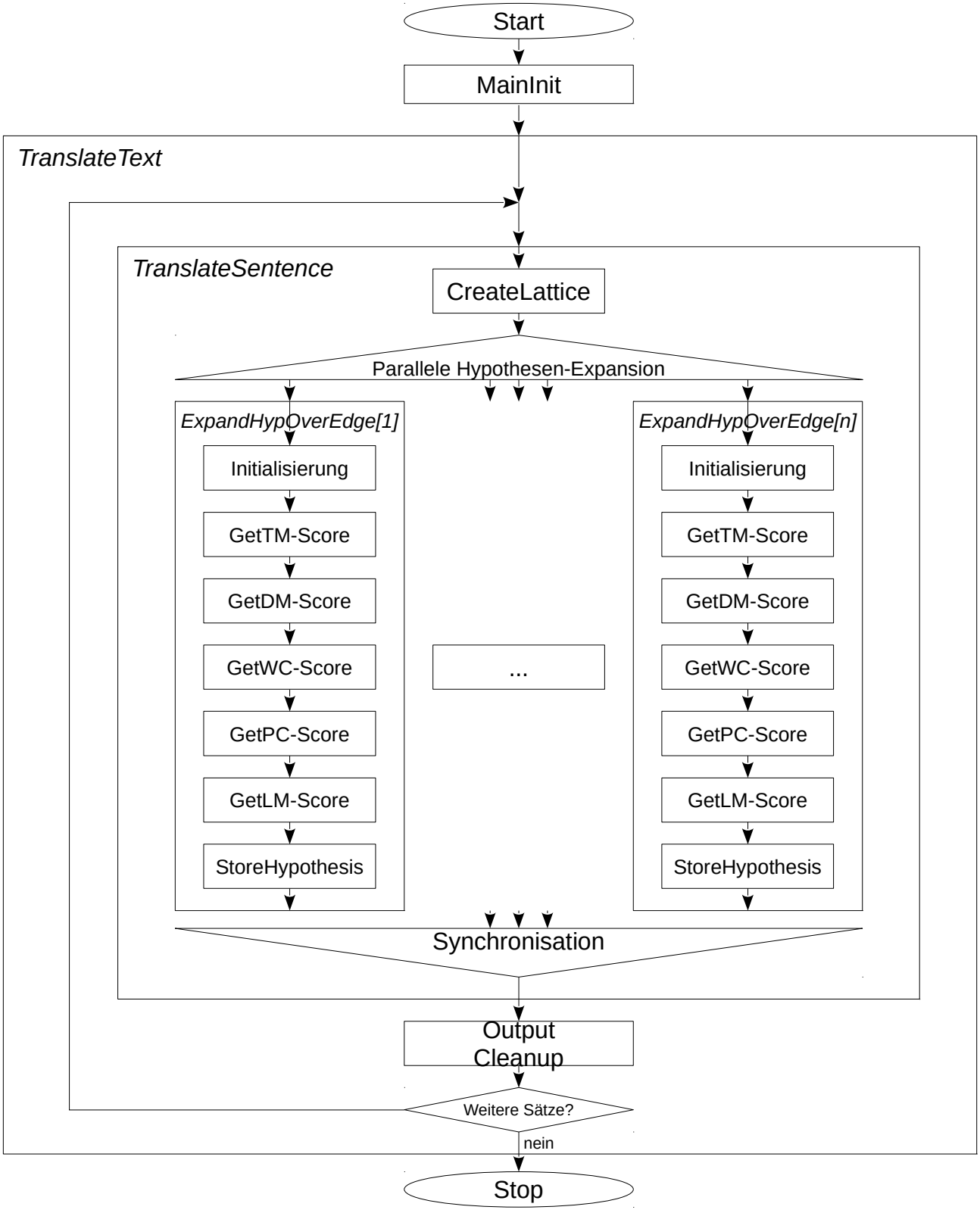


*Abbildung 9: Ablaufdiagramm parallelisierter KIT-Decoder*

Bei der nunmehr parallelen Verarbeitung traten einige der aus 4.1 (Grundlegende Probleme der Parallelverarbeitung) bekannten Probleme, wie die Fähigkeit der benutzen Programmteile nebenläufig zu arbeiten, der konkurrierende Zugriff auf zentrale Listen, sowie die bisher häufig benutzten Übergaben von Objektreferenzen auf.

### 7.1.1. Fähigkeit zur Nebenläufigkeit

Unter „threadsafeness“ versteht man die Fähigkeit von Prozeduren, Funktionen oder Methoden, selbstständig störungsfrei parallel zu weiteren Instanzen des gleichen Programmabschnitts arbeiten zu können. Diese Veranlagung muss bereits bei der Entwicklung berücksichtigt werden und kann nicht von außen eingebracht werden.

Bei der Implementierung der Parallelität erwies sich die Funktion `LogProb()` des Sprachmodells SriLM als dahin gehend problematisch.

Grundsätzlich existieren drei Methoden, Probleme hinsichtlich der threadsafeness zu umgehen. Man kann dabei die betroffene Sektion als „kritisch“ einstufen. Bei OpenMP bedeutet diese Direktive (`#pragma omp critical`), dass ein entsprechend markierter Bereich immer nur von einem Befehlsfaden ausgeführt werden darf. Die einfache Implementierung hat aber ein ungünstiges Laufzeitverhalten zur Folge: Die betroffenen Programmteile sind dann von der Parallelisierung ausgeschlossen.

In dieser Phase der Implementierung konnte zwar mit der Markierung des Sprachmodells als „kritisch“ ein korrekter Programmablauf nachgewiesen werden, ein Geschwindigkeitsgewinn hingegen nicht.

Ein weiterer möglicher Lösungsweg wäre die Neuimplementierung des Sprachmodells. Da dies weder dem Umfang noch dem Ziel dieser Arbeit entsprach, wurde davon abgesehen.

Da der KIT-Decoder so entwickelt wurde, dass die Sprachmodelle austauschbar sind, wurde nach einem Sprachmodell gesucht, das die benötigte Fähigkeit zur Nebenläufigkeit bereits enthält. Entsprechend wurde für die weitere Arbeit anstatt des bisherigen SriLM das Sprachmodell KenLM eingesetzt.

### 7.1.2. Konkurrierender Zugriff auf zentrale Listen

Da bei der bisherigen Implementierung des Decoders immer nur ein Befehlsfaden aktiv war, war der wahlfreie (lesende und schreibende) Zugriff auf die beiden globalen Listen AllHyps und BestHyps unproblematisch. Bei mehreren parallelen Zugriffen auf diese Listen können die in 4.1.2 vorgestellten Datenabhängigkeiten auftreten und Inkonsistenzen verursachen.

Eine simple Semaphore, die immer nur einen wahlfreien Zugriff erlaubt, wäre eine sehr starke Einschränkung, die den Programmablauf an dieser Stelle stark verlangsamen würde.

Aus diesem Grund wurde das weniger hemmende Konstrukt des Readers/Writers-Locks gewählt: Es erlaubt beliebig viele gleichzeitige lesende Zugriffe aber nur einen Schreibenden zur gleichen Zeit.

In die Methode `CdecoderBLM::ApplyLanguageModel()` fand dies folgendermaßen Eingang:

DecoderBLM.cc:

```
...
  //cw:
  if (g_allHypsLock!=NULL) delete[]g_allHypsLock;
  g_allHypsLock=new CReadersWriters[J - Start +1];
  if (g_bestHypsLock!=NULL) delete[]g_bestHypsLock;
  g_bestHypsLock=new CReadersWriters[J - Start +1];
  //cw: End
...
```

*Der grün dargestellte Teil stellt die Veränderung dar.*

Für den Fall, dass die Methode mehrfach aufgerufen wird, wird zuerst überprüft, ob ggf. ein Lock existiert und dieses vorsorglich entfernt.

Die Methode `CDecoderBLM::ExpandAllHypsWithCoverageBeamWidthPruning( int Coverage )` greift nun mithilfe dieses Schutzes auf diese beiden Listen zu:

DecoderBLM.cc:

```
...
// first we fetch the set of hyps which all have coverage c
// and we need an iterater to access the hyps in this set
g_allHypsLock[Coverage].lockReader(); //cw
HypContainer< CDecoderBLM_Hyp *, HypHash, HypEqual > &Hyps =
m_AllHyps[ Coverage ];
g_allHypsLock[Coverage].unlockReader(); //cw
HypContainer< CDecoderBLM_Hyp *, HypHash, HypEqual
>::iterator IterHyps;

// for pruning we need the best hyps with the current covrg
g_bestHypsLock[ Coverage ].lockReader(); //cw
HypContainer< CDecoderBLM_Hyp *, HypHashForPruning,
HypEqualForPruning > &BestHyps = m_BestHyps[ Coverage ];
g_bestHypsLock[ Coverage ].unlockReader(); //cw
HypContainer< CDecoderBLM_Hyp *, HypHashForPruning,
HypEqualForPruning >::iterator IterBestHyps;
...
```

*Der grün dargestellte Teil stellt die Veränderung dar.*

Analog dazu wurde mit der Methode `CDecoderBLM::ExpandAllHypsWithCoverageBeamSizePruning( int Coverage )` verfahren.

#### *7.1.2.1.Definition Readers/Writers Lock*

Das Reader/Writers-Lock ist ein potenziell leistungsfähigerer Ausschlussmechanismus als ein simpler Mutex, da bei dieser Sperre beim Zutritt in einen kritischen Bereich zwischen lesenden und schreibenden Zugriffen unterschieden wird. So erhalten beliebig viele Teilnehmer gleichzeitigen lesenden Zugriff, was die Wartezeiten erheblich reduzieren kann.

In das Readers/Writers-Lock wurde zusätzlich noch die Zeitmessung implementiert, um nachvollziehen zu können, wie viel Zeit an den Barrieren der Schlösser zugebracht wird.

Die vollständige Implementierung findet sich in multithreading.cc, hier lediglich der Konstruktor und der Destruktor:

multithreading.cc

```
...
CReadersWriters::CReadersWriters()
{
CReadersWriters::CReadersWriters()
     {
     readCount=0;
     writeCount=0;
     }
void CReadersWriters::operator=(const CReadersWriters& p)
      {
      readCount=p.readCount;
      writeCount=p.writeCount;
      mutex_1=p.mutex_1;
      mutex_2=p.mutex_2;
      mutex_3=p.mutex_3;
      write=p.write;
      read=p.read;
      }
CReadersWriters::CReadersWriters(const CReadersWriters& p)
:mutex_1(p.mutex_1), mutex_2(p.mutex_2), mutex_3(p.mutex_3),
read(p.read), write(p.write)
      {
      readCount=p.readCount;
      writeCount=p.writeCount;
      }
CReadersWriters::~CReadersWriters()
     {
     }
}
...
```

Will ein Befehlsfaden einen kritischen Bereich lesend betreten, so muss geprüft werden, ob bereits ein schreibender Zugriff diesen Bereich gesperrt hat. Wenn dies der Fall ist, muss der lesende Zugriff das Ende des schreibenden abwarten. Zum Eintritt in den kritischen Bereich ist die Methode `lockReader()` aufzurufen, zum Verlassen des Bereichs `unlockReader()` :

multithreading.cc

```
...
void CReadersWriters::lockReader()
{
     timespec enterLock,leaveLock;
     clock_gettime(CLOCK_REALTIME, &enterLock);
     mutex_3.lock();
     read.lock();
     mutex_1.lock();
     readCount = readCount + 1;
     if (readCount == 1) write.lock();
     mutex_1.unlock();
     read.unlock();
     mutex_3.unlock();
     clock_gettime(CLOCK_REALTIME, &leaveLock);
     cwtime sneeded=(cwtime)leaveLock.tv_sec-
(cwtime)enterLock.tv_sec;
     cwtime nsneeded=(cwtime)leaveLock.tv_nsec-
(cwtime)enterLock.tv_nsec;
     #pragma omp atomic
     cw_inAllReadersLock_s+=sneeded;
     #pragma omp atomic
     cw_inAllReadersLock_ns+=nsneeded;
}

void CReadersWriters::unlockReader()
{
     mutex_1.lock();
     readCount = readCount - 1;
     if (readCount == 0) write.unlock();
     mutex_1.unlock();
}
...
```

Beim schreibenden Zugriff auf eine globale Ressource muss diese gesperrt werden. Vor dem Eintritt in den kritischen Bereich muss der Austritt sämtlicher anderen Teilnehmer abgewartet werden. Der Eintritt muss durch Methode `lockWriter()` angezeigt werden, das Verlassen des Bereichs durch `unlockWriter()` :

multithreading.cc

```
...
void CReadersWriters::lockWriter()
{
     timespec enterLock,leaveLock;
     clock_gettime(CLOCK_REALTIME, &enterLock);
     mutex_2.lock();
     writeCount = writeCount + 1;
     if (writeCount == 1) read.lock();
     mutex_2.unlock();
     write.lock();
     clock_gettime(CLOCK_REALTIME, &leaveLock);
     cwtime sneeded=(cwtime)leaveLock.tv_sec-
(cwtime)enterLock.tv_sec;
     cwtime nsneeded=(cwtime)leaveLock.tv_nsec-
(cwtime)enterLock.tv_nsec;
     #pragma omp atomic
     cw_inAllWritersLock_s+=sneeded;
     #pragma omp atomic
     cw_inAllWritersLock_ns+=nsneeded;
}

void CReadersWriters::unlockWriter()
{
     write.unlock();
     mutex_2.lock();
     writeCount = writeCount - 1;
     if (writeCount == 0) read.unlock();
     mutex_2.unlock();
}
...
```

### 7.1.3. Übergabe von Objektreferenzen

In der sequenziellen Implementierung des KIT-Decoders wurde, um Arbeitsspeicher zu sparen, an Unterroutinen häufig lediglich die Referenz auf ein Objekt übergeben. Dabei wird der aufgerufenen Funktion lediglich ein Zeiger auf eine Speicherstelle als Übergabeparameter bekannt gegeben, an der sich das zu bearbeitende Objekt befindet. Die Berechnung erfolgt dann direkt auf diesem Objekt, die Ergebnisse werden direkt dort hinterlegt und von der aufrufenden Funktion weiter verarbeitet. Dieses Vorgehen erspart dem System das mehrfache Anlegen von Objekten und deren Übergabe an die aufgerufene Funktion.

Bei paralleler Verarbeitung ist es aber möglich, dass eine solche Funktion mehrfach gestartet wird und damit mehrere Entitäten auf einem Objekt arbeiten, was zu Inkonsistenzen und weiteren Konflikten führen kann (siehe Kapitel 4.1.2 Datenabhängigkeiten).

Deshalb musste der Programm-Code an mehreren Stellen dahin gehend geändert werden, dass in Abhängigkeit der Befehlsfäden für je einen Prozess ein eigenes Objekt zur Verfügung steht.

Im Konstruktor der Klasse `CDecoderBLM` wurde bislang global eine neue Hypothese erzeugt. Diese Erzeugung muss entsprechend der Anzahl der Befehlsfäden erweitert werden, sodass jeder Faden eine eigene Arbeitshypothese erhält:

DecoderBLM.cc:

```
...// the standard working hypothesis
//m_pNumberOneHypothesis = NewHypothesis();
m_pDecoderInfo = new CDecoderBLM_Info;
// m_pNumberOneHypothesis->m_pDecoderInfo = m_pDecoderInfo;
for (int i = 0 ; i<cw_num_threads ; i++)
{
     CDecoderBLM_Hyp* pHyp = NewHypothesis();
     pHyp->m_pDecoderInfo = m_pDecoderInfo;
     m_pNumberOneHypothesis.Append(pHyp);
}...
```

*Der orange dargestellte Teil entspricht dabei dem ursprünglichen und entfernten Code, der grün gefärbte stellt die Veränderung dar.*

Dies muss auch im Destruktor bedacht werden:

DecoderBLM.cc:

```
...
CDecoderBLM::~CDecoderBLM()
{
  Clear();
  //cout<<"Not removing the last numberOneHypothesis in
~DecoderNB"<<endl;
  //if (m_numInstances==1){
  //delete pNumberOneHypothesis;
  //delete m_pNumberOneBestHyp;
  //}
  for (int i = 0 ; i<m_pNumberOneHypothesis.Size() ; i++)
  {
       // delete m_pNumberOneHypothesis[i]; //cw: causes
same trouble as before in single threaded version
  }
  //cout<<"Remove decoder info"<<endl;
  delete m_pDecoderInfo;
  CDecoderBLM::m_NumberOfInstances--;
  //cw:
  if (m_NumberOfInstances==0)
  {
          if (g_allHypsLock!=NULL) delete[]g_allHypsLock;
          if (g_bestHypsLock!=NULL) delete[]g_bestHypsLock;
  }
  //cw: End
}
...
```

*Der grün dargestellte Teil stellt die Veränderung dar.*

Die Erzeugung einer neuen Kante in der Methode `CDecoderBLM::ExpandHyp()` übergab in der nicht parallelisierten Version ebenfalls nur eine Referenz auf ein Objekt, das außerhalb erzeugt wurde. Die Erzeugung wird in den parallelen Bereich gezogen und damit pro Befehlsfaden erzeugt.

DecoderBLM.cc:

```
...
// variable we need
CLatticeNode *pNode;
//CLatticeEdgeMS *pEdge; //wird lokal erzeugt
...
```

*Der grün dargestellte Teil stellt die Veränderung dar.*

Paralleler Bereich:

DecoderBLM.cc:

```
...
CLatticeEdgeMS *pEdge;
pEdge = (CLatticeEdgeMS *)Edges[ EdgeNum ];
...
```

*Der grün dargestellte Teil stellt die Veränderung dar.*

Ähnlich verhält es sich bei der Methode `CDecoderBLM::ExpandHypOverEdge( CDecoderBLM_Hyp *pHyp, CLatticeEdgeMS *pEdge, CDecoderBLM_Hyp *pNumberOneHypothesis )`. Auch diese muss dahin gehend abgeändert werden, dass sie nicht in die globale Liste NumberOneHypothesis schreibt, sondern eine Referenz auf eine eigene Hypothese zurück gibt.

Analog wurde mit allen weiteren Methoden verfahren, die Scores ermitteln.

### 7.1.4. StoreHypothesis()

Besondere Aufmerksamkeit muss der Speicherung der Hypothesen zu Teil werden, denn hier wurden die Listen AllHyps und BestHyps geschrieben und müssen entsprechend gesperrt und wieder frei gegeben werden.

Da die Hypothesen-Expansion getrennt für jede Abdeckung (Coverage) stattfindet, wurden um das Auftreten von echten Sperren weiter zu vermindern, die Readers/Writers-Locks pro Coverage-Basket angelegt.

DecoderBLM.cc:

```
...

// copy information to NumberOneBestHyp

//*m_pNumberOneBestHyp = *pNumberOneHypothesis;
g_bestHypsLock[ NewCoverage ].lockWriter(); //cw

IterFindHypForPruning =
m_BestHyps[ NewCoverage ].find( pNumberOneHypothesis );

...
```

*Der orange dargestellte Teil entspricht auch hier dem ursprünglichen und entfernten Code, der grün gefärbte stellt die Veränderung dar.*

Analog wurde mit folgenden Zeilen der gleichen Klasse verfahren, die den Zugriff auf die Liste aller Hypothesen realisieren:

DecoderBLM.cc:

```
...
     *pBestHypForPruning = *pNumberOneHypothesis;
     //#pragma omp end critical (CW_SH)
     } else {
          TraceIf( s_TraceOn, 0, "      Prune: Hyp is worse
than stored one, don't use it" );
     }
     g_bestHypsLock[ NewCoverage ].unlockWriter(); //cw
}...
```

In der Methode `CDecoderBLM::StoreHypothesis(CDecoderBLM_Hyp *pNumberOneHypothesis)` muss beim schreibenden Zugriff auf die AllHyps-Liste ein Writer-Lock gesetzt werden, welches aufgrund einer Verzweigung (Trace) nach dem Zugriff auf die AllHyps-Liste an zwei Stellen freigegeben werden muss.

Darüber hinaus erfuhr die Klasse `CDecoderBLM` folgende Änderungen:

- Die Methode `CDecoderBLM::DisplayAllHypsWithCoverage( int Coverage, int N )` musste um ein Reader-Lock erweitert werden, um die korrekte Ausgabe der Hypothesen zu gewährleisten.
- Die Erzeugung einer neuen Recombined Map in der Methode `CdecoderBLM::NewRecombinedMap()` konnte wegen des Ausschlusses durch das Readers/Writers-Lock als kritischer Bereich ausgenommen und weiterhin parallel ausgeführt werden.

- Die Zugriffe auf AllHyps und BestHyps in `CDecoderBLM::GetLMLogProbsAllHypsWithCoverage( int Coverage )` erhielten in gleicher Weise wie oben Reader-Locks.
- Die Methode `CDecoderBLM::ExpandHyp( CDecoderBLM_Hyp *pHyp )` iteriert über alle Kanten des aktuellen Knotens des Lattices. An dieser Stelle setzt die Parallelisierung ein:

DecoderBLM.cc:

```
...
        //cw: begin of parallel region
        #pragma omp parallel for
        for ( int EdgeNum = min;
                EdgeNum <= max;
                EdgeNum++ ) {
...
```

*Der grün dargestellte Teil stellt die Veränderung dar.*

### 7.1.5. Ausgabe der Messwerte

Zur besseren Übersicht und einfacheren Überprüfung der ermittelten Werte, wurde eine Funktion eingeführt, die im Anschluss an die Übersetzung des letzten Satzes die gesammelten Daten auf die Standardausgabe schreibt.

Die beispielhafte Ausgabe für einen Programm-Durchlauf mit einem Programm-Faden sieht dabei folgendermaßen aus:

Ausgabe der Messwerte:

```
...
cw:NumOfExpandHypCalls;1804272
cw:AvgDurationExpandHypCall;199152
cw:ShortestExpandHypCall;3040
cw:LongestExpandHypCall;19857805
cw:NumOfExpandHypOverEdgeCalls;132949240
cw:AvgDurationExpandHypOverEdgeCall;468
cw:ShortestExpandHypOverEdgeCall;468
cw:LongestExpandHypOverEdgeCall;11253761
cw:EHOE Init;67.8141
cw:EHOE GetTMScore;19.1373
cw:EHOE GetDMScore;65.6343
cw:EHOE GetWCScore;4.30155
cw:EHOE GetPCScore;3.65372
cw:EHOE GetLMScore;836.668
cw:EHOE_StoreHypothesis;113.135
cw:EHOE_TotalLoop;1120.43
cw:Store Hypothesis Total;104.991
cw:Store Hypothesis Init;40
cw:Store Hypothesis IFHFP;6.69307
cw:EHInit;0
cw:EHFindBestHypForPruning;19.2027
cw:EHFindBeamsizeBest;0.354171
cw:EHFindMHPCBest;1161.63
cw:EHTotalLoops;1181.24
cw: Time spent in all readers locks:69.1725
cw: Time spent in all writers locks:21.9694
cw: Time spent in LogProb:719.97
```

Dabei ist bemerkenswert, dass die Locks auch in einem Single-Thread-Betrieb Rechenaufwand verursachen.

# 8. Validierung der Ergebnisse und Messungen des parallelen Codes

## 8.1. Überprüfung der Ergebnisse

Um sicherzustellen, dass die Implementierung korrekte Ergebnisse liefert, wurden die Ausgaben mithilfe des Programms WinMerge einander gegenüber gestellt. WinMerge [WM2015] ist ein Programm, das zwei Quelldateien gleichzeitig darstellt und Unterschiede zwischen den Dateien farblich hervorhebt.

Hierzu dienten 2.000 Sätze, die von Englisch auf Deutsch mit einem 4-gram-Sprachmodell übersetzt wurden.

Der erste Durchlauf erfolgte einfädig, der Prüflauf in vierfädiger Parallelität, die Ausgabe wurde jeweils verkürzt auf die Ausgabe der besten Übersetzung.

Die Überprüfung ergab identische Ergebnisse.

## 8.2. Messungen des parallelen Codes

Die Messung der Übersetzungsgeschwindigkeit erfolgt dynamisch, wie zuvor in Kapitel 6.1 (Einsatz von Messpunkten) beschrieben. Daneben wurden mehrere Methoden zusätzlich instrumentiert um detailliertere Auswertungen zu ermöglichen:

- Die Initialisierungszeit wurde gesondert aufgenommen.
- Die Zeit, die die Funktion `LogProb()` benötigt wurde gesondert aufgezeichnet, da sie die Hauptmethode des Sprachmodells ist.
- Die Zeit, die an den Sperren verbracht wurde, wurde gesondert notiert.

Um die Messungen möglichst verzerrungsfrei durchführen zu können, wurden jeweils Rechner gewählt, die zum Zeitpunkt der Messung keine anderen Aufgaben bearbeiteten.

### 8.2.1. Messung mit Sprachmodell-Toolkit SriLM

Auf dem High-Performance-Cluster (i13hpc30) stehen 64 Prozessorkerne und 512GB Hauptspeicher zur Verfügung. Auf ihm wurden mit SriLM Übersetzungen von Englisch auf Französisch ausgeführt. Dabei wurde die Zeit der Übersetzung ermittelt (in der folgenden Abbildung 9: Ablaufdiagramm parallelisierter KIT-Decoder, Seite 42, blau dargestellt) sowie die Zeit die für die Initialisierung des Decoders, also das Laden der Sprachmodelle und der Phrasentabelle (grün dargestellt). Weiterhin wurde die Zeit gemessen, die die Funktion `LogProb()` des Sprachmodells verbraucht und durch die Anzahl der aktiven Befehlsfäden dividiert (rot dargestellt). Sie wird gesondert untersucht, da sie im Sprachmodell SriLM hauptsächlich aufgerufen wird und nicht parallelisierbar ist.

An der Abszisse ist die Anzahl der Befehlsfäden angetragen, an der Ordinate die Laufzeit in Sekunden.

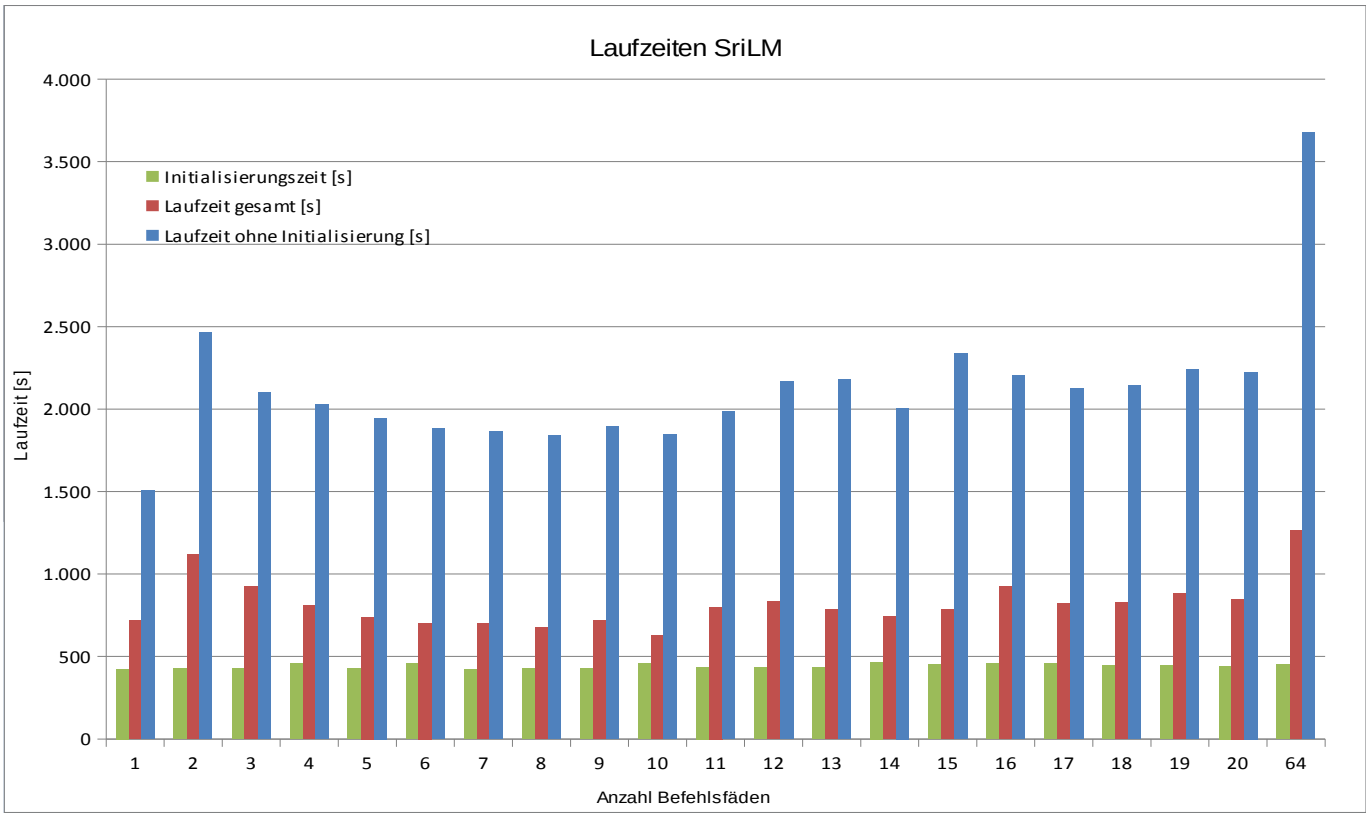


*Abbildung 10: Laufzeiten mit SriLM (Initialisierung, LogProb (realtiv), Gesamt)*

Wie ersichtlich ist, steigt die Ausführungszeit des gesamten Programms (blau) bei der Zuschaltung der Mehrfädigkeit, aufgrund des mit der Initialisierung der parallelen Befehlsfäden verbundenen Aufwands, stark an. So benötigt die Ausführung der Übersetzung mit zwei Fä-

den gegenüber der rein sequenziellen fast 60% mehr Zeit. Erst danach fällt sie leicht ab, um bei acht Fäden ihr Minimum im Parallelbetrieb zu erreichen. Danach steigt sie wieder an und nimmt tendenziell immer weiter zu. Der maximal gemessenen Wert liegt bei 64 benutzten Kernen, da dies der Anzahl der Kerne des benutzten Multiprozessorsystems entspricht.

Die Initialisierungszeit bleibt über alle Messungen hinweg weitgehend konstant.

Die Bearbeitungszeit der Funktion `LogProb()` im Sprachmodell zeigt einen ähnlichen Kurvenverlauf wie die Gesamtlaufzeit, sofern sie pro Kern angetragen wird. Ihr nomineller Anteil steigt korrelierend zur Anzahl der Befehlsfäden an.

Im folgenden Diagramm (Abbildung 11: Wartezeit Sperren und LogProb Sri LM (relativ)) sind v.o.n.u. die Ausführungszeiten der Funktion `LogProb()` (rot, dividiert durch die Anzahl der Threads), die kumulierten Wartezeiten an allen Writer-Locks (dunkelblau, realtiv zur Anzahl der Threads), die kumulierten Wartezeiten an allen Reader-Locks (grün, relativ zur Anzahl der Threads), sowie die absolute Initialisierungszeit (hellblau) des Programms angetragen.

Auf der Abszisse wird die Anzahl der beteiligten Kerne und auf der Ordinate die verstrichene Zeit bezeichnet.

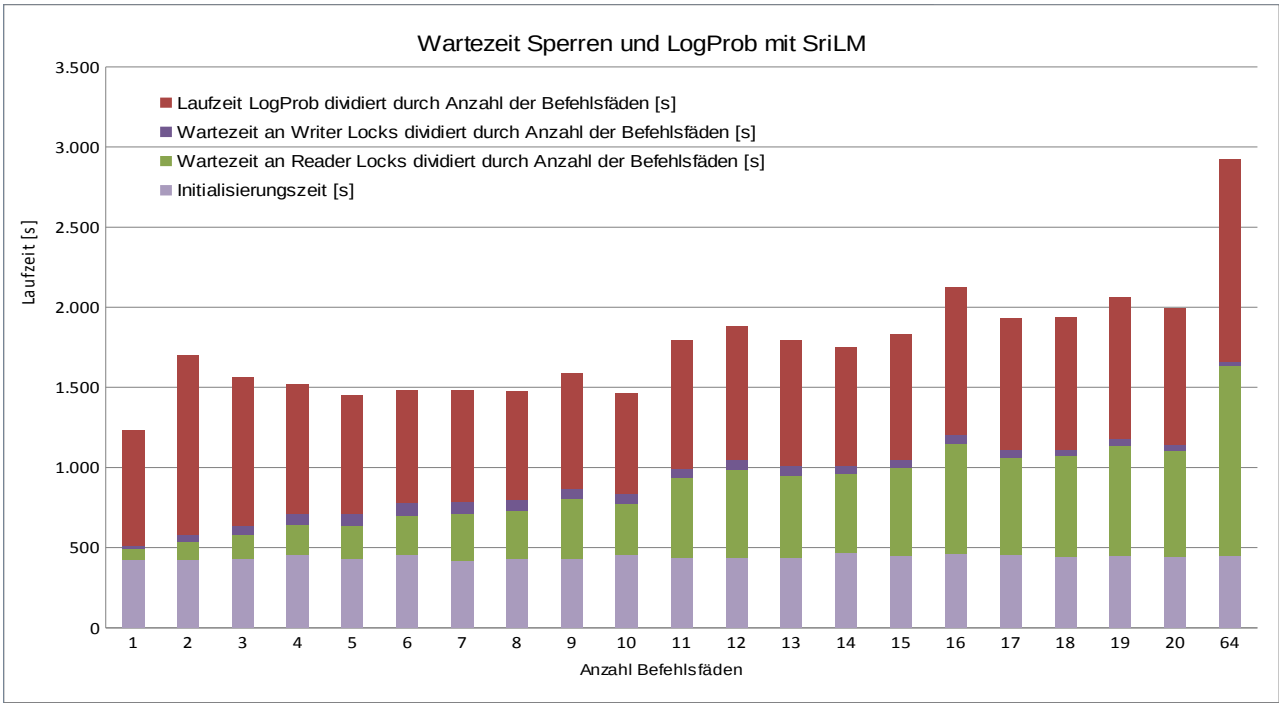


*Abbildung 11: Wartezeit Sperren und LogProb Sri LM (relativ)*

Betrachtet man die, in Relation zur Anzahl der beteiligten Kerne gesetzten, Wartezeiten an lesenden und schreibenden Sperren, fällt auf, dass die Wartezeit pro Kern bei schreibenden Zugriffen weitgehend stagniert, während die Wartezeit an Leser-Sperren stetig zunimmt.

Die Hauptfunktion des Sprachmodells `LogProb()` behält unabhängig von der Anzahl der beteiligten Prozessoren ihre Ausführungszeit konstant bei, da sie, wie zuvor erläutert, in SriLM nicht parallelisierbar („threadsafe“) ist und deshalb einen kritischen Bereich markiert.

Um ihren signifikanten Anteil an der Zunahme der Gesamtverarbeitungszeit zu mindern, wurde das Sprachmodell-Toolkit ausgetauscht und im Weiteren mit KenLM gearbeitet, da dieses grundsätzlich für die parallele Ausführung geeignet („threadsafe“ programmiert) ist und deshalb keine expliziten Sperren benötigt.

### 8.2.2. Messungen mit Sprachmodell-Toolkit KenLM

Nach dem Austausch des Toolkits wurden erneut Messungen auf einem vergleichbaren System durchgeführt: Der benutzte Rechner (i13hpc28) verfügt über 64 Rechenkerne und 512 GB RAM.

Es wurden die gleichen Parameter wie in Kapitel 8.2.1 (Messung mit Sprachmodell-Toolkit SriLM) benutzt.

In der folgenden Darstellung (Abbildung 12: Übersetzungsdauer mit KenLM, Seite 63) wurde wieder auf der Abszisse die Anzahl der beteiligten Kerne und auf der Ordinate die verstrichene Zeit angetragen. Mehrfache Indices bedeuten dabei mehrfache Messungen unter vergleichbaren Bedingungen.

In Blau ist die Zeit der Übersetzung (ohne die Initialisierung) verzeichnet, in Grün die Initialisierung und in Rot die Gesamtzeit, der an den explizit in den Code eingebrachten Writer-Locks zugebracht wurde. Die Reader-Locks wurden vernachlässigt, da an ihnen keine nennenswerte Wartezeiten mehr auftraten.

Die Funktion `LogProb()` ist innerhalb des Toolkits nicht mehr kritisch, weshalb hierfür kein Messwert mehr existiert. Darüber hinaus stellt KenLM keine Messwerte von eigenen Sperren zur Verfügung.

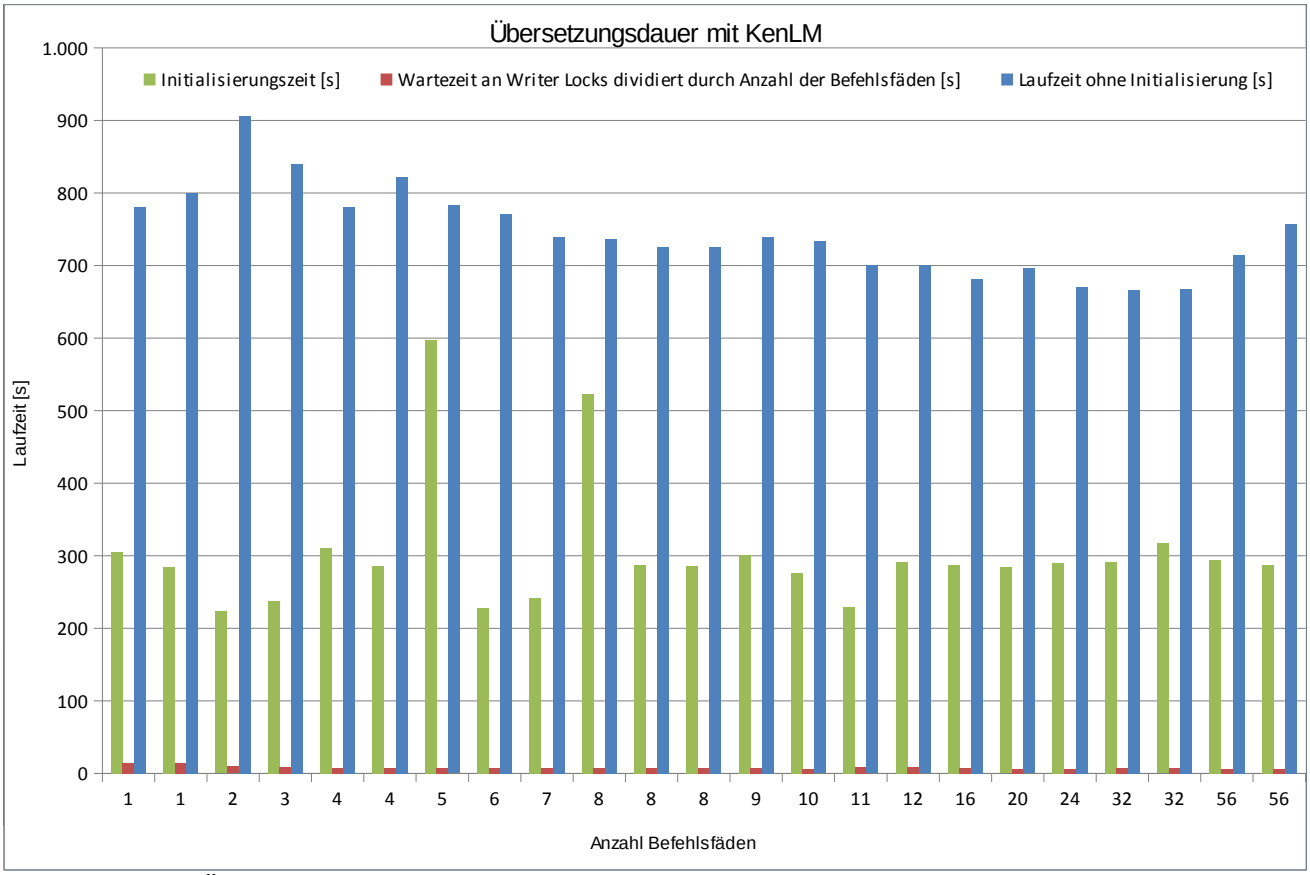


*Abbildung 12: Übersetzungsdauer mit KenLM*

Zuerst fällt der generelle Geschwindigkeitsgewinn auf, den der Wechsel des Toolkits mit sich bringt. Die einfädige Bearbeitung der gleichen Übersetzung benötigt hier nur noch etwas weniger als 800 Sekunden gegenüber 1500 Sekunden mit SriLM.

Die Initialisierungszeit (grün) schwankt leicht, bis auf zwei Ausreißer. Da die zu ladenden Daten (Sprachmodell, Phrasentabelle) umfangreich sind und nicht auf der gleichen Maschine residieren, auf der das Programm ausgeführt wird, können diese Ausreißer von Lastspitzen des verbindenden Netzwerks herrühren. Die weiteren Messwerte sind um die Initialisierungszeit bereinigt.

Die Sperren auf globale Listen (rot) bleiben pro Faden durchgehend konstant.

Es ist zu beobachten, dass die Gesamtausführungszeit auch hier bei der Zuschaltung der Mehrfädigkeit leicht ansteigt und dann tendenziell fällt. Ihr Minimum erreicht sie bei ca. 32 Kernen, danach übersteigt die Initialisierungszeit der Fäden und deren Handhabung den erzielten Geschwindigkeitszuwachs.

Weitere stichprobenartige Messungen mit deaktivierter interner Zeitmessung (mit 1, 2, 16 und 32 Kernen) ergaben vergleichbare Geschwindigkeitszuwächse von 5-10% gegenüber der einfädigen Ausführung.

Da davon auszugehen ist, dass die Implementierung von KenLM mit internen Sperren arbeitet, die das Modell erst zur mehrfädigen Ausführung befähigen, liegt der Geschwindigkeitsgewinn in einem überschaubaren Rahmen. Dieser wird vom Aufwand, den die OpenMP-Implementierung verursacht um die zusätzlichen Befehlsfäden zu initialisieren und zu starten, weitgehend kompensiert.

Eine theoretische Erklärung, die auf dem Aufbau der verwendeten Algorithmen gründet, lässt sich aus dem Amdahlschen Gesetz herleiten.

## 8.3. Vergleich der Ausführungszeit mit Amdahlschem Gesetz

Als Erklärung für die Entwicklung der Ausführungszeit in Abhängigkeit der beauftragten Prozessoren werden die bei der Analyse in Kapitel Fehler: Referenz nicht gefunden (Fehler: Referenz nicht gefunden) ermittelten Werte mit dem Amdahlschen Gesetz dargestellt und eine Korrelation zum gemessenen Laufzeitverhalten hergestellt.

Die Amdahlsche Formel für den Geschwindigkeitsgewinn (Speedup, $S$) berücksichtigt in ihrer ursprünglichen Form weder einen Zuschlag für den Aufwand, den die grundsätzliche Integration der Mehrfädigkeit mit sich bringt, noch die Initialisierungszeit für einen weiteren Befehlsfaden. Für den in der dynamischen Analyse ermittelten parallelisierbaren Anteil (42,2%) des KIT-Decoders ergäbe dies Ausführungszeiten, die mit Zunahme der beteiligten Prozessoren stetig sinkt und gegen den Anteil der sequenziellen Ausführung konvergiert (0,578). Abbildung 13 (Ausführungsdauer nach Amdahl, Seite 65) zeigt die entsprechende Kurve.

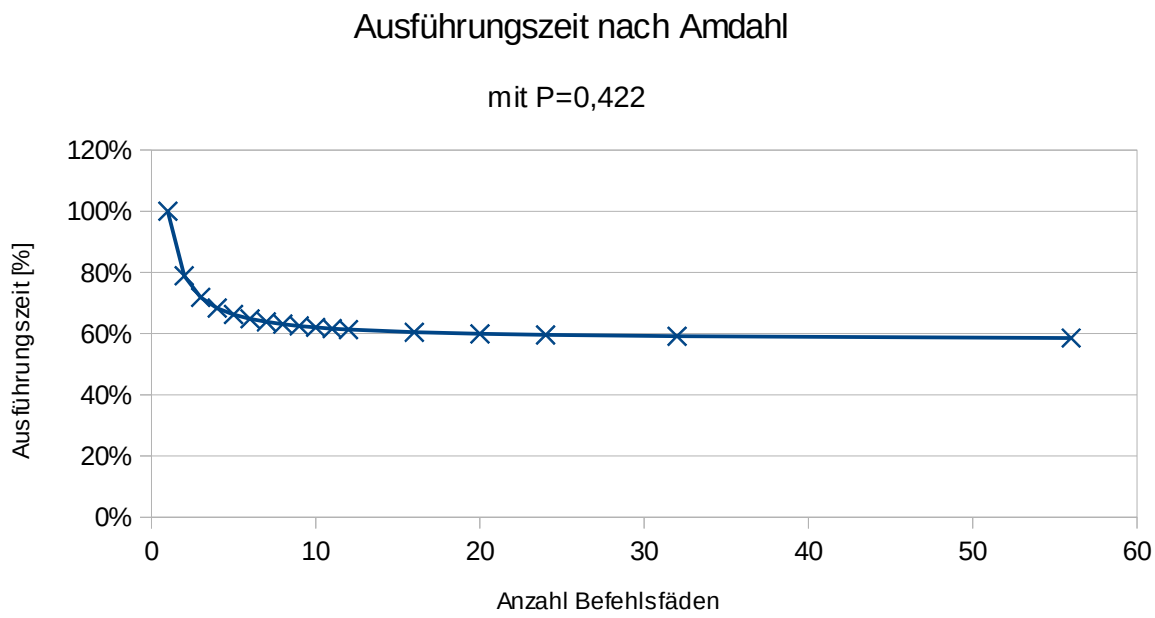


*Abbildung 13: Ausführungsdauer nach Amdahl*

Für das folgende Schaubild 14 (Gemessene Ausführungsdauer ohne Initialisierung) wurde die gemessene Ausführungsdauer um die Initialisierungszeit bereinigt und normalisiert auf 100% der Ausführungsdauer für einen Befehlsfaden.

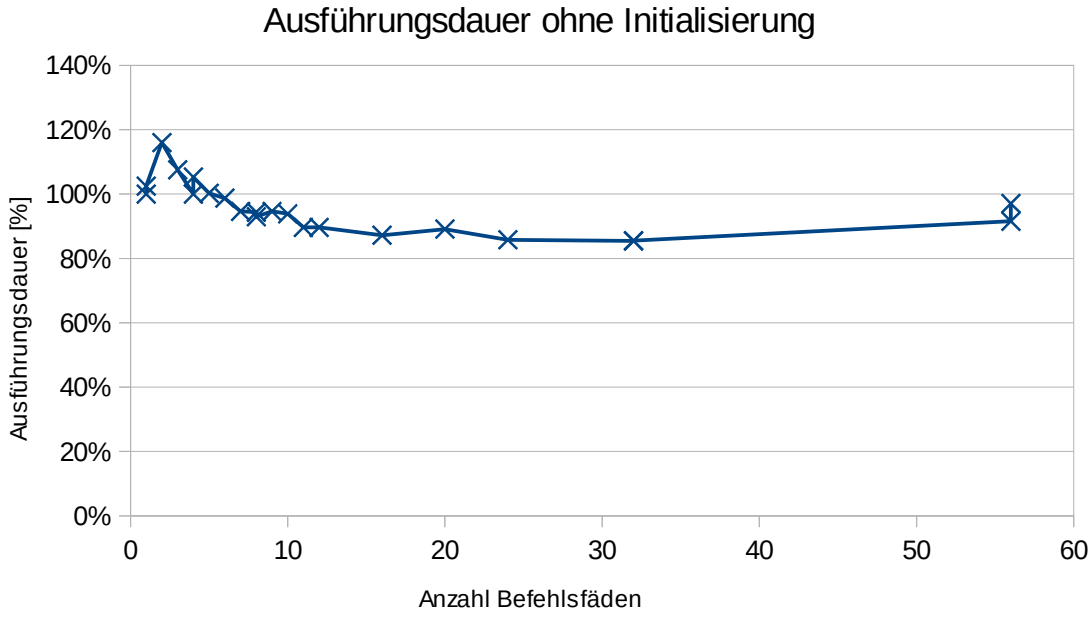


*Abbildung 14: Gemessene Ausführungsdauer ohne Initialisierung*

Stellt man die tatsächlich gemessene Ausführungsdauer der Übersetzung der nach Amdahl berechneten Ausführungsdauer gegenüber, fällt eine deutliche Differenz auf (gelbe Kurve, Abbildung 15).

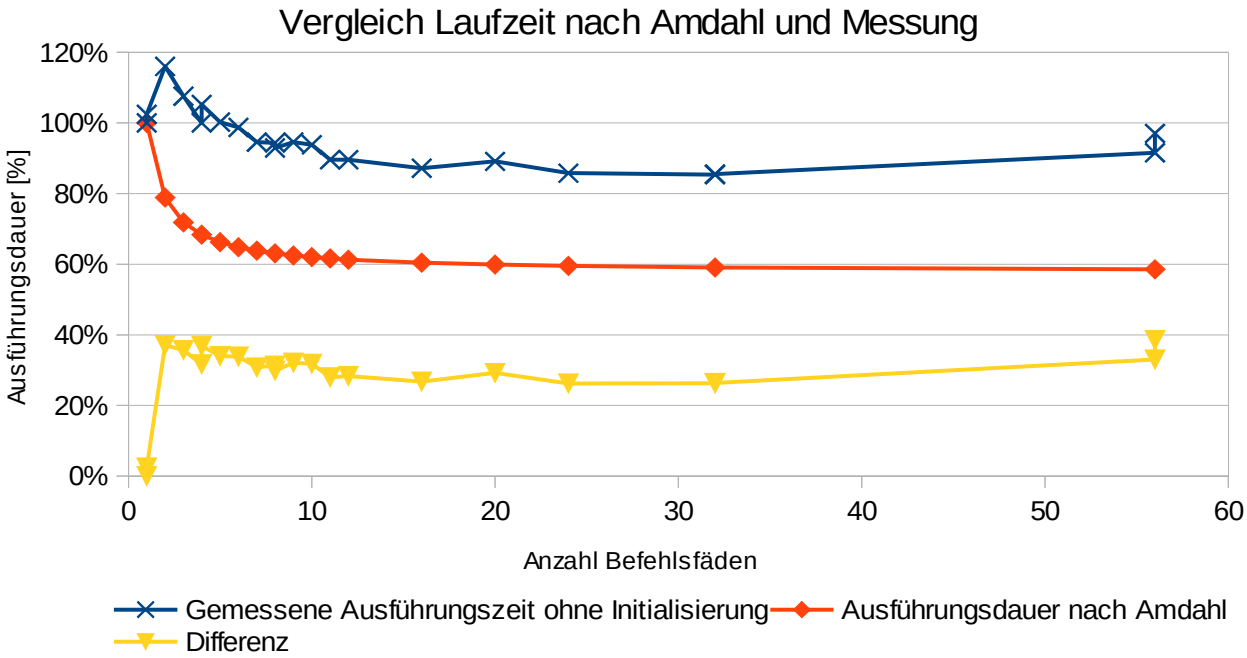


*Abbildung 15: Vergleich der Laufzeit nach Amdahl mit Messung*

*Abbildung 16* auf Seite 67 stellt nur die Differenz dar. Da die Amdahlsche Formel keinen Zuschlag für den Aufwand der Parallelisierung vorsieht, kann die gezeigte Differenz als Mehraufwand o(N) für die Parallelisierung angesehen werden.

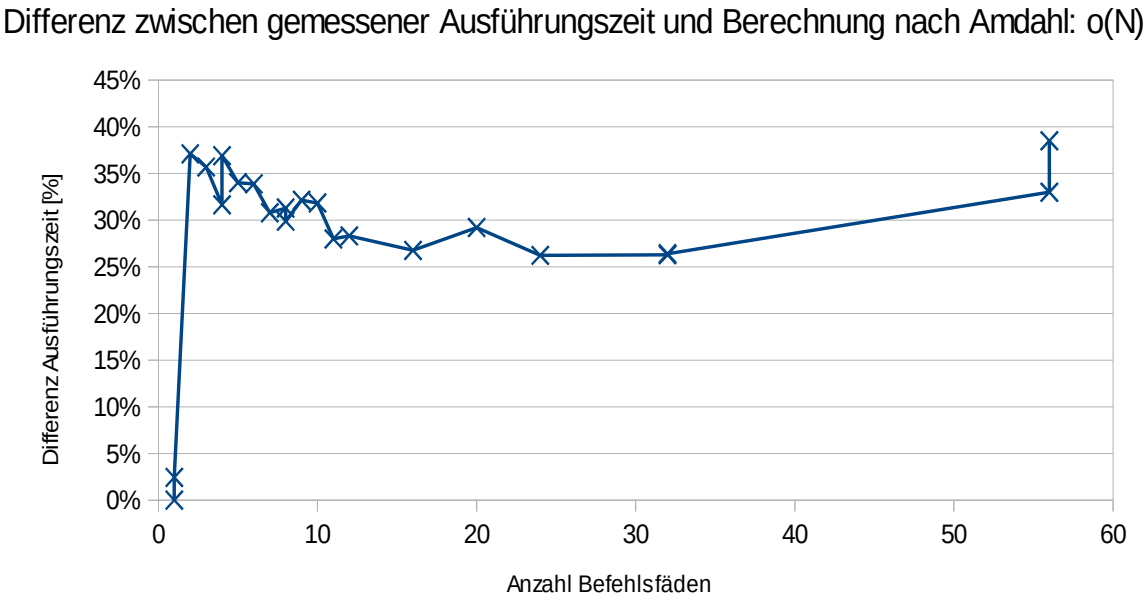


*Abbildung 16: Differenz zwischen gemessener Ausführungszeit und Berechnung nach Amdahl: o(N)*

Im nächsten Schritt wird versucht die zugehörige Aufwandsfunktion zu ermitteln. In einem Ansatz der ETH-Zürich [GP2011] wurde dies mit mehreren additiven Komponenten bewerkstelligt, die aus einem Funktions-Aufschlag in Abhängigkeit der Anzahl der zu initialisierenden Befehlsfäden sowie einem Aufschlag für die Wartezeiten an Synchronisationsbarrieren bestehen.

Die entsprechende Erweiterung des Amdahlschen Gesetztes lautet mit den angegebenen Variablen wie folgt:

*N:* Anzahl der Befehlsfäden

$S_{(N)}$*:* Speedup in Abh.v. *N*

$T_{(1)}$*:* Ausführungszeit bei sequenzieller Ausführung

*o(N):* Verwaltungsaufwand der Parallelität

*P:* parallelisierbarer Anteil

*1-P:* sequenzieller Anteil

$$S_{(N)} = \frac{T_{(1)}}{T_{(1-P)} + T_{(o(N))} + \frac{T_{(P)}}{N}}$$

Wird sie nun so umformuliert, dass sie die Ausführungszeit einer partiell parallelen Anwendung darstellt, erhält man die folgende Darstellung:

*N*: Anzahl der Befehlsfäden

$T_{(N)}$: Ausführungszeit in Abhängigkeit der Befehlsfäden

*o(N)*: Verwaltungsaufwand der Parallelität

*P*: parallelisierbarer Anteil

1-*P*: sequenzieller Anteil

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

Der in Abbildung 16 auf Seite 67 dargestellte Laufzeitunterschied zeigt einen sprunghaften Anstieg beim Wechsel von einem zu zwei Befehlsfäden. Aus diesem Grund sind die Ergebnisse der sequenziellen (N=1) und der parallelen Ausführung (N>1) getrennt durch eine Fallunterscheidung zu betrachten, da die Parallelverarbeitung bei OpenMP erst ab zwei Befehlsfäden zugeschaltet wird.

Die Laufzeit der parallelen Ausführung nimmt dabei bis zu einem bestimmten Punkt stetig ab, um anschließend wieder anzusteigen.

Um eine Näherungsfunktion mit entsprechender Tendenz zu ermitteln, wurde eine polynomische Regressionsanalyse durchgeführt, die in Abbildung 17 dargestellt wird.

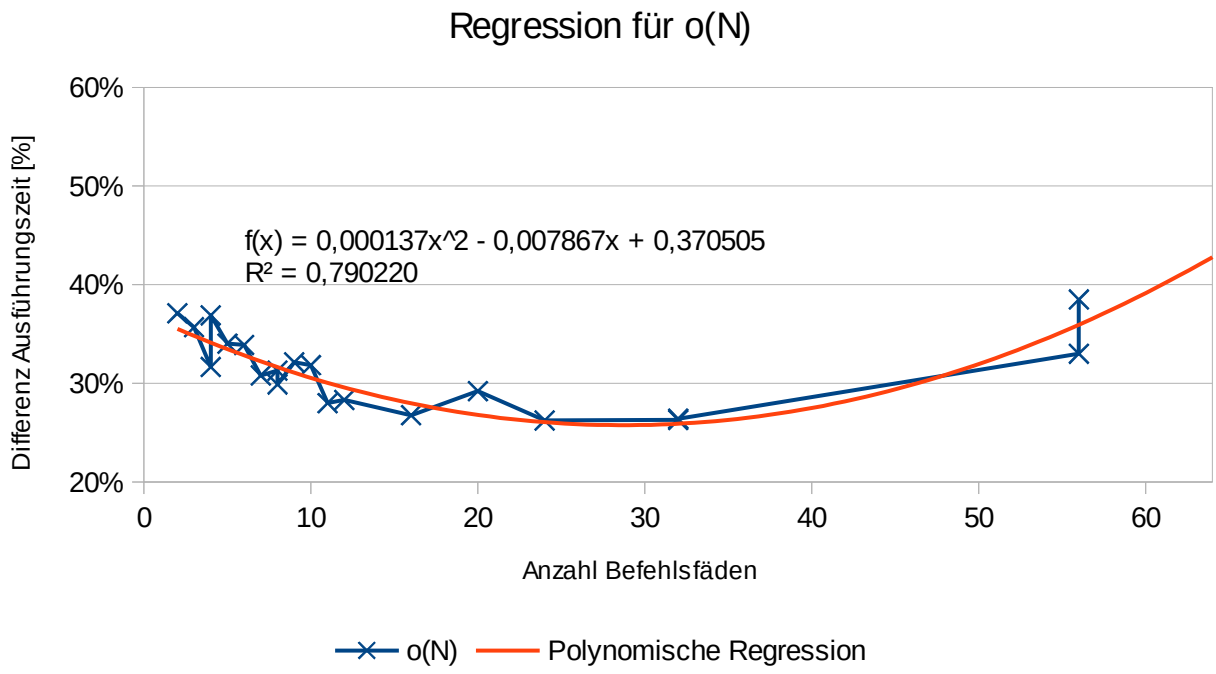


*Abbildung 17: Polynomische Regression für o(N)*

Als Näherungsfunktion erscheint $o(N) = aN^2+bN+c$ geeignet, wobei für *a, b* und *c* die Werte *0,000137, -0,007867* und *0,370505* errechnet wurden. Das Bestimmtheitsmaß $R^2 \in [0,1]$ deutet dabei eine gute Korrelation (ca. 0,8) an.

Der Kurvenverlauf legt eine polynomielle Zunahme des Aufwands für die Parallelisierung nahe, der den gegen *S = 1 – P = 0,578* (in Abbildung 13 Seite 65 dargestellten) konvergierenden Verlauf der Amdahlschen Vorhersage überkompensiert.

Unter der Annahme, dass der durch die Parallelisierung verursachte Aufwand *o(N)* abhängig ist von der Anzahl der beteiligten Prozessoren und wie in Abbildung 17 skizziert ansteigt, ergibt sich folgende Definition für *o(N)*:

$$o(N) = \begin{cases} 0 & (\textit{für } N = 1) \\ a N^2 + bN + c & (\text{ sonst }) \end{cases}$$

mit

$$a = 0{,}000137$$

$$b = -0{,}007867$$

$$c = 0{,}370505$$

# 9. Zusammenfassung und Ausblick

Für die Wahl der Parallelisierungstechnik wurden verschiedene Konzepte betrachtet: MPI, GPGPU, OpenMP und Cluster OpenMP. MPI erwies sich wegen der verteilten Speicherarchitektur und dem hohen Kommunikationsaufwand als ungeeignet, GPGPU schied wegen seines beschränkten Befehlsumfangs, vor allem hinsichtlich Verzweigungen, aus. Cluster OpenMP wurde nicht fertiggestellt und wird nicht mehr weiterentwickelt. Aufgrund der Speicherarchitektur und des moderaten Eingriffs in den bestehenden Programm-Code wurde OpenMP gewählt.

Die vorhandene Code-Basis wurde mit Messpunkten instrumentiert um günstige Ansatzpunkte für die Parallelisierung und die beste Granularität zu identifizieren. Dabei wurden drei mögliche Punkte genauer untersucht: die satzweise Parallelisierung, die Hypothesenexpansion und die Anwendung der genutzten Modelle mit der Speicherung der Hypothese. Hier erwies sich die Hypothesenexpansion als am meisten Erfolg versprechend. Entsprechend wurden die relevanten Programmteile für die Mehrfädigkeit erweitert oder geändert.

Bei der mehrfädigen Implementierung mit dem SRI-Sprachmodell stellte sich heraus, dass dieses nicht uneingeschränkt für die parallele Ausführung geeignet („threadsafe“) ist und deshalb einige kritische Stellen aus der parallelen Ausführung ausgenommen werden mussten. Deshalb wurde das Sprachmodell geändert und Ken-LM eingeführt, welches die o.a. Einschränkungen nicht aufweist und generell eine wesentlich schnellere Verarbeitung erlaubt.

Nach erfolgreichen Testläufen im mehrfädigen Modus wurden etliche Messläufe mit und ohne Instrumentierung ausgeführt, um das Laufzeitverhalten des Codes in Abhängigkeit der Prozessanzahl zu analysieren.

Die gemessenen Ergebnisse wurden den berechneten Werten aus der Amdahlschen Formel zur Ermittlung des möglichen Geschwindigkeitsgewinns gegenüber gestellt. Aus der Differenz dieser Kurven wurde der Aufwand der Parallelisierung berechnet und durch eine polynomische Regression eine Näherungsfunktion ermittelt, die die quadratische Zunahme der Ausführungszeit für mehr als 32 Befehlsfäden nahe legt.

Im Bereich von 16- bis 32-fädiger Ausführung sind mit dem vorliegenden Grad an Parallelität Geschwindigkeitsvorteile von ca. 16% realisierbar.

In einem künftigen Ansatz könnten neue Grundlagentechniken wie Transactional Memory weitere Zuwächse in der Geschwindigkeit ermöglichen. Der Erfolg versprechendste Ansatz dürfte aber sein, die grundlegende Architektur der Algorithmen zu ändern, um einen höheren Grad an Parallelität zu ermöglichen, der wiederum die Ausführungszeit relevant verkürzen könnte.

# A. Quellenverzeichnis


[A1967]: Gene Amdahl, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, 1967

**[**A1979]: Douglas Adams, Per Anhalter durch die Galaxis (The Hitchhiker's Guide to the Galaxy), 1979

**[**BU2002]: Theo Ungerer, Uwe Brinkschulte, Mikrocontroller und Mikroprozessoren, 2002

**[**COMP2005]: Jay P. Hoeflinger, Extending OpenMP*to Clusters, 2005

**[**D.etAl2010]: C. Dyer, A. Lopez, J. Ganitkevitch, J. Weese, F. Ture, P. Blunsom, H. Setiawan, V. Eidelman, and P. Resnik, cdec: A Decoder, Alignment, and Learning Framework for Finite-State and Context-Free Translation Models, 2010

**[**G1988]: John L. Gustafson, Reevaluating Amdahl's law, 1988

**[**G2012]: Jonas Gehring, Training Deep Neural Networks forBottleneck Feature Extraction, 2012

[GP2011]: Thomas Gross & Markus Pueschel, Design of Parallel and High-Performance Computing, Performance models: Amdahl's Law, 2011

**[**GPGPU2002]: Thompson et al., Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, 2002

[H2013]: Jochen Huck, Generierung parallelerArchitekturbeschreibungen

durchkombinierte statische und dynamischeAnalysen, 2013

[KetAl2007]: Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, Evan Herbst, Moses: Open Source Toolkit for Statistical Machine Translation, 2007

[L.etAl2009]: Zhifei Li, Chris Callison-Burch, Chris Dyer,Juri Ganitkevitch, Sanjeev Khudanpur, Lane Schwartz, Wren N. G. Thornton, Jonathan Weese and Omar F. Zaidan, Joshua: An Open Source Toolkit for Parsing-based Machine Translation, 2009

[LK2008]: Zhifei Li and Sanjeev Khudanpur, Association for Computational Linguistics A Scalable Decoder for Parsing-based Machine Translationwith Equivalent Language Model State Maintenance, 2008

[MPI1994]: Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, 1994

[OCL2009]: OpenCL Reference Pages, 2009, https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/

[OMPI2004]: Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall , Title: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, 2004

[RS2001]: Ren, F. ; Hongchi Shi, Parallel machine translation: principles and practice, 2001

[S1997]: W. T. Sullivan, III (U. Washington), D. Werthimer, S. Bowyer, J. Cobb (U.California, Berkeley), D. Gedye, D. Anderson (Big Science, Inc.), A new major SETI project based on Project Serendip data and 100,000 personal computers , 1997

**[**S2005]: Herb Sutter, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, 2005

**[**VHKW2004]: Stephan Vogel, Sanjika Hewavitharana+, Muntsin Kolss, Alex Waibel+, The ISL Statistical Translation Systemfor Spoken Language Translation, 2004

[VSHN2010]: D. Vilar, D. Stein, M. Huck, and H. Ney, Jane: Open Source Hierarchical Translation, Extended with Reordering and Lexicon Models, 2010

[WM2015]: WinMerge ist ein Open-Source-Programm zum Unterscheiden und Zusammenführen von Dateien sowie Ordner, , http://winmerge.org/

# B. Glossar

**Befehlsfaden (thread):** Aktivitätsträger in der Abarbeitung eines (Teil-) Programmes. Er kommt in sequenziellen Programmen genau ein Mal vor und kann bei paralleler Verarbeitung potenziell beliebig häufig gleichzeitig auf mehreren Prozessoren ablaufen.

**cluster:** Verbund von Rechnern, die hinsichtlich einer bestimmten Eigenschaft gleich oder ähnlich sind. Cluster-Rechner sind i.d.R. vollständige Rechnersysteme, die Teilaufgaben der Berechnung umfangreicher Probleme lösen.

**Datenflussabhängigkeit (data dependency):** Eine Datenflussabhängigkeit ist eine Situation, in der die Daten einer Anweisung auch von einer anderen Anweisung benutzt werden. Dadurch können Konkurrenzen und Konflikte entstehen.

**decoder (SMT):** Der Dekodierer übersetzt bei der statistischen Maschinenübersetzung mit Hilfe verschiedener Modelle einen Quelltext in eine Zielsprache.

**Destruktor:** Ein Destruktor ist eine Prozedur, die in der Programmierung Objekte und Variablen auflöst und den für sie verwendeten Speicher wieder frei gibt.

**framework:** Ein Rahmengerüst, das in der Software-Entwicklung bestimmte Entwicklungsansätze zur Verfügung stellt.

**GPGPU (General Purpose Computation on Graphics Processing Units):** GPGPU bezeichnet die Verwendung von Grafikprozessoren für die Berechnung allgemeiner Aufgaben.

**Granularität:** Ein Maß für die Größe (Körnigkeit) einzelner Segmente eines Systems.

**Hypothese (SMT):** In der statistischen Maschinenübersetzung bedeutet eine Hypothese eine mögliche Übersetzung einer Phrase eines Quellsatzes.

**Konstruktor:** Ein Konstruktor ist eine Prozedur, die in der Programmierung Speicher reserviert und Objekte und Variablen sowie deren Struktur erzeugt.

**Kontrollflussabhängigkeit (control dependency):** Eine Kontrollflussabhängigkeit ist eine Situation, in der eine Anweisung von der Aktion einer vorhergehenden abhängig ist. Dadurch können bei paralleler Ausführung Konflikte entstehen.

**Lattice (SMT):** In der statistischen Maschinenübersetzung bezeichnet der Lattice die Datenstruktur, die alle möglichen Übersetzungen eines Quellsatzes umfasst.

**Metrik:** Eine Metrik ist eine Funktion, die Elemente einer Menge hinsichtlich einer bestimmten Eigenschaft mit einer Maßzahl versieht und die Elemente somit sortierbar macht.

**MPI:** Das Message Passing Interface ist ein Protokoll, das den Nachrichtenaustausch zwischen verschiedenen Rechnersystemen bei parallelen Berechnungen definiert.

**Mutex:** Ein Mutex ist ein Verfahren, das den kontrollierten Zugriff auf eine Ressource (wechselseitiger Ausschluss) bei paralleler Befehlsverarbeitung regelt.

**Nebenläufigkeit (threadsafeness):** Die Fähigkeit zur Nebenläufigkeit ermöglicht es, mehreren Befehlsfäden gleichzeitig, störungsfrei parallel abgearbeitet zu werden. Diese Eigenschaft muss bereits bei der Entwicklung berücksichtigt werden.

**Objektreferenz:** In der Programmierung ein Verweis auf ein Objekt ohne dessen Inhalt.

**OpenMP (Open Multi-Processing):** Eine herstellerübergreifende Programmierschnittstelle für die parallele Programmierung auf Shared-Memory-Rechnern.

**paralleler Korpus (SMT):** Ein paralleler Korpus ist eine Liste von Phrasenpaaren mit Zuordnungen von der Quell- zur Zielsprache.

**Phrase (SMT):** Eine Phrase bezeichnet in der statistischen Maschinenübersetzung die syntaktisch korrekte Anordnung von einem oder mehreren Wörtern.

**Phrasenanzahlmodell (Phrase Count Model, PC):** Das PC ist ein Modell der statistischen Maschinenübersetzung, das mögliche Übersetzungen hinsichtlich der Anzahl der verwendeten Phrasen bewertet.

**Phrasenpaar (SMT):** Ein Phrasenpaar stellt die Übersetzung eines oder mehrerer Wörter einer Quellsprache in der jeweiligen Zielsprache dar.

**Quellsatz (SMT):** Ein vollständiger zu übersetzender Satz aus der Quellsprache.

**Semaphor:** Ein Semaphor ist eine Datenstruktur die den wechselseitigen Ausschluss zur Vermeidung von Konflikten als „Reservieren“ / „Freigeben“ für aufrufende Programme selbstständig bereit stellt.

**Shader-Prozessor:** Ein Shader-Prozessor ist eine Verarbeitungseinheit eines Grafiksystems, das für die Verarbeitung bestimmter elementarer Rechenschritte optimiert ist und für die parallele Bearbeitung meist mehrfach zur Verfügung steht.

**Simultanübersetzung**: Eine Form der Übersetzung, die die Übersetzung eines Quellsatzes nahezu gleichzeitig produziert und ausgibt.

**SMT (Statistical Machine Translation):** In der statistischen Maschinenübersetzung werden Übersetzungen auf Basis statistischer Modelle generiert.

**Speichermodell (SMM, DMM, DSM):** In der Parallelverarbeitung werden grundsätzlich drei Speichermodelle unterschieden: Shared Memory (alle Systeme greifen auf den gleichen Speicher zu), Distributed Memory (jedes System hat eigenen Speicher, Zugriff erfolgt nur über explizite Kommunikation der Prozesse) und Distributed Shared Memory (eine Kommunikationsschicht erlaubt den Zugriff auf lokalen Speicher über eine darübergelegte globale, virtuelle Adressierung).

**Sprachmodell (Language Model, LM):** Das LM ist ein Modell der statistischen Maschinenübersetzung, das mögliche Übersetzungen anhand ihrer Ähnlichkeit zu vorliegenden Phrasen aus der Zielsprache bewertet.

**Übersetzungsmodell (Translation Model, TM):** Das TM ist ein Modell der statistischen Maschinenübersetzung, das mögliche Übersetzungen anhand vorliegender Phrasenpaare aus dem parallelen Korpus bewertet.

**Umordnungsmodell (Distortion Model, DM):** Das DM ist ein Modell der statistischen Maschinenübersetzung, das mögliche Übersetzungen anhand bekannter Umordnungen von Worten aus der Quell- in die Zielsprache bewertet.

**Verklemmung (deadlock):** Eine Verklemmung ist eine Wartesituation zwischen mindestens zwei parallelen Prozessen, bei der jeweils ein Prozess auf die Freigabe einer Ressource wartet, selbst jedoch eine Ressource belegt, die wiederum von dem oder den anderen Prozess benötigt wird.

**Wortanzahlmodell (Word Count Model, WC):** Das WC ist ein Modell der statistischen Maschinenübersetzung, das mögliche Übersetzungen anhand der Anzahl der übersetzen Worte der Zielsprache bewertet.

**Zielsatz:** Ein vollständig übersetzter Satz der Zielsprache.

# C. Details der Implementierung

Um die erarbeiteten Änderungen des Codes auch in andere Entwicklungsäste des Decoders einbringen zu können, werden hier die relevanten Änderungen dokumentiert:

## *I. Hinzufügen des High-Precision-Timers zum Makefile:*

Die Datei `UpdateMakefiles.sh` wurde in Zeile 83 `"set ld_flags..."` um den Zusatz `"-lrt"` erweitert.

Das Testset ist `test.btec.en.Final.sc` ; es besteht aus einer Menge von 2000 zu übersetzenden Sätzen, wobei die Quellsprache Deutsch und die Zielsprache Englisch ist. Als Übersetzungstabelle wird `phrase-table.0-0.grow-diag-final-and.pruned.sttk` benutzt, das Sprachmodell ist `train.de-en.1-99.en.4gram.no-cutoff34.srilm`.

Zeile 13 (Translate09_SOURCES = Translate09.cc )

muss um "multithreading.cc" erweitert werden

### II. Includes

Um OpenMP zu nutzen, muss die zugehörige include-Datei `omp.h` eingebunden werden; zur Steuerung der Multithreading-Funktionen wurde die Klasse `multithreading.hh` eingeführt:

Translate09.cc Zeile 154:

```
...
//cw:
#include <omp.h>
#include <multithreading.hh>
//cw: end
...
```

Das Makefile-Erzeuger-Script wurde für OpenMP angepasst:

Die Datei `UpdateMakefiles.sh` wurde in Zeile 84 `"set cxx_flags=..."` um den Zusatz `"-fopenmp"` erweitert.

### III. Prüfausgabe in Translate09.cc

Um sicher zu gehen, dass alle Threads korrekt initialisiert werden, wurde zu Beginn des Hauptprogramms eine Testausgabe implementiert:

translate09.cc Zeile 356:

```
...
#pragma omp parallel
{
     printf("cw2: I  am  thread  %d  of  %d.\n",
omp_get_thread_num()+1,  omp_get_num_threads());
...
} //Ende der parallelen Region
...
```

### IV. Implementierung der Zeitmessung

multithreading.hh Zeile 17:

```
...
typedef long long cwtime;
extern cwtime cw_LSLPT_s, cw_LSLPT_ns;
...
extern cwtime cw_TSTL_s, cw_TSTL_ns;
...
```

DecoderBLM():

multithreading.hh Zeile 34:

```
...
//DecoderBLM.cc
extern cwtime cw_ALGLMWI_s, cw_ALGLMWI_ns;
...
extern cwtime cw_ALTL_s, cw_ALTL_ns;
...
```

ExpandHypOverEdge():

multithreading.hh Zeile 45:

```
...
//DecoderBLM.cc:ExpandAllHyps...
extern long EHOE_TotalLoop;
...
extern long EHOE_StoreHypothesis_ns;
...
extern cwtime cw_EHOEI_s, cw_EHOEI_ns;
...
extern cwtime cw_EHOETL_s, cw_EHOETL_ns;
...
```

`StoreHypothesis()`:

multithreading.hh Zeile 45:

```
...
extern long SH_Total_s;
extern long SH_Total_ns;
extern long SH_Init_s;
extern long SH_Init_ns;
extern long SH_IFHFP_s;
extern long SH_IFHFP_ns;
...
extern cwtime cw_SHT_s, cw_SHT_ns;
extern cwtime cw_SHInit_s, cw_SHInit_ns;
extern cwtime cw_SHIFHFP_s, cw_SHIFHFP_ns;
...
```