technische universität
dortmund

Bachelorarbeit

Parallelisierung einer speichereffizienten Approximation der LZ77-Faktorisierung

Gajann Sivarajah

Gutachter:
Prof. Dr. Johannes Fischer
M.Sc. Patrick Dinklage

Technische Universität Dortmund
Fakultät für Informatik
LS-11
http://afe.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Die Entwicklung, Verbreitung und Nutzung digitaler Technologien hängt im hohen Maße von der Fähigkeit ab, große Mengen an Daten speichern, transportieren und analysieren zu können. Der Umgang mit großen Datenmengen geht jedoch mit entsprechend hohen Kosten einher. Ein wichtiges Werkzeug zur Bewältigung dieses Problems sind Kompressionstechniken, die Relationen und Redundanzen in Datenmengen extrahieren, um ihre Größe möglichst auf ihre inhärente Komplexität zu reduzieren. Im Laufe der Zeit wurden zahlreiche Kompressionsalgorithmen entwickelt, die wiederum über mehrere Iterationen verbessert wurden. Viele solcher Kompressionstechniken können der Familie der LZ77-Algorithmen [19] zugeordnet werden, wobei diese sich in Statistiken, wie der Laufzeit, der Speicheranforderung oder Kompressionrate unterscheiden. In [5] wird eine Variante der LZ77-Faktorisierung beschrieben, die über drei Phasen eine 2-Approximation einer exakten LZ77-Faktorisierung [14] erreichen kann. Diese beschränkten Einbußen in der Qualität der Ausgabe werden jedoch dadurch kompensiert, dass der Algorithmus die Speicheranforderung weit unterbieten kann. In dieser Arbeit untersuchen wir diesen Algorithmus auf ihr Potenzial zur Verbesserung der Laufzeit durch eine Parallelisierung.

1.2 Ziele und Methodik

Im Rahmen der Parallelisierung des approximativen LZ77-Algorithmus werden wir die erste Phase des Algorithmus dahingehend anpassen, dass mehrere Threads im shared-memory-Modell [8]) konfliktfrei auf Datenstrukturen zugreifen und eine korrekte Ausgabe liefern können. Für die praktische Evaluation der beschriebenen Konzepte wird eine Implementierung in C++ herangezogen. Die Parallelisierung wird hauptsächlich über OpenMP-Instruktionen [1] realisiert. Im Rahmen dieser Arbeit wird insbesondere die parallele Generierung einer Suchtabelle für Referenzen, sowie die parallele Suche nach Referenzen über

die gesamte Eingabe hinweg betrachtet. Wir führen eine theoretische und praktische Evaluation der Qualität und Performanz der Algorithmen durch. Insbesondere stellen wir einen Vergleich der Laufzeit und Speicheranforderung der sequentiellen und parallelen Approximation mit einer exakten LZ77-Faktorisierung [14] an. Die Güte der Parallelisierung werden wir anhand der gemessenen Beschleunigung der Laufzeit bewerten. Für jegliche Messungen verwenden wir Testdaten aus unterschiedlichen Kontexten des Pizza & Chili Corpus [12].

Kapitel 2

Grundlagen

Zunächst stellen wir die verwendete Terminologie und relevante Konzepte bzw. Phänomene dar.

2.1 Eingabe

Unsere Eingabe sei durch eine n-elementige Zeichenfolge $S = e_1...e_n$ über dem beschränkten numerischen Alphabet Σ mit $e_i \in \Sigma \ \forall i = 1, ..., n$ gegeben. Für jede beliebige Zeichenfolge S wird mit $|S|$ dessen Länge, hier n, bezeichnet. Der Ausdruck $S[i..j] \in \Sigma^{j-i+1}$ mit $1 \leq i \leq j \leq n$ beschreibt die Teilfolge $e_i...e_j$, wobei im Falle, dass $i = j$ ist, das einzelne Zeichen e_i referenziert wird. Alternativ kann ein einzelnes Zeichen e_i auch durch $S[i]$ referenziert werden. Eine Teilfolge der Form $S[1..k]$ mit $1 \leq k \leq n$ wird als Präfix von S bezeichnet. Im Gegensatz dazu wird eine Teilfolge der Form $S[k..n]$ als Suffix von S bezeichnet. Für zwei Teilfolgen S_1 und S_2 beschreibt der Ausdruck $S_1 \cdot S_2$ die Konkatenation der beiden Teilfolgen.

2.2 Ausgabe → Faktorisierung

Ein charakteristisches Merkmal der Familie der Lempel-Ziv-Kompressionsverfahren [19] ist die Repräsentation der Ausgabe in Form einer Faktorisierung. Für eine Eingabe $S = e_1...e_n$ wird eine Faktorisierung $F = f_1 \cdot \cdots f_z$ mit $z \leq n$ derart erzeugt, dass die Eingabe S durch die Faktorisierung in eine äquivalente Folge von nichtleeren Teilfolgen zerlegt wird. Dabei ist jeder Faktor f_i mit $1 \leq i \leq z$ als nichtleerer Präfix von $S[|f_1 \cdot \cdots f_{i-1}| + 1..n]$ definiert, der bereits in $S[1..|f_1 \cdot \cdots f_i|]$ vorkommt, oder als einzelnes referenzloses Zeichen. Die im Folgenden betrachteten Algorithmen können speziell der Klasse der LZ77-Kompressionsverfahren zugeordnet werden, dessen Faktoren im Schema des Lempel-Ziv-Storer-Szymanski(LZSS) [18] repräsentiert werden sollen.

$$F = f_1 \cdots f_z \text{ mit } f_i = \begin{cases} (\text{Länge}, \text{Position}) & \text{falls Referenz} \\ (0, \text{Zeichen}) & \text{sonst} \end{cases} \tag{2.1}$$

Zur Darstellung von Referenzen wird das Tupel aus der Position des vorherigen Vorkommens und der Länge des Faktors genutzt. Einzelne Zeichen können wiederum durch das Tupel aus dem Platzhalter, 0, und dem entsprechenden Zeichen dargestellt werden. Das in 2.1 definierte Format beschreibt die gewünschte Ausgabe der im Folgenden betrachteten Algorithmen.

2.3 Kompression

2.3.1 Verlustfreie Kompression

Der Prozess der Kompression überführt eine Repräsentation einer finiten Datenmenge in eine möglichst kompaktere Form. Eine verlustfreie Kompression ist gegeben, falls die Abbildung zwischen der ursprünglichen und komprimierten Repräsentation bijektiv ist. Die Korrektheit einer verlustfreien Kompression kann daher durch die Angabe einer Dekompressionsfunktion nachgewiesen werden. Ist diese Vorraussetzung nicht gegeben, so handelt es sich um eine verlustbehaftete Kompression, da eine Rekonstruktion der ursprünglichen Datenmenge nicht garantiert werden kann.

2.3.2 Dekompression

Die Dekompression beschreibt den Umkehrprozess der Kompression und erlaubt im Falle eine verlustfreien Kompression die Rekonstruktion der ursprünglichen Datenfolge. Im Falle von Verfahren der LZ77-Familie, kann die Dekompression durch die folgende Abbildung definiert werden,

$$DECOMP_{LZ77} : F(1..z) \rightarrow S(1..n). \tag{2.2}$$

Der dargestellte Algorithmus in 2.1 beschreibt eine mögliche Implementierung der Dekompression von einer Faktorisierung $F = f_1...f_z$ zu der originalen Eingabe $S = e_1...e_n$. Der beschriebene Algorithmus iteriert durch alle Faktoren und fügt die referenzierten Zeichen einzeln in S ein. Damit kann die Laufzeit des Algorithmus auf $O(n)$ geschätzt werden.

2.3.3 String-Matching → Rabin-Karp

Im Rahmen des approximativen Algorithmus, welcher in dieser Arbeit beschrieben wird, werden Vergleiche von Zeichenfolgen mithilfe des Rabin-Karp-Fingerprints(RFP) [9] durch-

Algorithmus 2.1 DECOMP$_{LZ77}$

Eingabe:$F = f_1...f_z$ *Ausgabe:*$S = e_1...e_n$

```
1: S ← ∅
2: for i = 1 to z do
3:    (len, ref) ← f_i
4:    if len = 0 then
5:       S ← S · ref
6:    else
7:       for j = 0 to len − 1 do
8:          S ← S · S[ref + j]
9:       end for
10:   end if
11: end for
12: return S
```

geführt. Sei $p \in \mathbb{P}$ eine Primzahl und $b \in \mathbb{N}$ eine Basis, so kann der RFP einer Zeichenfolge S der Länge n durch den Ausdruck

$$\begin{aligned} RFP(S) &= \sum_{i=1}^{n} S[i]b^{n-i} \mod p \\ &\in \{0, ..., p-1\} \end{aligned} \tag{2.3}$$

berechnet werden. Hierbei wird eine Zeichenfolge beliebiger Länge in eine Zahl aus dem Intervall $[0, p-1]$ abgebildet. Der RFP erlaubt es, die Gleichheit zweier Zeichenfolgen zu widerlegen im Falle von unterschiedlichen Werten. Im Falle von gleichen RFPs, kann die Gleichheit der Zeichenfolgen jedoch nicht garantiert werden. Die Wahrscheinlichkeit einer Kollision dieser Art bei Zeichenfolgen gleicher Größe ist jedoch beschränkt und praktisch gering. Insbesondere kann die Wahrscheinlichkeit durch die passende Wahl von p und b minimiert werden.

Rabin-Karp-Fingerprints erlauben verschiedene Operationen auf Zeichenfolgen, die im Rahmen der approximativen LZ77-Faktorisierung effizient genutzt werden können. Zum einen kann ein beschränktes Fenster $S_W = S[j..j+w]$ der Länge $w < n$ leicht verschoben werden. Sei $RFP(S_W)$ der RFP des Fensters, so kann der resultierende RFP durch die Verschiebung um ein Zeichen nach rechts durch den Ausdruck,

$$RFP(S(j+1..j+w+1)) = (RFP(S_W) - S[j]b^{w-1})b + S[j+w+1] \mod p, \tag{2.4}$$

beschrieben werden. Des Weiteren seien zwei Teilfolgen S_1 und S_2 der gleichen Länge n gegeben. Der RFP der Konkatenation, $S_1 \cdot S_2$, der beiden Teilfolgen kann durch den Ausdruck,

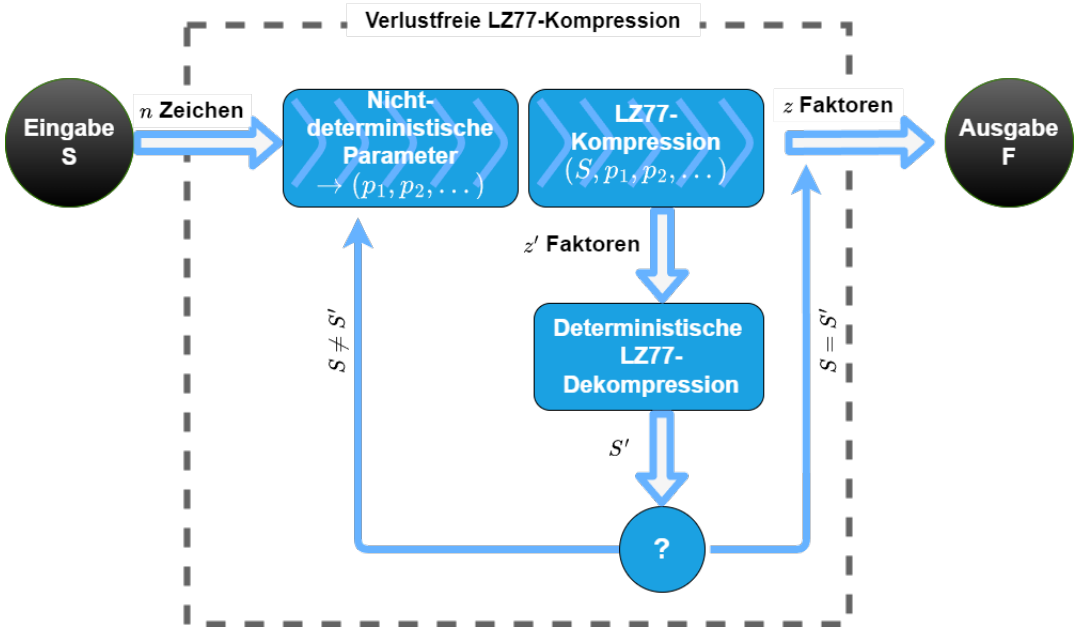
$$RFP(S_1 \cdot S_2) = (RFP(S_1)b^n + RFP(S_2)) \mod p, \tag{2.5}$$

berechnet werden. Analog zur Konkatenation von Zeichenfolgen ist die Operation 2.5 ebenfalls assoziativ, jedoch nicht kommutativ.

2.3.4 Verlustbehaftete Kompression

Im Rahmen dieser Arbeit werden wir einen Approximationsalgorithmus betrachten, der aufgrund der verwendeten RFP-Technik für Vergleiche von Zeichenfolgen eine fehlerhafte Faktorisierung mit einer beschränkten Wahrscheinlichkeit erzeugen kann. Die Korrektheit der Dekompression kann intern und extern durch explizite Vergleiche der Zeichenfolgen erkannt werden. Da der Kompressionsprozess in diesem Fall mit anderen Werten für die Parameter p und b der RFP-Berechnung wiederholt werden kann, können wir einen verlustfreien Las-Vegas-Algorithmus konstruieren.

Abbildung 2.1: Konstruktion eines verlustfreien Las-Vegas-Algorithmus durch Wiederholung des verlustbehafteten Kompressionsprozesses mit neuen Steuerparametern



In Abbildung 2.1 wird eine verallgemeinerte Regelsteuerung illustriert. Die fehleranfällige Kompression wird, solange mit neuen Parametern, hier p und b, wiederholt, bis eine korrekte Faktorisierung erzeugt wird. In der praktischen Evaluation in Kapitel 4 werden wir sehen, dass die Auftrittswahrscheinlichkeit einer fehlerhaften Faktorisierung praktisch irrelevant ist.

2.3.5 Binäre (De-)Kodierung

Die Kodierung $K_{IN} : \Sigma^* \to \{0,1\}^*$ überführt unsere Eingabe aus dem Alphabet Σ in eine binäre Repräsentation. Die Umkehrabbildung $K_{IN}^{-1} : \{0,1\}^* \to \Sigma^*$ definiert die Dekodierung und überführt eine binäre Repräsentation in eine Zeichenfolge aus dem Alphabet Σ.

Im Rahmen dieser Arbeit gehen wir davon aus, dass unsere Eingabe S über dem Alphabet $\Sigma = \{1, ..., 255\}$ erzeugt wurde und jedes Zeichen durch 8 Bits bzw. 1 Byte dargestellt wird. Für die Länge, $|S|_{Bin}$, der binären Repräsentation folgt,

$$|S|_{Bin} = |K_{IN}(S)| = 8 * |S|. \tag{2.6}$$

Die eingelesene Eingabefolge wird durch den Kompressionsalgorithmus in die Faktorfolge $F = f_1...f_z$ überführt. Die bijektive Abbildung $K_{OUT} : F \to \{0,1\}^*$ definiert die Kodierung der Faktoren in eine binäre Repräsentation. Analog dazu wird die Dekodierung $K_{OUT}^{-1} : \{0,1\}^* \to F$ definiert. Im Gegensatz zur Eingabe, werden wir keine Kodierung bzw. Dekodierung der Faktoren vorgeben, da diese durch den Kompressionsalgorithmus nicht beschränkt wird. Für eine beliebige lineare Kodierung K_{OUT} ergibt sich die binäre Ausgabegröße $|F|_{Bin}$ durch

$$|F|_{Bin} = \sum_{i=1}^{z} |K_{OUT}(f_i)|. \tag{2.7}$$

2.3.6 Metriken

Die Qualität einer Kompression kann durch verschiedene Metriken quantifiziert werden. Zum einen beschreibt die Kompressionsrate CR den Grad der Kompression und ist durch den Ausdruck,

$$CR = \frac{|F|_{Bin}}{|S|_{Bin}}, \tag{2.8}$$

definiert. Da die Kodierung der Faktoren nicht eindeutig aus der Wahl des Kompressionsalgorithmus eingegrenzt wird, ist stattdessen die Anzahl der erzeugten Faktoren ein weiteres geeignetes Gütemaß. Für die Eingabe S der Länge n und der Ausgabe $f_1...f_z$ sei die Faktorrate durch

$$FR = \frac{z}{n} \tag{2.9}$$

gegeben. In beiden Fällen wird ein niedriger Wert bevorzugt, da dieser auf eine bessere Extraktion von Redundanzen hinweist.

2.4 Parallelität

Das Ziel dieser Arbeit ist die Entwicklung und Evaluation eines parallel Kompressionsalgorithmus. Im Folgenden definieren wir die Rahmenbedingungen und Konzepte der Parallelität.

2.4.1 Shared-Memory-Modell

Unser Algorithmus agiert auf einem Shared-Memory-Modell [8] mit P Ausführungseinheiten, welches im Gegensatz zum Distributed-Memory-Modell allen beteiligten Ausführungseinheiten bzw. Prozessoren einen gemeinsamen Zugriff auf den Speicher ermöglicht.

Im Rahmen der parallelen Programmierung muss der simultane Lese- bzw. Schreibzugriff auf Speicherbereiche synchronisiert werden, um Konflikte zu vermeiden. Die Konsequenz einer mangelnden oder ineffizienten Synchronisation können Inkonsistenzen in der Korrektheit und Performanz des Algorithmus sein. In der Praxis manifestieren sich diese Probleme beispielsweise in Form von Data-Races oder False-Sharing [16]. Unser parallel modellierte Algorithmus muss explizit seine Korrektheit bewahren mit dem Ziel einer möglichst hohen Beschleunigung der Laufzeit.

2.4.2 Metriken

Das Ziel der Parallelisierung eines Algorithmus liegt hauptsächlich in einer Verbesserung der Laufzeit, insbesondere unter Berücksichtigung der bereits beschriebenen Ressourcenkonflikten. Die zeitliche Beschleunigung der Laufzeit kann durch den Speedup SP bemessen werden. Für eine Eingabe S der Länge n brauche ein sequenzieller Durchlauf $T(n, p = 1)$ Zeit, während ein paralleler Algorithmus mit P Prozessoren $T(n, p = P)$ an Zeit benötige. Der Speedup ist dabei definiert durch

$$SP(n, P) = \frac{T(n, 1)}{T(n, P)}. \tag{2.10}$$

Ein idealer Speedup ist gegeben durch $SP(n, P) = P$. Verschiedene Effekte im Rahmen des Speicherzugriffs, der Synchronisation und der Kommunikation über mehrere Prozessoren können jedoch die Effizienz der Parallelisierung stark beeinträchtigen. Insbesondere können sequenzielle Abschnitte im Algorithmus auf Basis des Amdahl'schen Gesetzes [11] eine obere Schranke für den Speedup setzen.

Kapitel 3

Kompressionsalgorithmen

3.1 (exakte) LZ77-Kompression

Der im Folgenden beschriebene Algorithmus für die Generierung einer exakten LZ77-Faktorisierung dient als Referenz für die Evaluation der approximativen Algorithmen.

3.1.1 Konzept

Wie bereits in Kapitel 2.2 beschrieben, erzeugen Algorithmen der LZ77 - Familie eine Faktorisierung einer Eingabezeichenfolge S, wobei die Faktoren entweder Referenzen zu vorherigen Zeichenfolgen oder einzelne Zeichen sein können. Im Rahmen der exakten LZ77 - Faktorisierung wird ein Greedy - Ansatz verwendet, um von links nach rechts stets die längste Zeichenfolge zu referenzieren, die bereits links von der aktuellen Position vorkommt. In 3.1 wird der Algorithmus zur Generierung einer exakten LZ77-Faktorisierung illustriert, welcher in [14] beschrieben ist. Der Algorithmus erzeugt zunächst ein Suffix-Array, welches allen Suffixen der Eingabe einen lexikografischen Rang zuweist. Anhand des lexikografischen Rangs können Kandidaten für Referenzen effizient bestimmt werden. Hierfür werden mithilfe des Suffix-Arrays zwei Arrays, das Next Smaller Value(NSV) und das Previous Smaller Value(PSV) erzeugt.

Sei S eine Eingabezeichenfolge und SA das Suffix-Array von S. Das Next Smaller Value(NSV) und das Previous Smaller Value(PSV) sind definiert als:

$$NSV[k] = SA[\min\{SA^{-1}[i] > SA^{-1}[k] | i < k\}] \tag{3.1}$$

$$PSV[k] = SA[\max\{SA^{-1}[i] < SA^{-1}[k] | i < k\}] \tag{3.2}$$

für $1 \leq k \leq n$.

Die Minimierung der lexikografischen Distanz indiziert eine Maximierung der Länge des übereinstimmenden Präfixes zwischen zwei Teilfolgen der Eingabe. Sei die aktuelle Position in der Eingabe k, so muss aufgrund von positionellen und lexikografischen Einschränkungen die Position der längsten vorherigen Referenz entweder $NSV[k]$ oder $PSV[k]$ sein, da

Algorithmus 3.1 COMP_{LZ77}: Exakte LZ77-Faktorisierung mithilfe des Suffix-Arrays

Eingabe: $S = e_1...e_n$ *Ausgabe:* $F = f_1...f_z$

1: $SA \leftarrow SuffixArray(S)$
2: $(NSV, PSV) \leftarrow (NSVArray(S, SA), PSVArray(S, SA))$
3: $F \leftarrow \emptyset$
4: $k \leftarrow 1$
5: **while** $k \leq n$ **do**
6: $(len, ref) \leftarrow (0, 0)$
7: $len \leftarrow \max\{l_{nsv} = LCP(S[NSV[k]..n], S[k..n]), l_{psv} = LCP(S[PSV[k]..n], S[k..n]\}$
8: **if** $len = 0$ **then**
9: $ref \leftarrow S[k]$
10: **else if** $l_{nsv} \geq l_{psv}$ **then**
11: $ref \leftarrow NSV[k]$
12: **else**
13: $ref \leftarrow PSV[k]$
14: **end if**
15: $F \leftarrow F.append((len, ref))$
16: $k \leftarrow k + len + 1$
17: **end while**
18: **return** F

beide in Anlehnung an 3.1 und 3.2 die minimale lexikografische Distanz aufweisen. Die maximale Länge der übereinstimmenden Präfixe zwischen $S(NSV[k]..n)$ und $S(k..n)$ bzw. $S(PSV[k]..n)$ und $S(k..n)$ wird durch die Funktion LCP berechnet. Der LCP-Wert zweier Zeichenfolgen kann durch einen Scan durch beide Zeichenfolgen berechnet werden. Das Ergebnis dieser Berechnung bestimmt sukzessive Faktoren durch die Länge und Position der Referenzen. Der Algorithmus terminiert, wenn die gesamte Eingabe abgearbeitet wurde.

3.1.2 Theoretisches Laufzeit- und Speicherverhalten

Für die Berechnung des Suffix-Arrays stehen zahlreiche effiziente Algorithmen in der Literatur zur Verfügung, wobei viele auf dem SA-IS-Algorithmus [13] basieren. Ein möglicher Algorithmus für die kombinierte Berechnung des NSV- und PSV-Arrays ist in [4] beschrieben. Die Laufzeit für die Generierung des Suffix-Arrays, NSV-Arrays und PSV-Arrays lässt sich durch die genannten Algorithmen auf $O(n)$ abschätzen. In der abschließenden Schleife repräsentiert die k-te Iteration den k-ten Faktor, wobei die Iteration für die Berechnung der Faktorlänge $O(|f_k|)$ Laufzeit benötigt. Damit ergibt sich eine Gesamtlaufzeit von $O(n + \underbrace{\sum_{i=1}^{z} |f_i|}_{n}) = O(n)$ für die Generierung der exakten LZ77-Faktorisierung. Der

Speicherbedarf des Algorithmus beträgt $O(n)$, da die Größe des Suffix-, NSV- und PSV-Arrays durch die Eingabelänge gegeben sind. Es sollte jedoch angemerkt werden, dass die Linearität des Speicherbedarfs einen hohen konstanten Faktor hat. Repräsentiert man die Elemente der genannten Arrays durch 32-Bit-Integer, so werden bereits $12n$ Byte Speicher benötigt. In dieser Betrachtung wurde jedoch der Speicherbedarf für die resultierende Faktorfolge nicht berücksichtigt.

3.2 Approximation der LZ77-Faktorisierung(Approx. LZ77)

3.2.1 Konzept

Im Rahmen dieser Arbeit wird in Anlehnung an [5] die erste Phase einer speichereffizienten Approximation des LZ77-Algorithmus betrachtet, welche wir im Folgendem mit Approx. LZ77 bezeichnen. Wie in [5] beschrieben, kann die Kombination aller drei Phasen des Algorithmus eine 2-Approximation bezüglich der Faktorrate ermöglichen. Die resultierenden Faktoren entsprechen jedoch ebenfalls dem LZSS-Schema 2.1, sodass eine verlustfreie Dekompression mit 2.1 möglich ist. Im Gegensatz zur exakten LZ77-Faktorisierung werden Referenzen nicht durch einen Greedy-Ansatz mit einem Scan von links nach rechts gefunden. Stattdessen wird eine Approximation der exakten LZ77-Faktorisierung erzeugt, die einen Tradeoff zwischen der Faktorrate und der Performanz, hier dem Speicherverbrauch, des Algorithmus darstellt. Die resultierenden Faktoren sind insbesondere dadurch definiert, dass ihre Länge einer Zweierpotenz entspricht. Analog dazu gehen wir ohne Beschränkung der Allgemeinheit davon aus, dass die Länge der Eingabe ebenfalls eine Zweierpotenz ist. Eine abweichende Eingabelänge kann stets durch entsprechendes Padding erreicht werden. Der Ablauf des Algorithmus ist durch mehrere Runden definiert, in denen Faktoren mit einer vorgegebenen Blockgröße extrahiert werden. In der ersten Runde beträgt die Blockgröße die Hälfte der Eingabelänge und im Übergang zwischen sukzessiven Runden wird die Blockgröße jeweils halbiert. Entsprechend werden Blöcke über der unverarbeiteten Eingabe in jeder Runde in der Hälfte geteilt. Der Algorithmus terminiert, wenn die Eingabe vollständig verarbeitet wurde, was spätestens nach $log_2(|S|)$ Runden der Fall ist, da Blöcke der Größe 1 nicht weiter geteilt werden können. Im Folgenden bezeichne B_i die Menge der Blöcke in Runde $i = 1, 2, ..., log_2(|S|)$. Innerhalb einer beliebigen Runde $i \in [1, log(|S|)]$ wird die Menge der Blöcke B_i auf Referenzen, also auf vorherige Vorkommen in S, untersucht. Im Erfolgsfall wird für den Block und dessen repräsentierter Zeichenfolge ein Faktor generiert. Weiterhin deklarieren wir den entsprechenden Block als markiert. Die Menge aller markierten Blöcke, die in der Runde i erzeugt wurden, wird durch B_i^{marked} mit $B_i^{marked} \subset B_i$ bezeichnet. Die verbleibenden Blöcke, $B_i^{unmarked} = B_i \setminus B_i^{marked}$, werden im Übergang zur nächsten Runde jeweils gespaltet. Im Anschluss an die letzte Runde

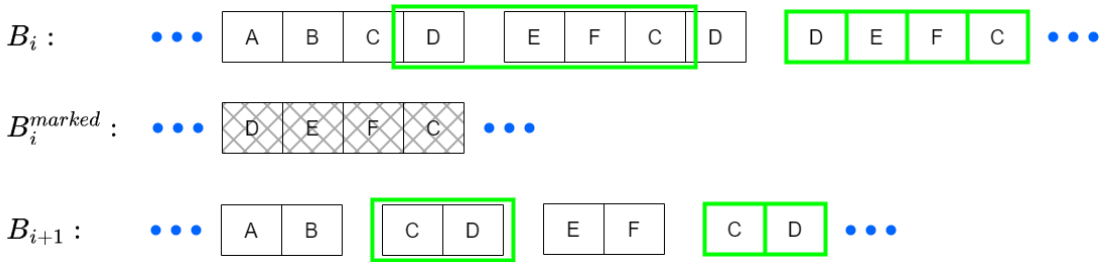


Abbildung 3.1: Die Abbildung illustriert die rundenbasierte Verarbeitung von Blöcken über der Eingabe in Approx. LZ77. In Runde i werden die Blöcke in B_i auf Referenzen untersucht und die Funde in die Menge der markierten Blöcke übernommen. Nur die unmarkierten Blöcke werden geteilt und in die nächste Runde übernommen.

werden die repräsentierten Zeichen der unmarkierten Blöcke referenzlos interpretiert und faktorisiert.

In 3.1 wird der Übergang zwischen zwei sukzessiven Runden an einem Beispiel verdeutlicht. Im Folgenden widmen wir uns einer algorithmischen Realisierung der beschriebenen Methodik. Insbesondere wird die Extraktion von Referenzen in jeder Runde konkret thematisiert. In 3.2 werden die notwendingen Prozesse vor, während und nach der rundenbasierten Ausführungsschleife von Approx. LZ77 beschrieben. Zu Beginn der initialen Runde $r_{init} = 1$ wird die gesamte Eingabe in $2^{r_{init}}$ Blöcke gleicher Größe eingeteilt. Dieser Prozess wird durch die Routine InitBlocks erfasst. Darauf folgt eine Schleife über eine beschränkte Anzahl an Runden, die durch $r_{end} = log_2(|S|)$ definiert ist. Die Routine ProcessRound beschreibt die bereits erwähnte Extraktion von Referenzen über die Menge der Blöcke, die eine Menge von markierten Blöcken und den zugehörigen Faktoren generiert. Im Übergang zur nächsten Runde werden die unmarkierten Blöcke in der Routine NextNodes jeweils gespaltet. Nach einer abschließenden Verarbeitung von referenzlosen Zeichen steht die Faktorfolge F bereit.

Die Funktionsweise der ProcessRound-Routine wird in 3.3 beschrieben. Die Ausführung kann in drei Schritte unterteilt werden. Im ersten Schritt werden die Tabellen RFPTable und RefTable mithilfe der Routine InitTables initialisiert. Die RFPTable ordnet jedem Rabin-Karp-Fingerprint(RFP), der über die Menge aller Blöcke berechnet wird, den linkesten Block zu, welcher diesen Wert ebenfalls aufweist. In diesem Rahmen vernachlässigen wir explizit Kollisionsfälle des RFP und gehen davon aus, dass die Gleichheit der RFPs zweier Zeichenfolgen ebenfalls die Gleichheit der Zeichenfolgen impliziert. Die RefTable speichert für jeden Block die Position des linkesten Vorkommens, also einer Referenz, seiner Zeichenfolge in der Eingabe, die zum aktuellen Zeitpunkt bekannt ist.

Die Routine InitTables, wie in 3.4 beschrieben, erzeugt die RFPTable, indem alle Blöcke von links nach rechts durchlaufen werden und das Paar aus einem Block und dessen RFP in die Tabelle nur dann eingefügt wird, wenn der RFP noch nicht vorhanden ist. Als Konsequenz wird für jeden RFP, der mindestens einem Block zugeordnet werden kann, der

Algorithmus 3.2 COMP$_{ApproxLZ77}$: Approximation der exakten LZ77-Faktorisierung durch eine blockweise Referenzsuche

Eingabe:$S = e_1...e_n$ *Ausgabe:*$F = f_1...f_z$

1: $F \leftarrow \emptyset$
2: $r_{init} \leftarrow 1$
3: $r_{end} \leftarrow log_2(|S|)$
4: $Blocks[1..2^{r_{init}}] \leftarrow InitBlocks(S, 2^{r_{init}})$ // Split S into $2^{r_{init}}$ equal blocks
5: **for** $r \leftarrow r_{init}$ **to** r_{end} **do**
6: $markedBlocks[1..z_r] \leftarrow ProcessRound(r, S, Blocks, F)$
7: $Blocks \leftarrow NextNodes(Blocks \setminus markedBlock[1..z_r])$ // Halve unmarked blocks
8: **end for**
9: **for** $e \in Blocks$ **do**
10: $F \leftarrow F.insert(0, e)$ // Process remaining Characters
11: **end for**
12: **return** F

Algorithmus 3.3 ProcessRound: Jede Runde enkapsuliert die Referenzsuche unter allen Blöcken und innerhalb der Eingabe.

Eingabe:$r, S, Blocks$ *Ausgabe:*$markedBlocks$

1: $(RFPTable, RefTable) \leftarrow InitTables(Blocks)$
2: $ReferenceScan(S, Blocks, RFPTable, RefTable)$
3: **for** $i \leftarrow 1$ **to** $|Blocks|$ **do**
4: **if** $RefTable[i] < Blocks[i].Pos$ **then**
5: $markedBlocks.insert(i)$
6: $F.insert(\frac{|S|}{2^r}, RefTable[i])$ // Ordered Insert
7: **end if**
8: **end for**
9: **return** $markedBlocks$

entsprechende linkeste Block gespeichert. Die RefTable wird im gleichen Ablauf erzeugt. Falls der RFP eines Blocks bereits in der RFPTable vorhanden ist, so stellt der zugehörige Blockeintrag eine Referenz dar. In diesem Fall wird die Position des eingetragenen Blocks in der RefTable eingetragen. Andernfalls ist der Block selbst das linkeste Vorkommen seiner Zeichenfolge, sodass die eigene Position eingetragen wird. Es ist zu betonen, dass die InitTables-Routine durch ihre Aktualisierungen der RefTable bereits markierte Blöcke bzw. Faktoren implizit erzeugt.

Im zweiten Schritt, dem RefereneScan 3.5, werden zusätzliche Referenzen innerhalb der gesamten Eingabe S gesucht. In einer beliebigen Runde r wird der RFP eines Fensters der Größe $\frac{|S|}{2^r}$ über der Eingabe berechnet und mit den Einträgen der RFPTable verglichen.

Algorithmus 3.4 InitTables: Initialisierung der RFPTable und RefTable durch einen Scan der RFPs aller Blöcke

Eingabe:$Blocks$ *Ausgabe:*$RFPTable, RefTable$

1: $RFPTable \leftarrow HashTable[RFP, LeftMostBlock]$
2: $RefTable[1..|S|] \leftarrow (0, ..., 0)$
3: **for** $i \leftarrow 1$ **to** $|Blocks|$ **do**
4: $RFP \leftarrow RFP(Blocks[i])$
5: **if** RFP **in** $RFPTable$ **then**
6: $RefTable[i] \leftarrow RFPTable[RFP].Pos$
7: **else**
8: $RFPTable.insert(RFP, Blocks[i])$
9: $RefTable[i] \leftarrow Blocks[i].Pos$
10: **end if**
11: **end for**
12: **return** $RFPTable, RefTable$

Im Falle eines Treffers wird die Position des Fensters in der RefTable eingetragen, falls er den vorherigen Wert unterbietet. Die Berechnung des RFP des Fensters in Folge der sukzessiven Verschiebung um eine Position kann wie in 2.4 beschrieben in konstanter Zeit durchgeführt werden. Schließlich wird im dritten Schritt die RefTable genutzt, um die implizit markierten Blöcke zu extrahieren. Ein Block wird markiert, wenn die RefTable die Position einer Referenz indiziert, die kleiner als die des Blocks ist. Die Position der Referenz und die rundenbedingte Blockgröße als Faktorlänge definieren einen eindeutigen Faktor, der in die Menge der Faktoren F eingefügt wird. Die Reihenfolge der Faktoren wird endgültig durch die Position der repräsentierten Zeichenfolge bestimmt. Für die Einhaltung der Reihenfolge sei im Rahmen der Analyse der Laufzeit und des Speicherverbrauchs eine nachträgliche Sortierung der Faktoren in F vorgegeben.

Algorithmus 3.5 ReferenceScan: Suche zusätzliche Referenzen durch einen Scan der gesamten Eingabe

Eingabe:$S, Blocks, RFPTable, RefTable$

1: blockSize $\leftarrow \frac{|S|}{|Blocks|}$
2: $RFP \leftarrow RFP(S[1..blockSize])$
3: **for** $i \leftarrow 1$ **to** $|S| - blockSize$ **do**
4: **if** RFP in $RFPTable$ **and** $i < RefTable[RFPTable[RFP]]$ **then**
5: $RefTable[RFPTable[RFP]] \leftarrow i$
6: **end if**
7: $RFP \leftarrow RFP(S[i+1..i+blockSize])$ // Rolling Hash
8: **end for**

3.2.2 Theoretisches Laufzeit- und Speicherverhalten

Die Laufzeit des Algorithmus wird durch die Anzahl der Runden und der Extraktion von Referenzen in jeder Runde bestimmt. Die Anzahl der Runden beträgt maximal $log_2(|S|) = log_2(n)$. In jeder Runde werden Referenzen unter den Blöcken durch die InitTables-Routine bestimmt. Die Menge aller Blöcke, die in dieser Routine bearbeitet werden, decken maximal die gesamte Eingabe S ab. Die Laufzeit der Routine erhält durch die Berechnung des RFPs über dieser Zeichenfolge eine Laufzeitschätzung von $O(n)$. Die Referenzsuche über die gesamte Eingabe durch die ReferenceScan-Routine benötigt ebenfalls $O(n)$ Laufzeit, da das berücksichtigte Fenster in linearer Zeit über die Eingabe verschoben werden kann. Durch die Verwendung einer Hashtabelle sind die Suchoperationen jeweils in konstanter Zeit durchführbar. Die Laufzeit der ProcessRound-Routine beträgt somit $O(n)$. Insgesamt ergibt sich eine Gesamtlaufzeit von $O(n \log n)$ für Approx. LZ77. Der Speicherbedarf des Algorithmus wird durch die Größe der Blockmenge, RFPTable und der RefTable bestimmt, die wiederum alle durch die Anzahl der Blöcke bestimmt werden. In jeder Runde repräsentiert die Menge der Blöcke die noch unverarbeitete Eingabe, sodass aus jedem Block im Laufe des Algorithmus mindestens ein Faktor extrahiert wird. Die Anzahl der Blöcke einer Runde übersteigt damit nie die Größe der endgültigen Faktorfolge, $|F| = z$. Der Speicherbedarf kann damit konservativ auf $O(z)$ abgeschätzt werden.

3.3 Parallelisierung von Approx. LZ77(Approx. LZ77Par)

3.3.1 Konzept

Im Folgenden beschreiben wir die verwendete Methodik zur Parallelisierung von Approx. LZ77. Die parallele Variante des Algorithmus, die wir Approx. LZ77Par nennen, basiert auf der abgeschlossenen Parallelisierung von Abschnitten von Approx. LZ77, die eine chronologische Abhängigkeit aufweisen. Für die Parallelisierung der Verarbeitung wird jeweils die Eingabe auf verschiedene Prozessoren bzw. Threads verteilt und ein gemeinsames Ergebnis erzeugt. In 3.2 wird die parallele Generierung der initialen Blöcke dargestellt. Hierbei wird

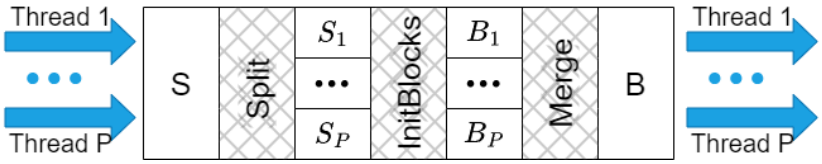


Abbildung 3.2: Parallele Generierung der initialen Blöcke

die Eingabe S in P Teile aufgebrochen, sodass die Routine InitBlocks auf jedem Prozessor die zugehörigen Blöcke erzeugen kann. Da die Ordnung und die Größe der Ergebnisse bekannt ist, kann eine umfassende Blockmenge ohne zusätzlichen Aufwand erzeugt werden. In 3.3 wird die parallele Initialisierung der RFPTable und der RefTable dargestellt. Die

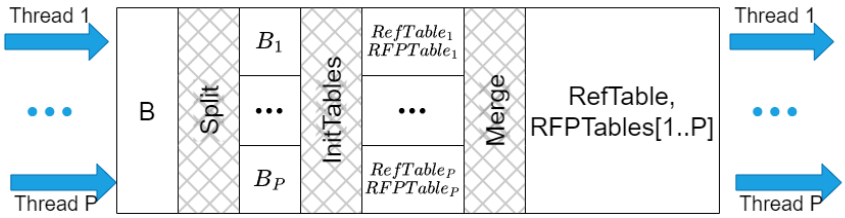


Abbildung 3.3: Parallele Initialisierung der RFP-Tabelle und der Referenztabelle

Menge aller Blöcke wird in P Teile aufgeteilt, wobei als Kriterium für die Aufteilung der RFP jedes Blocks verwendet wird. Als Konsequenz werden identische Zeichenfolgen nicht auf verschiedene Prozessoren verteilt. Jeder Prozessor erzeugt eine eigene RFPTable, wobei die RefTable durch alle Prozessoren gemeinsam aktualisiert wird. Auch hier wird durch den RFP als Aufteilungsschlüssel eine Überlappung der Zugriffe vermieden. Das Ergebnis ist eine konsistente RefTable und eine P-elementige Menge von Instanzen der RFPTable, die jeweils eine klar abgegrenzte Teilmenge aller RFPs abdecken.

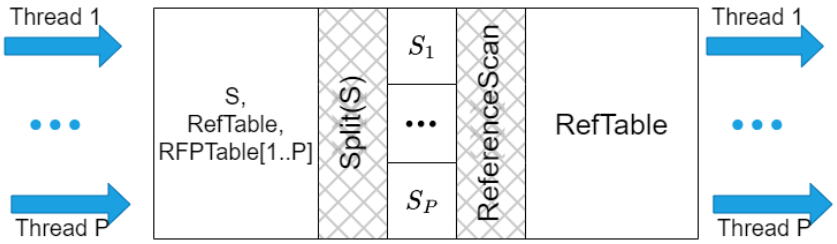


Abbildung 3.4: Paralleler Scan der Eingabe nach zusätzlichen Referenzen

In 3.4 wird der parallele Scan der Eingabe S nach zusätzlichen Referenzen dargestellt. Die Eingabe S wird in P Teile aufgeteilt. Auf jede Teilfolge wird die sequenzielle Routine ReferenceScan angewendet, wobei die RefTable durch alle Prozessoren gemeinsam aktualisiert wird und ein potenziell paralleler Lesezugriff auf Instanzen der RFPTable stattfindet. Schließlich müssen die implizit erzeugten Faktoren in die Menge der Faktoren F eingefügt werden. Da die Reihenfolge der Faktoren erhalten bleiben muss, wird eine nachträgliche parallele Sortierung verwendet. Die Implementierung der parallelen Sortierung ist jedoch nicht Gegenstand dieser konzeptuellen Analyse und wird daher nicht weiter betrachtet. Eine mögliche Implementierung ist in [16] beschrieben.

3.3.2 Theoretisches Laufzeit- und Speicherverhalten

Da die Eingabe stets in gleiche Teile aufgeteilt und die Ausgabe ohne einen relevanten Aufwand kombiniert werden kann, kann eine theoretische Laufzeit von $O(\frac{n \log n}{P})$ geschätzt werden, wobei P die Anzahl der Prozessoren definiert. Der Speicherbedarf des Algorithmus beträgt weiterhin $O(z)$. Dies stellt jedoch eine ideale Abschätzung dar, die in der

Praxis nicht erreicht werden kann. Insbesondere die Interaktion mit dem Speicher und die Kommunikation zwischen den Prozessoren führen zu einer oberen Schranke des Speedups.

3.4 Praktische Optimierungen

Im Folgenden betrachten wir optionale Optimierungen, die die durchschnittliche Laufzeit von Approx.LZ77 und Approx.LZ77Par verbessern können auf Kosten von anderen Metriken. Jede einzelne Technik ist optional und unabhängig von den Anderen nutzbar, wobei eine positive Korrelation zu erwarten ist.

3.4.1 Dynamische Endrunde(DynEnd) - Laufzeit vs. Qualität*

Sei eine beliebige lineare Kodierung K_{OUT} für die Übersetzung der erzeugten Faktorenfolge F gegeben. Der Wert,

$$Min_{Bin}^{Ref} = min\{|K_{OUT}(f)||f \in F, f \text{ ist Referenz}\} \tag{3.3}$$

gibt die minimale Anzahl an Bits an, die für die Kodierung einer beliebigen Referenz benötigt wird. Analog dazu beschreibt

$$Max_{Bin}^{Lit} = max\{|K_{OUT}(f)||f \in F, f \text{ ist Zeichen}\} \tag{3.4}$$

die maximale Anzahl an Bits, die für die Kodierung eines referenzlosen Zeichens benötigt wird. Sei f_{ref} ein beliebiger referenzierender Faktor, welcher $|f_{ref}| \leq \frac{Min_{Bin}^{Ref}}{Max_{Bin}^{Lit}}$ Zeichen referenziert. Die referenzierte Zeichenfolge von f_{ref} wird im Folgenden als S_{ref} mit $|S_{ref}| = |f_{ref}|$ bezeichnet. Dann gilt für die Länge der kodierten Repräsentation von f_{ref}:

$$\begin{aligned} |K_{OUT}(f_{ref})| &\geq Min_{Bin}^{Ref} \\ &\geq |f_{ref}| \cdot Max_{Bin}^{Lit} \\ &\geq \sum_{i=1}^{|f_{ref}|} |K_{OUT}((0, S_{ref}[i]))|. \end{aligned} \tag{3.5}$$

Es folgt, dass ein referenzierender Faktor, dessen Länge eine obere Schranke von $\frac{Min_{Bin}^{Ref}}{Max_{Bin}^{Lit}}$ Zeichen nicht überschreitet, nicht effizient kodiert werden kann. Stattdessen sollten die referenzierten Zeichen einzeln kodiert werden. Die Technik der dynamischen Endrunde greift diese Idee auf, indem Referenzen unterhalb einer Grenzlänge nicht extrahiert werden. Gibt uns die Kodierung eine Grenzlänge $l_{min}^{ref} = \lceil \frac{Min_{Bin}^{Ref}}{Max_{Bin}^{Lit}} \rceil$ vor, so kann der Algorithmus in Runde $r_{DynEnd} = \lceil log_2|S| - log_2 l_{min}^{ref} \rceil$ terminieren. Da potenziell referenzierende Faktoren aufgebrochen werden, kann die Qualität der Faktorisierung sinken, wobei das binäre Endprodukt kleiner wird. Es ergibt sich also eine steigende Faktorrate bei einer gleichzeitig sinkenden Kompressionsrate.

$$CR_{DynEnd}^{Approx.LZ77} \leq CR^{Approx.LZ77} \tag{3.6}$$

$$FR_{DynEnd}^{Approx.LZ77} \geq FR^{Approx.LZ77} \tag{3.7}$$

3.4.2 Dynamische Startrunde(DynStart) - Laufzeit vs. Speicher

Gegeben seien zwei initiale Runden r_{init1} und r_{init2} mit $1 \leq r_{init1} < r_{init2} \leq log_2|S|$, die auf der gesamten Eingabe S angewendet werden, so wird die Eingabe jeweils in $2^{r_{init1}}$ bzw. $2^{r_{init2}}$ Blöcke gleicher Größe eingeteilt. Die Mengen der Blöcke werden im Folgenden als B_{init1} bzw. B_{init2} bezeichnet. Im Rahmen der Bearbeitung der Runden wird eine Menge von markierten Blöcken $B_{init1}^{marked} \subset B_{init1}$ bzw. $B_{init2}^{marked} \subset B_{init2}$ erzeugt, für die ein vorheriges Vorkommen bestimmt wurde. Aufgrund der Natur der Blockspaltung in jeder Runde, kann jedem Block in B_{init1} eine Gruppe von $2^{r_{init2}-r_{init1}}$ Blöcken in B_{init2} zugeordnet werden, die die gleiche Zeichenfolge repräsentieren. Die Folgerung lässt sich insbesondere auch auf die markierten Blöcke anwenden, sodass die folgende Beziehung hergeleitet werden kann:

$$|B_{init2}^{marked}| \geq 2^{r_{init2}-r_{init1}} \cdot |B_{init1}^{marked}|. \tag{3.8}$$

Weiterhin folgt, dass die Existenz eines markierten Blocks in B^{init1} die Existenz von $2^{r_{init2}-r_{init1}}$ benachbarten markierten Blöcken in B^{init2} impliziert. Die Umkehrung dieser Aussage liefert,

$$longestChain(B_{init2}^{marked}) < 2^{r_{init2}-r_{init1}} \Rightarrow B_{init1}^{marked} = \emptyset, \tag{3.9}$$

wobei $longestChain(B_{init2}^{marked})$ die Größe der längsten Kette von benachbarten markierten Blöcken in B_{init2}^{marked} bezeichnet. Die Technik der dynamischen Startrunde greift diese Beziehung auf, indem initial die Runde $r_{init} = log_2|S|/2$ auf die gesamte Eingabe S angewendet wird. Im Anschluss kann der Wert $longestChain(B_{init}^{marked})$ mithilfe eines Scans über die markierten Blöcke bestimmt werden. Der errechnete Wert impliziert eine Runde $r_{DynStart}$ derart, dass vorherige Runden garantiert keine markierten Blöcke erzeugen und damit ausgelassen werden können. Der Wert $r_{DynStart}$ ergibt sich wie folgt,

$$r_{DynStart} = r_{init} - \begin{cases} -1, & \text{falls } longestChain(B_{marked}^{init}) = 0 \\ \lfloor log_2 longestChain(B_{marked}^{init}) \rfloor, & \text{sonst.} \end{cases} \tag{3.10}$$

In Abhängigkeit von der Beschaffenheit der Eingabe, kann maximal die Hälfte aller Runden ausgelassen werden, ohne eine Veränderung der Ergebnisse zu verursachen. In Runde $r_{init} = log_2|S|/2$ werden jedoch $2^{log_2|S|/2} = \sqrt{|S|}$ Blöcke erzeugt. Dies führt zu einer zusätzlichen unteren Schranke für den Speicheraufwand des Algorithmus. Falls diese Technik angewandt wird, kann der Speicheraufwand mit $O(\max\{\sqrt{n}, z\})$ abgeschätzt werden.

3.4.3 Vorberechnete Runde(PreMatching) - Laufzeit vs. Speicher

Analog zu der dynamischen Startrunde kann eine vorberechnete Runde ebenfalls genutzt werden, um den Arbeitsaufwand vorheriger Runden zu reduzieren. Sei $r_{PreMatch}$ mit $1 \leq r_{PreMatch} \leq log_2|S|$ eine Runde, die auf die gesamte Eingabe angewendet wird. Als Ergebnis erhalten wir die Menge der markierten Blöcke, $B_{PreMatch}^{marked}$. Weiterhin speichern wir uns

die RFPs aller Blöcke, die im Rahmen der Runde generiert wurden. Wie in 3.4.2 gezeigt, kann jedem Block in einer vorherigen Runde einer Gruppe von Blöcken in einer späteren Runde zugeordnet werden, die die gleiche Zeichenfolge repräsentieren. Die Konkatenation von Zeichenfolgen kann in Anlehnung an 2.5 auf eine Operation auf der Basis des RFP abgebildet werden. Gegeben sei eine Runde r_m mit $1 \leq r_m \leq r_{PreMatch}$. Für einen beliebigen Block $b \in B_m$ können $2^{r_{PreMatch}-r_m}$ viele Böcke $(b_1, b_2, ..., b_{2^{r_{PreMatch}-r_m}}) \in B_{prematch}$ gefunden werden, deren Konkatenation die gleiche Zeichenfolge repräsentieren. So ergibt sich für den zugehörigen RFP,

$$RFP(b) = RFP(b_1 \cdot b_2 \cdot ... \cdot b_{2^{r_{PreMatch}-r_m}}), \tag{3.11}$$

wobei jede Konkatenation in eine Operation mit konstanter Laufeit für die RFPs abgebildet werden kann. Die Anzahl der Rechenschritte für die Berechnung des RFP eines Blockes hängt nun nicht mehr von der Länge der repräsentierten Zeichenfolge, sondern der Rundendistanz zur vorberechneten Runde ab. Weiterhin kann die Menge der unmarkierten Blöcke $B^{unmarked}_{PreMatch} = B_{PreMatch} \setminus B^{marked}_{PreMatch}$ genutzt werden, um die Menge der Blöcke B_m in Runde r_m zu filtern. Sei ein Block $b \in B^{marked}_m$ gegeben, so muss die equivalente Sequenz von Blöcken $(b_1, b_2, ..., b_{2^{r_{PreMatch}-r_m}}) \in B_{prematch}$ auch markiert sein bzw. Teil von $B^{marked}_{prematch}$ sein. Die Umkehrung dieser Aussage liefert einen Filter für alle vorherigen Runden, um Blöcke auszugrenzen, die keine Kandidaten für eine Markierung sind. Die zusätzlich gespeicherten Daten erhöhen den Speicherbedarf des Algorithmus. Sei $r_{PreMatch} \in [1, log_2|S|]$ der Index der vorberechneten Runde, so kann der resultierende Speicherbedarf mit $O(\max\{2^{r_{PreMatch}}, z\})$ abgeschätzt werden.

3.4.4 Minimale Tabellengröße(ScanSkip) - Laufzeit vs. Qualität

Wie in 3.4 beschrieben, wird die RFPTable und die RefTable durch die InitTables-Routine initialisiert. Im Rahmen der Routine werden alle Blöcke auf Duplikate überprüft, sodass nur einzigartige Blöcke in die RFPTable eingefügt werden und die RefTable resultierende implizite Faktoren speichert. Die Anzahl der Blöcke, die im nachfolgenden Schritt in der ReferenceScan-Routine 3.5 markiert werden können, ist durch die Anzahl der Einträge in der RFPTable beschränkt. Der Wert $k \in [0, 1]$ gebe den Anteil der Einträge in der RFPTable in Relation zur Gesamtzahl der Blöcke an. Unsere Optimierung sieht vor, dass in jeder Runde die ReferenceScan-Routine nur angewendet wird, falls k größer als ein Schwellwert $k_{min} \in [0, 1]$ ist. Faktoren, die durch die ausgelassene Suche in dieser Runde nicht erzeugt wurden, werden in einer der nächsten Runden ausgegeben, wobei diese durch die Halbierung der Blöcke sukzessive in ihrer Anzahl verdoppelt werden. Damit steigt zwangsläufig die Faktorrate, wobei die ausgelassene Referenzsuche einen zeitlichen Gewinn bringt. Es wurde bereits etabliert, dass die Anzahl der Blöcke in jeder Runde durch die Anzahl der Faktoren, z, beschränkt ist. Die Anzahl der Faktoren, die durch die Auslassung von ReferenceScan in einer Runde nicht erzeugt werden, ist durch $k_{min} \cdot z$ beschränkt.

Ähnlich dazu ist die Anzahl der zusätzlich erzeugten Faktoren in der nächsten Ausführung der Referencescan-Routine durch $2 \cdot k_{min} \cdot z$ beschränkt. Die Häufigkeit dieser Ereigniskette hängt von der Beschaffenheit der Eingabe ab, wobei die Hälfte der Anzahl der Runden als konservative obere Schranke genutzt werden kann. Insgesamt ergibt sich eine konservative Schätzung für die relative Verschlechterung der Faktorrate mit

$$FR^{Approx.LZ77} \leq FR^{Approx.LZ77}_{ScanSkip} \leq FR^{Approx.LZ77} * (1 + k_{min} \cdot log_2|S|). \tag{3.12}$$

3.4.5 Korrelation der Optimierungen

Die Einschränkung der initialen und terminalen Runde durch DynStart und DynEnd reduziert die potenzielle Rundendistanz zu einer gewählten vorberechneten Runde, $r_{PreMatch}$. Dies unterstützt die zeitsparende Wirkung von PreMatching. Die Filterung von Blöcken, die keine Kandidaten eine Markierung sind, durch PreMatching erlaubt eine Reduktion der RFPTable, die durch die InitTables-Routine erzeugt wird. Entsprechend kann für die ScanSkip-Optimierung ein kleinerer Wert für k_{min} gewählt werden, um die Verschlechterung der Faktorrate bei gleicher Häufigkeit der ausgelassenen ReferenceScan-Routine zu reduzieren. Diese Überlegungen stellen eine zu erwartende Korrelation der Optimierungen dar, wobei die spezifischen Auswirkungen von der Beschaffenheit der Eingabe und der Wahl der Parameter abhängen.

Kapitel 4

Praktische Evaluation

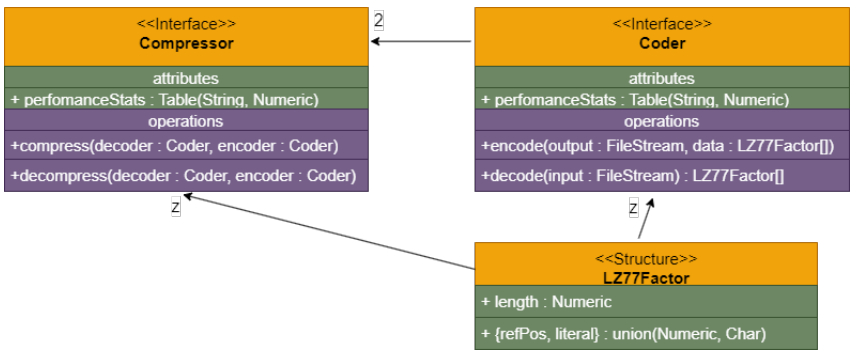
4.1 Testumgebung

Die folgenden Experimente wurden auf mithilfe einer AMD EPYC 7763 64-Core CPU mit 16 nutzbaren Hardwarethreads und 64GB Arbeitsspeicher durchgeführt. Das System verwendet Ubuntu 24.04 als Betriebssystem und GCC in der Version 13.2.0 als Compiler. Die Algorithmen wurden in C++20 implementiert und mit der Optimierungsstufe `-O3` kompiliert. Die Ausführung der Algorithmen mit einer spezifischen Anzahl von Threads wurde softwareseitig über OpenMP-Instruktionen [1] realisiert.

4.2 Implementierung

4.2.1 Klassenstruktur

Abbildung 4.1: Klassenstruktur der Implementierung



Die in 4.1 dargestellte Klassenstruktur illustriert die grundlegende Abstraktion, die für die Implementierung der Algorithmen verwendet wurde. Compressor und Coder beschreiben jeweils ein Interface bzw. ein Template, welches durch ein konkretes Kompressionsverfahren und einer Kodierung spezialisiert werden kann. Jegliche Spezialisierungen teilen sich jedoch eine gemeinsame Definition eines Faktors im LZ77-Schema.

4.2.2 Externe Bibliotheken

Im Folgenden werden die genutzten externen Bibliotheken aufgelistet, die im Rahmen der Implementierung der Algorithmen, sowie deren Evaluation genutzt wurden.

Malloc Count

Malloc Count [2] ist eine C++-Bibliothek, die es ermöglicht, Speicherallokationen und -freigaben auf dem Heap zu überwachen und zu messen. Im Rahmen unserer Evaluation gibt uns diese Bibliothek die Spitze des allokierten Speichers innerhalb der Ausführung der Algorithmen aus.

Unordered-Dense-Map

Die Unordered-Dense-Map [10] stellt eine performante Hashtabelle dar, die insbesondere in der Dauer ihrer Suchoperationen gegenüber der std::unordered_map aus der Standardbibliothek deutlich trumpft. Dies Hashtabelle wurde in Approx.LZ77 und Approx.LZ77Par für die Speicherung der RFPs in jeder Runde verwendet. Im Rahmen der Entwicklung von Approx.LZ77Par hat sich die Verwendung mehrerer Instanzen von Unordered-Dense-Map im Vergleich zu anderen inherent parallelen Hashtabellen [7] [15] als effizienter herausgestellt.

LibSaiS

Für die Implementierung der exakten LZ77-Faktorisierung, die als Referenzalgorithmus fungiert, wurde die Bibliothek LibSaiS [6] zu Hilfe genommen. Diese Bibliothek stellt eine effiziente Implementierung für die Konstruktion des Suffix-Arrays bereit.

STL - Sortierung

Die Standardbibliothek von C++ stellt bereits Implementierungen von Sortieralgorithmen bereit. Diese wurden in der Implementierung von LZ77 und Approx.LZ77 für die nachträgliche Sortierung der Faktoren verwendet. Insbesondere wurden Execution Policies [17] genutzt, um die Sortierung parallelisieren zu können.

4.2.3 Parametrisierte Einstellung

Die folgenden Einstellungen definieren eine Kalibrierung von Approx. LZ77 und Approx. LZ77Par in der jeweils optimierten(opt) bzw. unoptimierten(unopt) Variante. Die Einstellungen für beide Varianten sind in der Tabelle 4.1 aufgeführt.

Falls nicht anders ausgewiesen, wurden die Approximationsalgorithmen in der opt-Variante mit den in Tabelle 4.1 festgelegten Parametern ausgeführt. In der Ausführung von Approx.LZ77Par wurde die Ausführung standardmäßig mit 16 Threads durchgeführt.

Tabelle 4.1: Parameter und Einstellung für die unoptimierte(unopt) und optimierte(opt) Ausführung von Approx.LZ77

Einstellung	unopt	opt
DynEnd	Deaktiviert	$r_{DynEnd} = 26$
DynStart	Deaktiviert	Aktiv
PreMatching	Deaktiviert	$r_{PreMatch} = 23$
ScanSkip	$k_{min} = 0\%$	$k_{min} = 3\%$

4.3 Messung

4.3.1 Eingabedaten

Die folgenden Algorithmen wurden auf verschiedenen Dateien aus dem Pizza & Chili-Corpus [12] getestet. Die verwendeten Dateien decken verschiedene Kontexte und damit Kompressionspotentiale ab. In der Tabelle 4.2 sind die verwendeten Dateien aufgelistet. Die

Tabelle 4.2: Auflistung der verwendeten Eingabedaten aus dem Pizza & Chili-Corpus [12]

Datei	Größe	Alphabetgröße	Beschreibung
`dna`	200MB	4	DNA-Sequenzen
`english`	200MB	256	Englische Texte
`proteins`	200MB	20	Proteinsequenzen
`sources`	200MB	256	Quellcode
`xml`	200MB	256	XML-Dateien

Größe der Dateien wurde auf 200MB beschränkt, um einen angemessenen Rahmen für die Laufzeitmessung zu erhalten. Detaillierte Messungen werten wir auf der Eingabe proteins aus, da diese Datei die schlechtesten Metriken aufweist. Weitergehende Laufzeitmessungen der anderen Dateien sind in A.1 aufgeführt.

4.3.2 Messgrößen

Laufzeit

Die Laufzeit der Algorithmen wurde innerhalb der Ausführung gemessen. Dabei wird die Zeitmessung nach dem Laden der Eingabedatei gestartet und mit dem vollständigen Auffüllen einer Faktorfolge beendet. Damit wird das Einlesen der Eingabe und eine eventuelle Kodierung der Ausgabe nicht in die Laufzeitmessung einbezogen. Diese Strategie hat ihren Hintergrund in der Tatsache, dass die konkrete Ausprägung des Eingabe- und Ausgabestroms keine Aussagekraft über die Qualität der Kompression hat.

Speicher

Der Speicherverbrauch der Algorithmen wurde auch intern mithilfe einer externen Bibliothek 4.2.2 gemessen. Dabei wurden Speicherallokationen auf dem Heap überwacht und gemessen. Im Rahmen dieser Arbeit wurde die Spitze des allokierten Speichers im Zeitraum nach dem Einlesen der Eingabedatei und nach dem vollständigen Auffüllen der Faktorfolge gemessen. Zu Vergleichszwecken wird der Speicherverbrauch in Relation zur Eingabegröße angegeben.

Kompressionsrate CR*

Die Kompressionsrate wird neben der Anzahl der Faktoren zum Großteil von der verwendeten Kodierung bestimmt. Wie bereits erwähnt, sind wir in der Wahl der Kodierung nicht beschränkt, sodass die Aussagekraft bezüglich der Qualität der Kompression eingeschränkt ist. Es ist jedoch zu beachten, dass Faktoren, die durch Approx.LZ77 erzeugt werden stets eine Zweierpotenz als Länge annehmen. Die binäre Repräsentation dieser Längen kann daher in Abhängigkeit von der gewählten Kodierung kompakt konstruiert werden. Um dieses Phänomen zu illustrieren, geben wir im Folgenden eine naive Kodierung vor, auf dessen Grundlage wir die Kompressionrate CR^* definieren.

$$|K_{OUT}^{LZ77}(f)| = 1 + \begin{cases} 2\log_2(n) & \text{falls } |f| > 1 \\ 8 & \text{sonst} \end{cases} \tag{4.1}$$

Im Falle von LZ77 bestimmt ein Bit, ob es sich um eine Referenz oder ein einzelnes Zeichen handelt. Im Falle einer Referenz wird die Länge und die Position der Referenz mithilfe von $\log_2(n)$ Bits kodiert. Im Falle eines einzelnen Zeichens wird die bereits definierte Kodierung K_{IN} aus 2.6 mit 8 Bits genutzt.

$$|K_{OUT}^{Approx.LZ77}(f)| = 1 + \begin{cases} \log_2(n) + \log_2(\log_2(n)) & \text{falls } |f| > 1 \\ 8 & \text{sonst} \end{cases} \tag{4.2}$$

Im Falle von Approx.LZ77 kann die Länge mithilfe von $\log_2(\log_2(n))$ Bits kodiert werden, da die Länge anhand einer einzelnen Bitposition bestimmt wird. In Anlehnung an die Größe der verwendeten Testdaten erhalten wir für die Optimierung, DynEnd, die folgende Endrunde:

$$r_{DynEnd} = \lceil log_2|S| - log_2\lceil \frac{Min_{Bin}^{Ref}}{Max_{Bin}^{Lit}} \rceil \rceil = 26 \tag{4.3}$$

4.3.3 Messwerte

Tabelle 4.3: Messwerte der Algorithmen auf verschiedenen Eingabedateien

Eingabe	Algorithmus	Laufzeit[s]	Speicher	FR_{unopt}	FR	CR*
proteins	LZ77	15.76	14.88		9.95%	70.92%
	Approx.LZ77	44.06	9.94	15.07%	15.34%	63.95%
	Approx.LZ77Par	6.99	10.21	15.07%	15.34%	63.95%
sources	LZ77	13.79	13.44		5.50%	39.20%
	Approx.LZ77	40.43	6.42	9.64%	10.05%	40.14%
	Approx.LZ77Par	6.49	5.90	9.64%	10.05%	40.14%
english	LZ77	15.32	13.44		6.66%	47.45%
	Approx.LZ77	51.00	7.06	10.22%	10.42%	43.39%
	Approx.LZ77Par	7.46	6.16	10.22%	10.42%	43.39%
dna	LZ77	14.91	13.44		6.66%	47.46%
	Approx.LZ77	30.10	8.38	10.67%	10.71%	45.53%
	Approx.LZ77Par	4.80	6.66	10.67%	10.71%	45.53%
xml	LZ77	13.21	12.72		3.35%	23.89%
	Approx.LZ77	29.38	3.46	6.41%	6.62%	26.78%
	Approx.LZ77Par	4.91	3.46	6.41%	6.62%	26.78%

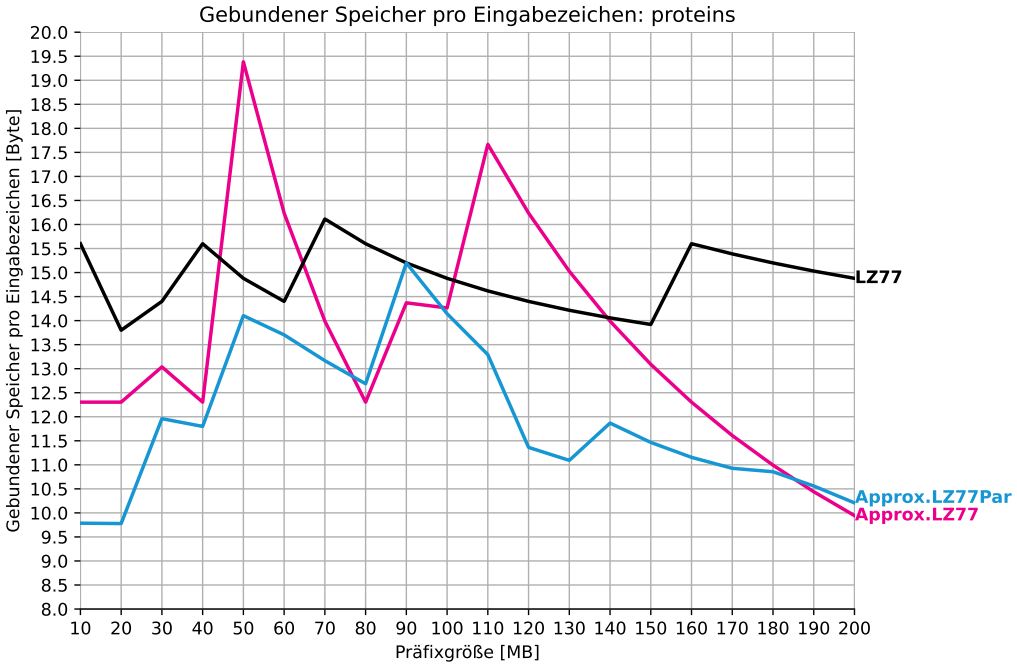


Abbildung 4.2: Speicherverbrauch von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von proteins. Aufgezeichnet wurde das Verhältnis von allokiertem Speicher zur Eingabegröße.

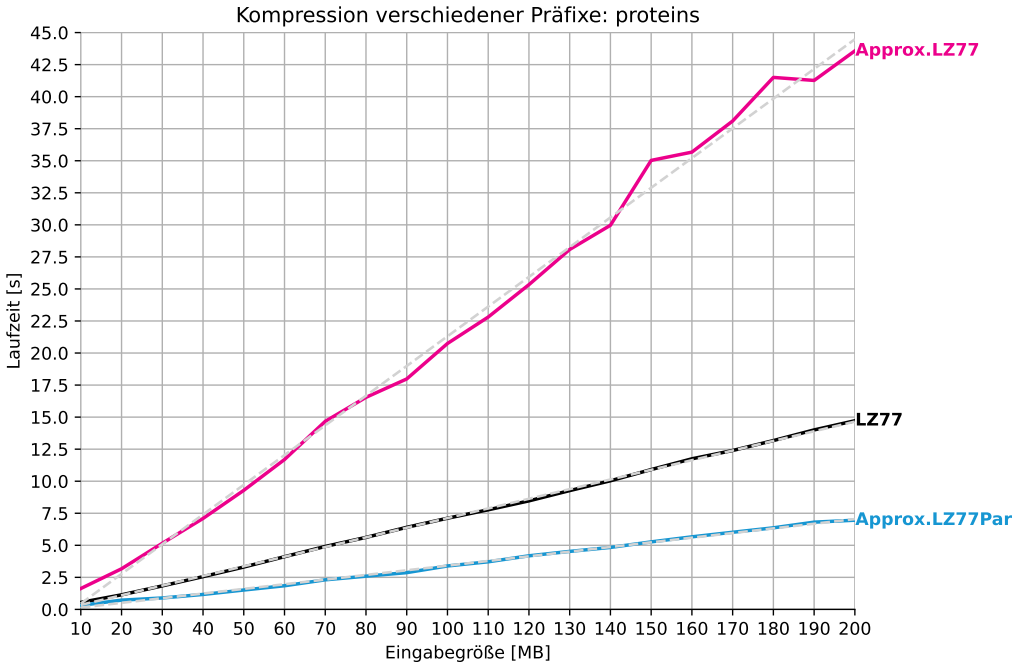


Abbildung 4.3: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von proteins. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

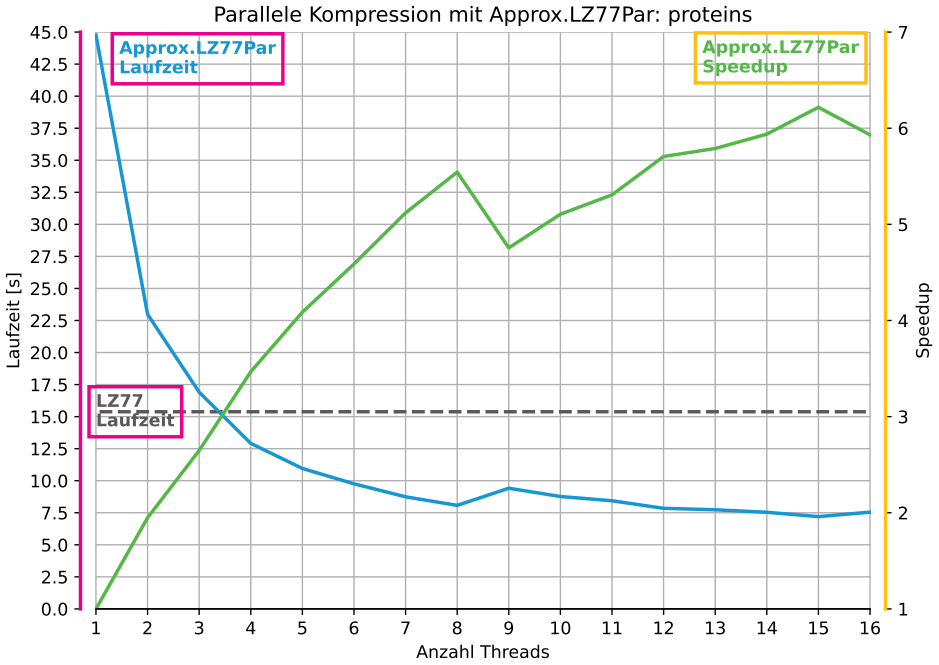


Abbildung 4.4: Laufzeit-(Blau) und Speedup(Grün)-Messung von Approx.LZ77Par mit verschiedener Anzahl an Threads für proteins

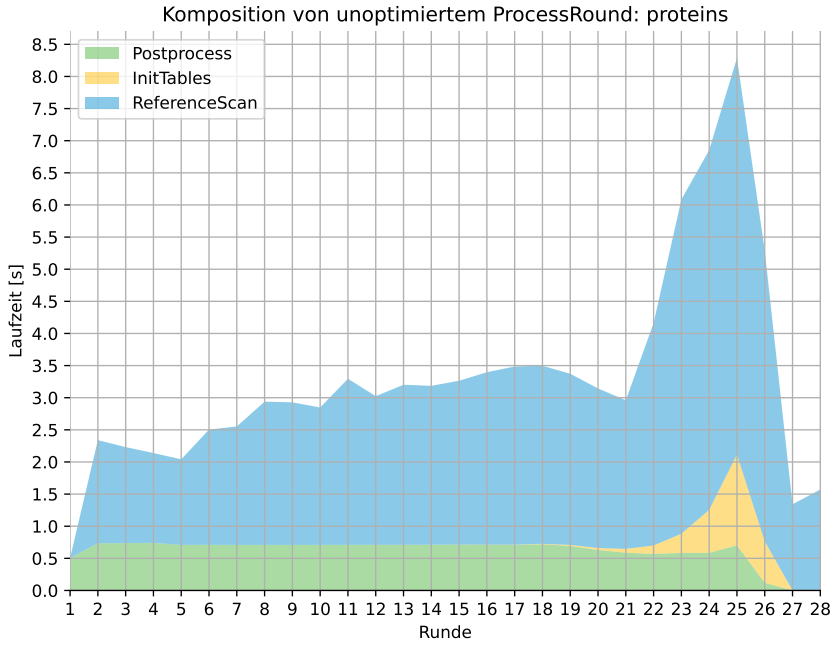


Abbildung 4.5: Darstellung der Verteilung der Laufzeit der Subroutinen innerhalb jeder Runde im Falle einer Ausführung von Approx.LZ77 ohne jegliche Optimierungen 4.1. Postprocess deckt dabei die nötigen Prozesse zum Rundenübergang, wie das Spalten der Blöcke und der Extraktion der Faktoren, ab.

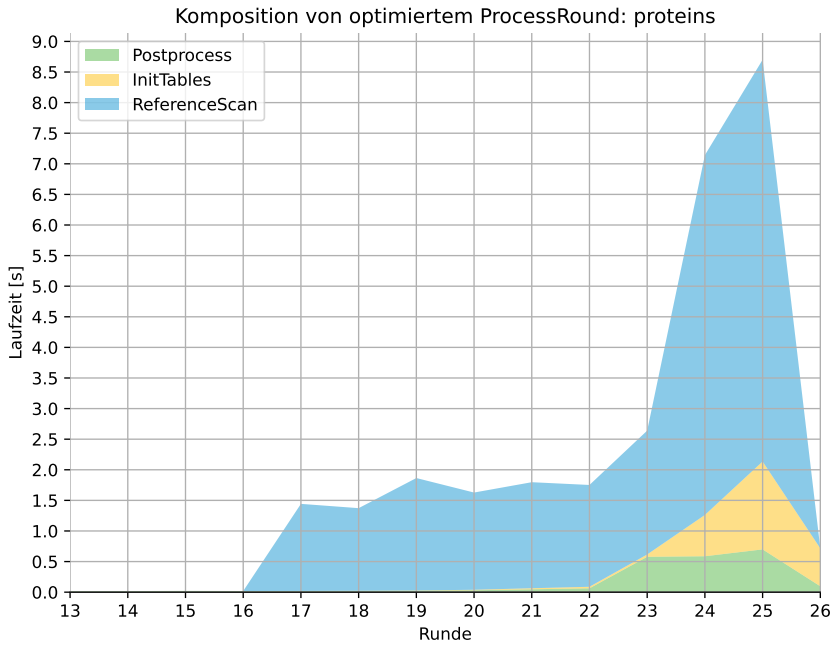


Abbildung 4.6: Darstellung der Verteilung der Laufzeit der Subroutinen innerhalb jeder Runde im Falle einer Ausführung von Approx.LZ77 mit allen Optimierungen 4.1. Postprocess deckt dabei die nötigen Prozesse zum Rundenübergang, wie das Spalten der Blöcke und der Extraktion der Faktoren, ab.

4.4 Auswertung

4.4.1 LZ77

Wie in Kapitel 3.1 beschrieben, zeigt der verwendete Algorithmus zur Generierung einer exakten LZ77-Faktorisierung ein lineares Verhalten bezüglich der Laufzeit. Dies wird in Abbildung 4.3 deutlich, wo die Laufzeitmessung von LZ77 auf verschiedenen Präfixen von proteins dargestellt ist. Die lineare Regression der Laufzeit deckt sich mit der tatsächlichen Laufzeitmessung. Auch in Bezug auf die Speichernutzung wird die theoretische Analyse bestätigt. In Abbildung 4.2 ist der Speicherverbrauch von LZ77 auf verschiedenen Präfixen von proteins aufgezeichnet. Der in Kapitel 3.1 beschriebene Speicherverbrauch von 12 Byte pro Eingabezeichen stellt auch für die Messwerte eine untere Schranke dar. Der zusätzliche Speicherverbrauch ist durch die Speicherung der Faktoren bestimmt, die in die Messung einfließen. Hiermit konnten wir die theoretische Analyse des Laufzeit- und Speicherverhaltens empirisch bestätigen.

4.4.2 Approx. LZ77

In 4.3 wird deutlich, dass Approx. LZ77 dem exakten LZ77-Algorithmus in der Laufzeit und der Faktorrate stets deutlich unterlegen ist. Wie bereits in Kapitel 3.2 beschrieben, liegt der Fokus von Approx. LZ77 auf der Reduktion des Speicherverbrauchs. In der theoretischen Abschätzung der Laufzeiten wurde bereits festgestellt, dass Approx. LZ77 eine schlechtere Laufzeitkomplexität als LZ77 aufweist. Im Rahmen des Speicherbedarfs zeigt sich, dass Approx. LZ77 in den meisten Fällen eine geringere Speichernutzung aufweist als LZ77. In Relation zur Eingabegröße verfügt Approx. LZ77 jedoch eine hohe Varianz in der Speichernutzung. Dies stellt eine Konsequenz der theoretischen Abschätzung der Speichernutzung dar, die insbesondere eine hohe Abhängigkeit von der Größe der Faktorfolge aufweist. In Abbildung 4.2 wird ebenfalls deutlich, dass selbst verschiedene Präfixe einer Eingabedatei unterschiedliche relative Speichernutzungen aufweisen. Dies lässt sich auf ein unterschiedliches Maß der Redundanz in den Eingaben zurückführen. Die Kompressionsrate CR^* von Approx. LZ77 ist in 4.3 ebenfalls aufgeführt. Es ist zu erkennen, dass die Abweichung der Kompressionsrate von LZ77 in den meisten Fällen geringer ausfällt als die Abweichung der Faktorrate. Dies ist auf die in 4.3.2 beschriebene Eigenart der Kodierung zurückzuführen.

4.4.3 Approx. LZ77 Optimierungen

Die praktischen Optimierungen aus Kapitel 3.4 wurden in der Evaluation von Approx. LZ77 und Approx. LZ77Par standardmäßig aktiviert. Um den Nutzen dieser Optimierungen zu verdeutlichen wurden in 4.5 und 4.6 die Auswirkungen auf die Laufzeiten der einzelnen Runden von Approx.LZ77 aufgezeichnet. Aufgrund der Optimierungen DynStart

und DynEnd ist sofort ersichtlich, dass der Algorithmus einen großen Anteil der sonst nötigen Runden auslässt. Konkret werden die ersten 12 und die letzten 2 Runden im Falle der Eingabe proteins ausgelassen. Dies bringt bereits eine signifikante Reduktion der Gesamtlaufzeit. Die Postprocess-Routine umfasst hier die Prozesse, die im Rundenübergang stattfinden, wie das Spalten der Blöcke und die explizite Extraktion der Faktoren. Weiterhin fällt auf, dass die InitTables- und Postprocess-Routine im optimierten Fall deutlich weniger Zeit in Anspruch nehmen. Dies lässt sich auf die Anwendung von PreMatching zurückführen, da Blöcke in InitTables vorgefiltert werden und so weniger Einfügeoperationen stattfinden, sowie im Rundenübergang die Spaltung der Blöcke durch vorberechnete RFPs beschleunigt wird. Dieser Effekt verfällt enstprechend nach der vorberechneten Runde, hier die 24.te Runde. Daneben zeigt sich die Wirkung von ScanSkip in den Ausfällen der ReferenceScan-Routine in den ersten vier und der letzten Runde. Die Kombination aus der Filterung von Blöcken durch PreMatching und der Eliminierung von Duplikaten in InitTables führt in beiden Fällen zu einer relativ kleinen RFPTable. Zudem zeigt der Vergleich der Faktorraten in 4.3, dass die Optimierungen nur eine geringe Abweichung der Faktorraten veursachen. In der Gesamtheit haben wir die Sinnhaftigkeit der Nutzung der Optimierungen in Approx.LZ77 bestätigt. Die hier beschriebenen Effekte sind aufgrund des identischen Programmablaufs auch auf Approx.LZ77Par übertragbar.

4.4.4 Approx. LZ77Par

In Bezug auf die Qualität der Kompression weist die Approx.LZ77Par keine Unterschiede zu Approx.LZ77 auf, was als Indiz für die Korrektheit der Implementierung interpretiert werden kann. Die Laufzeitmessung in 4.3 zeigt, dass Approx.LZ77Par mit 16 Threads eine deutlich bessere Laufzeit aufweist als Approx.LZ77. Es anzumerken, dass wir im Rahmen der parallelen Implementierung der InitTables-Routine die Verteilung der Blöcke auf die Threads durch eine binäre Maske realisiert haben. Als Konsequenz ist die Anzahl der genutzten Threads in der InitTables-Routine auf die größte Zweierpotenz beschränkt, die kleiner als die Anzahl der verfügbaren Threads ist. Die Verschlechterung der Laufzeit im Übergang von 8 auf 9 Threads ist auf diese Beschränkung zurückzuführen. Weiterhin zeigt die Laufzeitmessung in 4.4, dass die Laufzeit von Approx.LZ77Par mit wachsender Anzahl an Threads eine asymptotische Grenze erreicht. Analog dazu zeigt die Aufzeichnung des Speedups auch obere Schranke. Die beschriebenen Asymptoten in der Laufzeitmessung sind auf externe Faktoren zurückzuführen, wie der Bandbreite der Speicherzugriffe und Symptomen von False-Sharing (2.4.1). Dies folgt aus der Nutzung von Datenstrukturen für die RFPTable und RefTable, die auf einem kontinuierlichen Speicherbereich angelegt sind. Eine logische Abgrenzung der Speicherzugriffe kann daher nichtsdestotrotz auf der Ebene der Cache-Lines gegenseitige Invalidierungen verursachen [11]. Der Speicherverbrauch von Approx.LZ77Par ist in 4.2 ebenfalls aufgezeichnet. Es fällt auf, dass der Speicherverbrauch

von Approx.LZ77Par im Vergleich zu Approx.LZ77 in den meisten Fällen geringer ausfällt. Dies ist auf die Aufteilung der RFPTable auf mehrere Instanzen von Unordered-Dense-Map aus 4.2.2 zurückzuführen, die aufgrund ihrer inhärenten Struktur einen geringeren gesamten Speicherverbrauch aufweisen.

Kapitel 5

Fazit

5.1 Zusammenfassung und Einordnung

Im Rahmen dieser Arbeit haben wir die erste Phase des approximativen LZ77-Algorithmus [5], der in erster Linie auf die Reduktion seines Speicherbedarfs abzielt, auch in seiner Laufzeit optimiert. Hierbei spielte neben praktischen Optimierungen, insbesondere eine parallele Ausführung eine entscheidende Rolle. Die Parallelisierung des Algorithmus zeigte eine deutliche Beschleunigung der Laufzeit, ohne die Qualität der Kompression zu beeinträchtigen. Es ist jedoch zu beachten, dass wir mit einer oberen Schranke für den Grad der Parallelisierung konfrontiert wurden, die mit hoher Wahrscheinlichkeit durch die begrenzte Bandbreite des Speichersystems verursacht wurde. Weiterhin stellen die optionalen Optimierungen einen TradeOff zwischen Metriken des Algorithmus dar, welcher je nach Eingabe und der Anforderungen abgewägt werden muss.

5.2 Ideen für die Zukunft

Die beschriebenen Schwächen bzw. Grenzen des implementierten Algorithmus bieten Raum für zukünftige Verbesserungen. So können weitergehende Optimierungen der Parallelisierung, die auf einer hardwarenahen Steuerung der Speicherzugriffe basieren, die Grenzen der Bandbreite des Speichersystems besser ausnutzen. Im Laufe der Ausarbeitung dieses Algorithmus wurden alternative Techniken und Bibliotheken getestet, die jedoch zum Zeitpunkt der Finalisierung dieser Arbeit nicht zufriedenstellend optimiert waren. Beispielsweise könnte die Verwendung eines Bloom-Filters [3] das Volumen der Suchoperationen von Referenzen reduzieren. Weiterhin wäre eine Paralleliserung der weiteren zwei Phasen des approximativen LZ77-Kompressionsalgorithmus [5] eine sinnvolle Erweiterung.

Anhang A

Weitere Informationen

A.1 Alternative Eingabedaten

Im Folgenden sind die vollständigen Messungen für die restlichen Eingabedaten aus 4.2. Während die absoluten Werte der Laufzeit und des Speicherverbrauchs für die verschiedenen Eingabedaten variieren, zeigen die Kurven der Laufzeitmessungen eine ähnliche Tendenz und bestätigen die Aussagen der Auswertung.

Abbildung A.1: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von sources. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

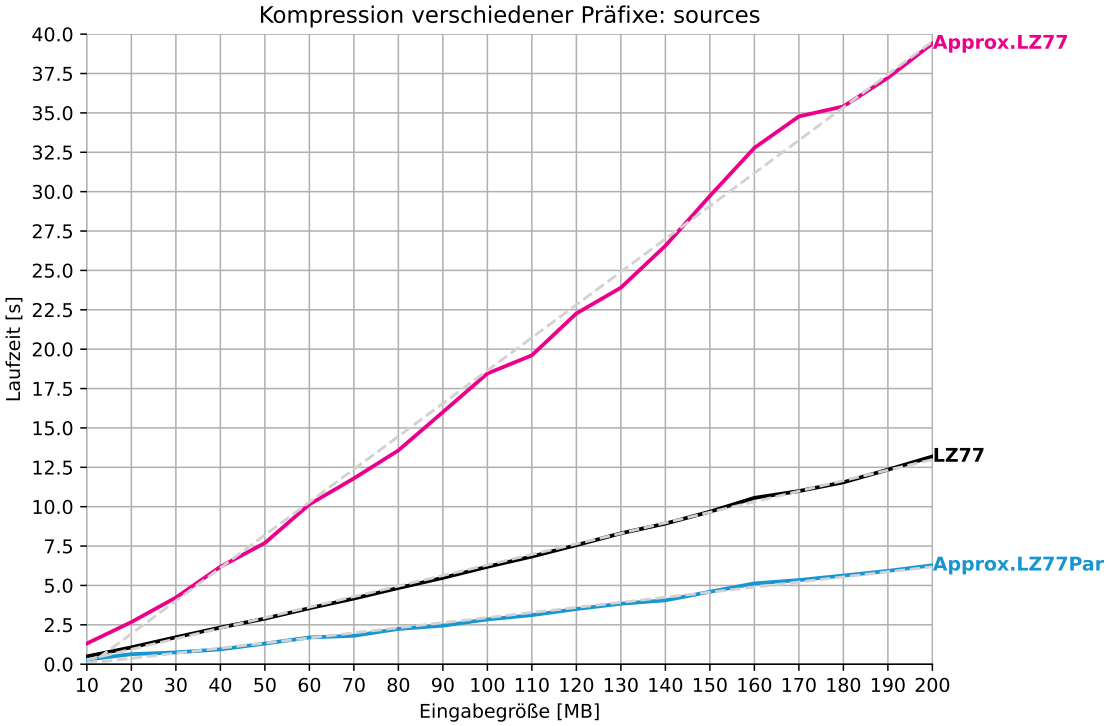


Abbildung A.2: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für sources

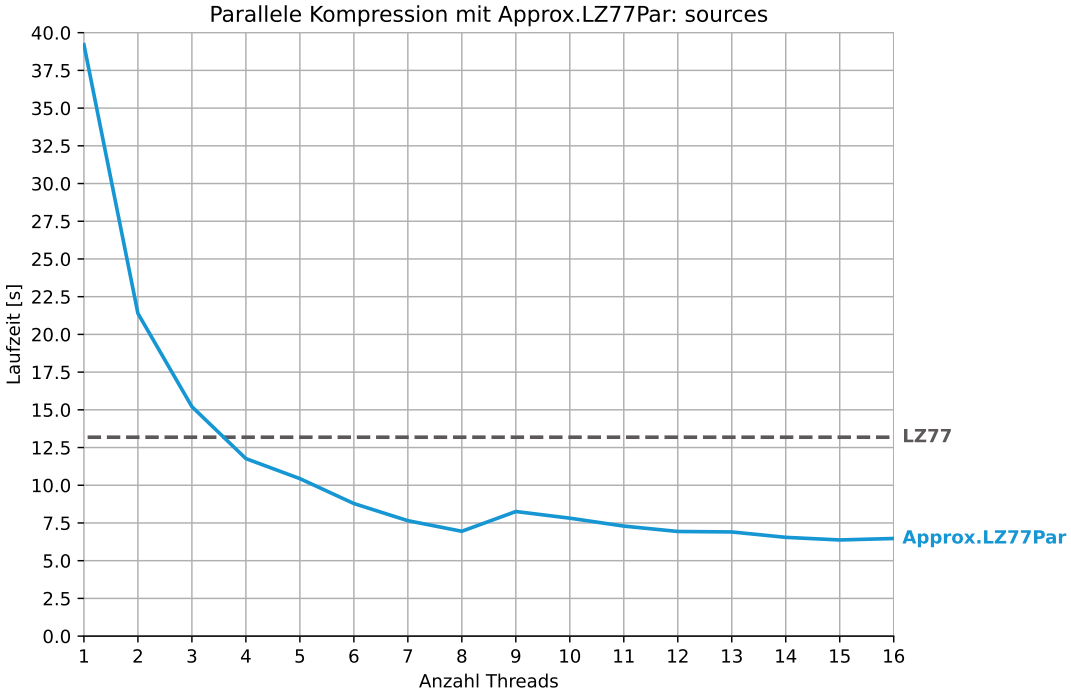


Abbildung A.3: Speicherverbrauch von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von sources. Aufgezeichnet wurde das Verhältnis von allokiertem Speicher zur Eingabegröße.

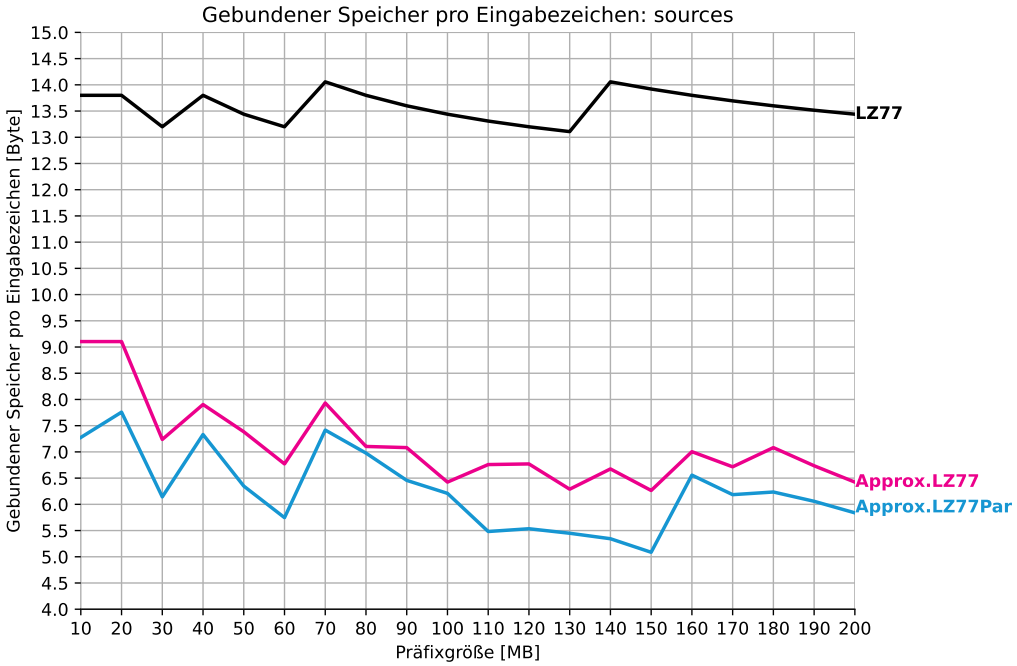


Abbildung A.4: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von english. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

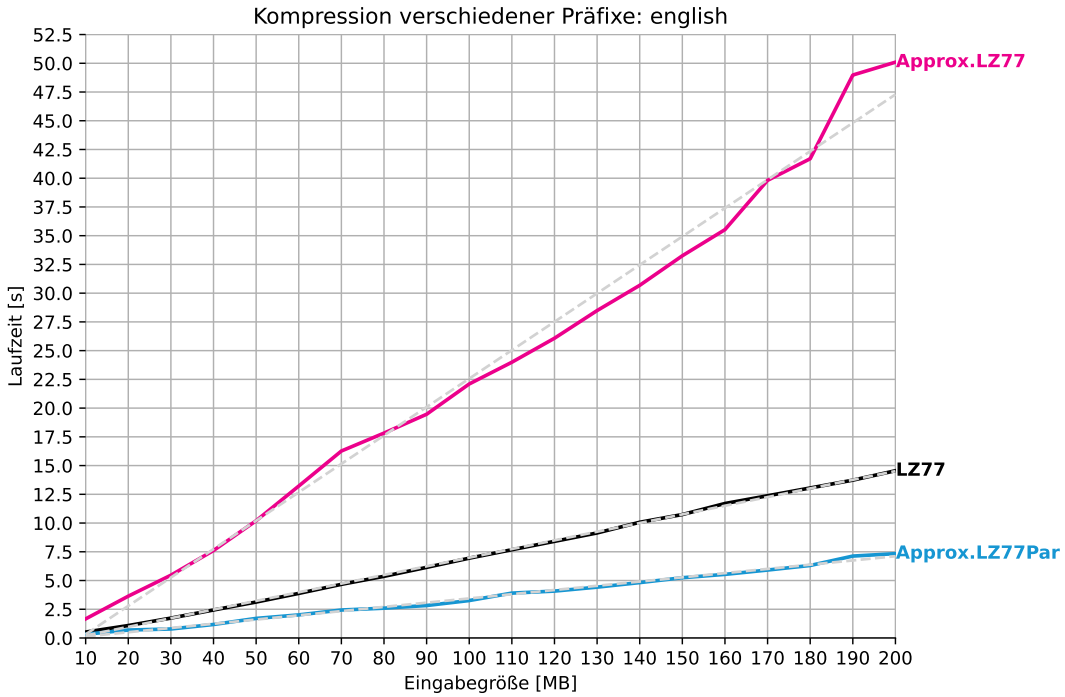


Abbildung A.5: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für english

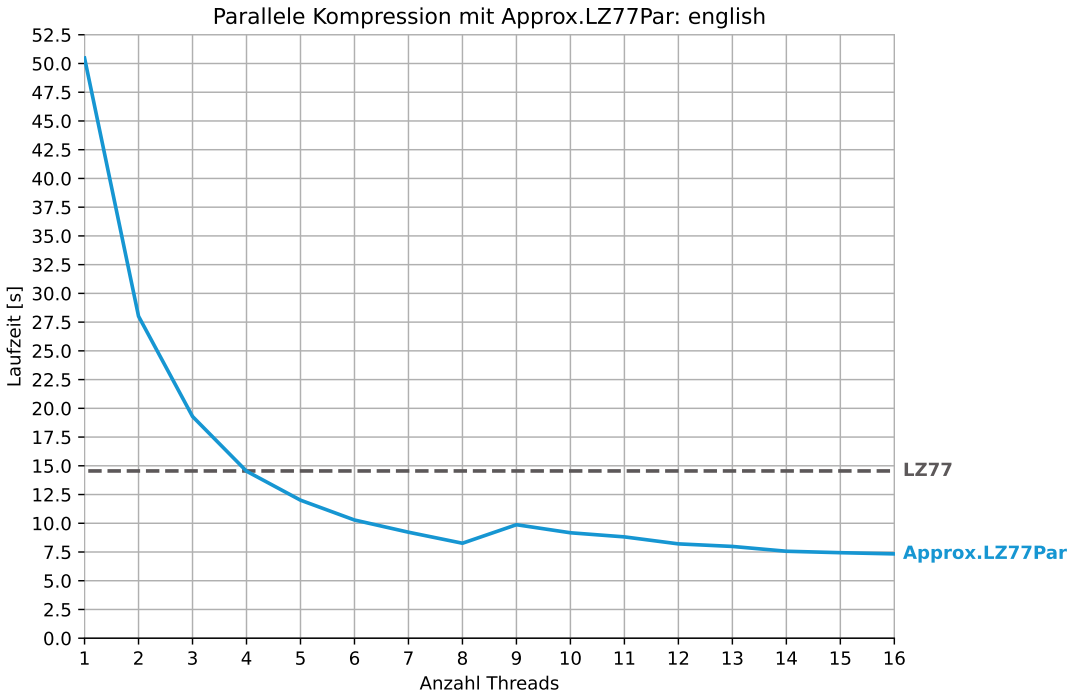


Abbildung A.6: Speicherverbrauch von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von english. Aufgezeichnet wurde das Verhältnis von allokiertem Speicher zur Eingabegröße.

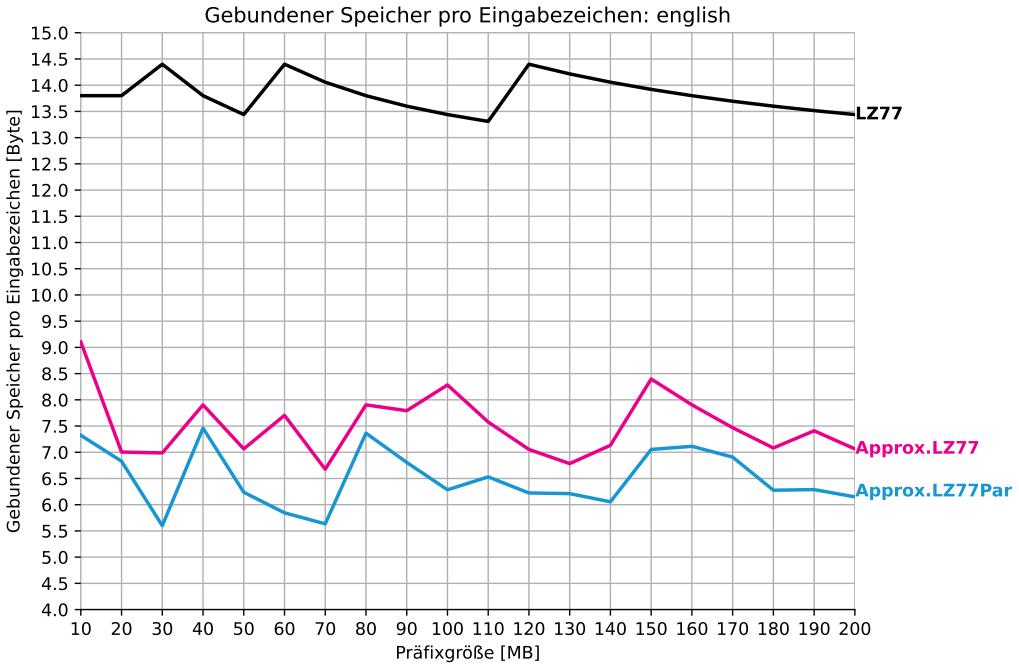


Abbildung A.7: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von dna. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

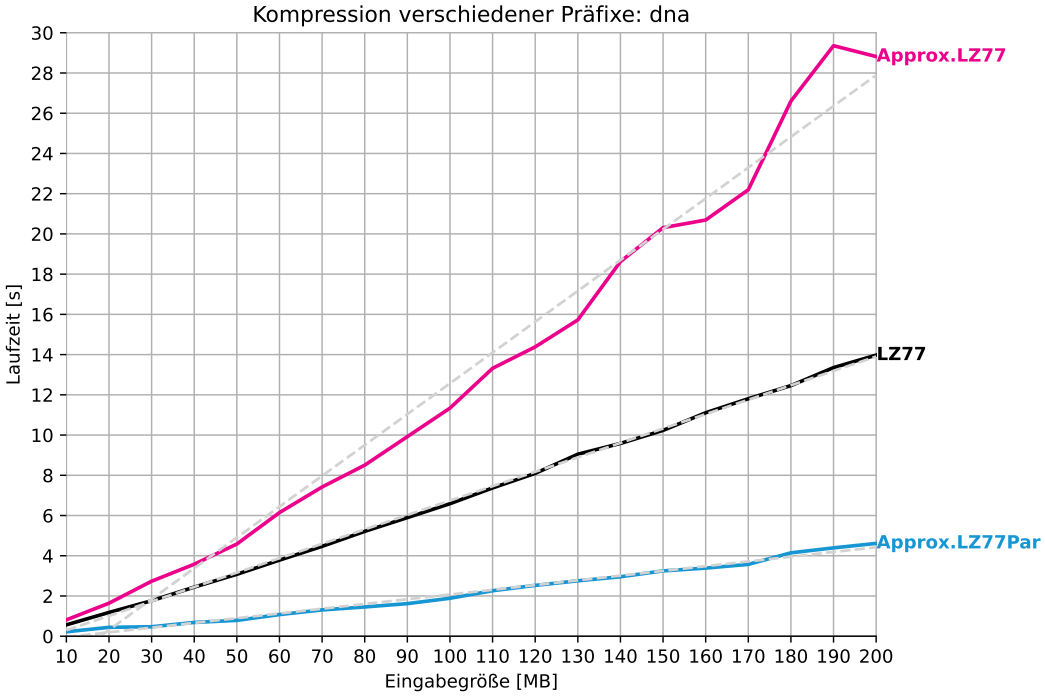


Abbildung A.8: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für dna

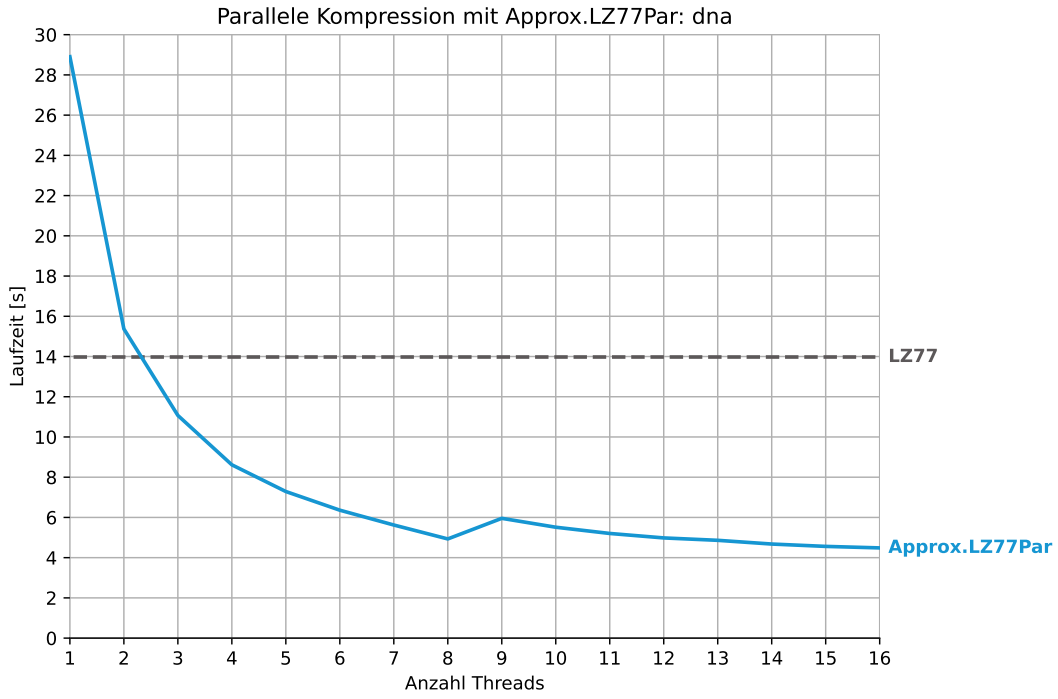


Abbildung A.9: Speicherverbrauch von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von dna. Aufgezeichnet wurde das Verhältnis von allokiertem Speicher zur Eingabegröße.

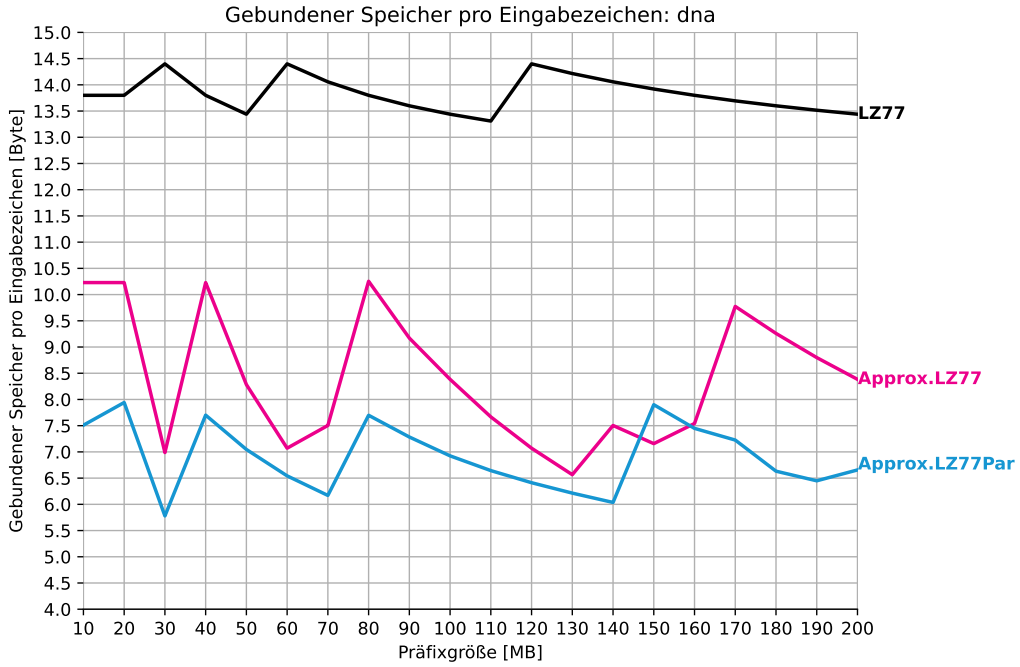


Abbildung A.10: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von xml. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

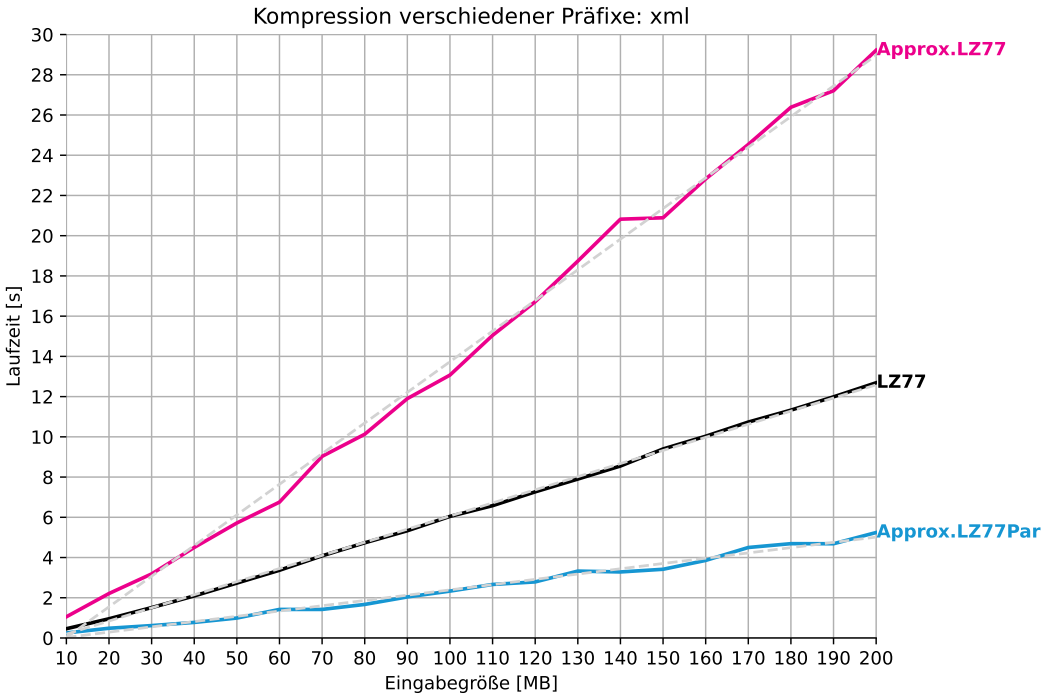


Abbildung A.11: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für xml

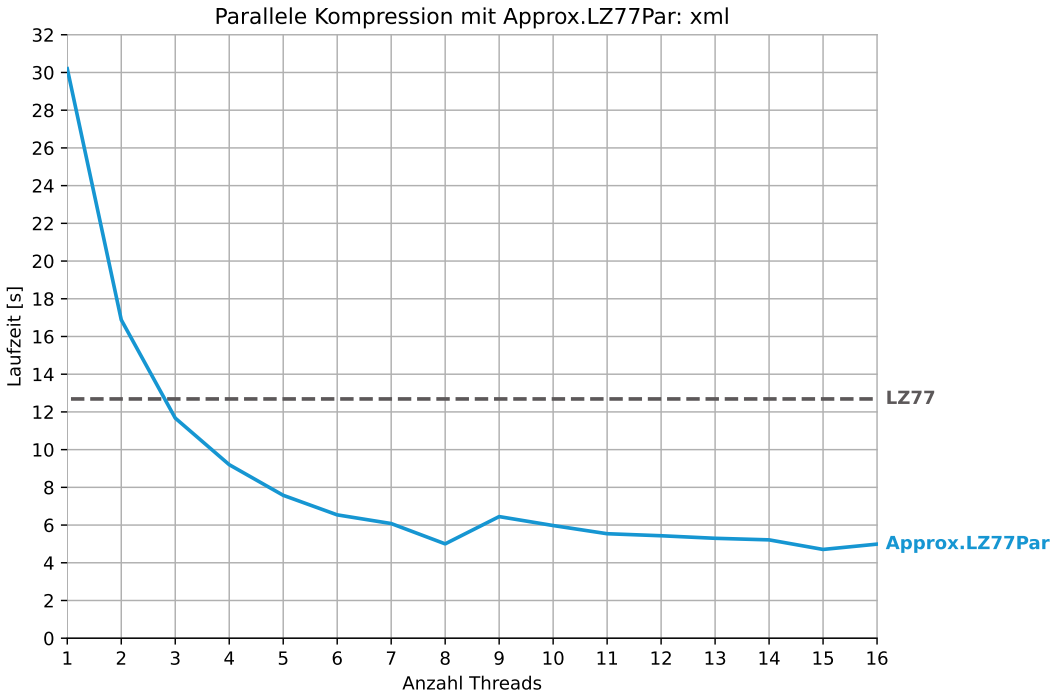
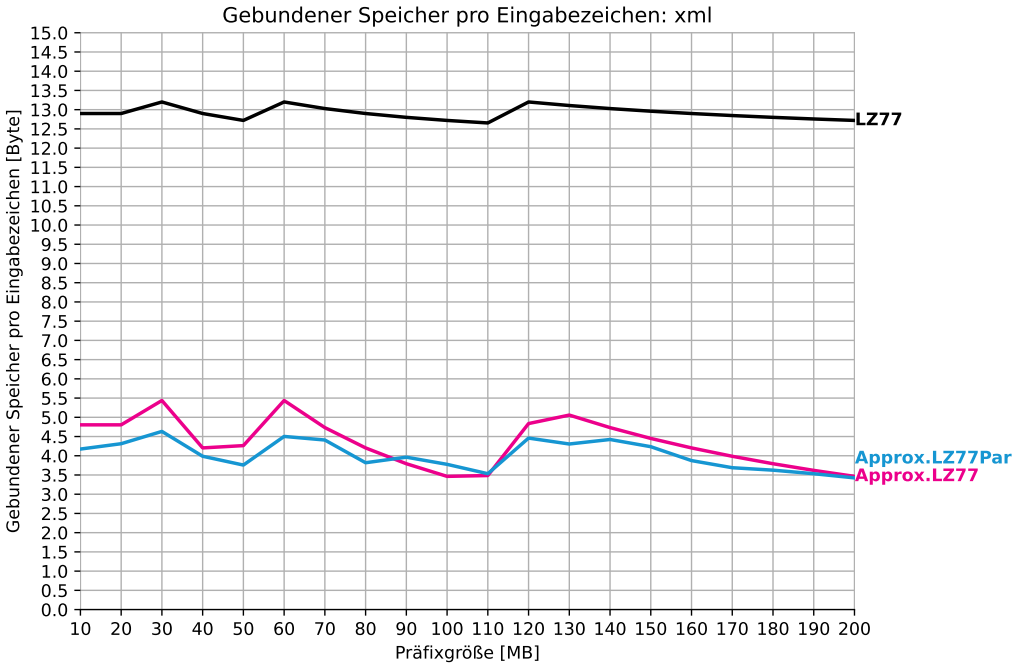


Abbildung A.12: Speicherverbrauch von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von xml. Aufgezeichnet wurde das Verhältnis von allokiertem Speicher zur Eingabegröße.



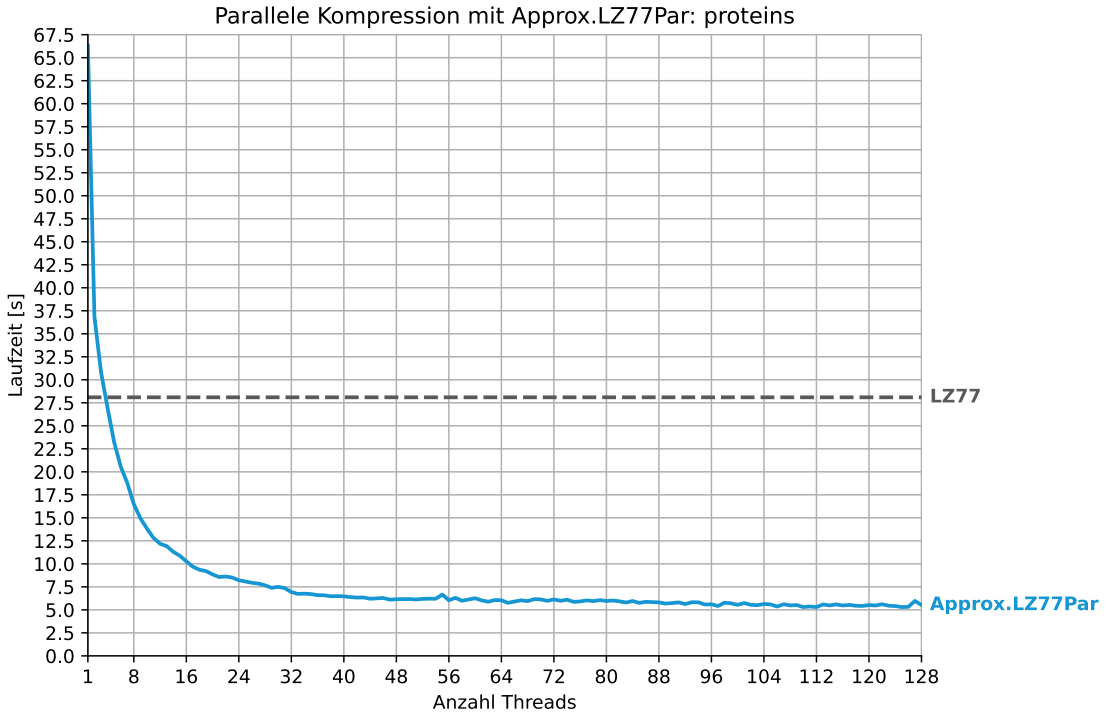
A.2 Alternative Testumgebung

Für die folgenden Messwerte wurden die Algorithmen auf einem Rechner mit einem AMD EYPC 7452 32-Core Prozessor mit 128 nutzbaren Threads ausgeführt. Die Einstellungen der Algorithmen, die in 4.1 für die optimierte Ausführung etabliert wurden, wurden beibehalten.

Tabelle A.1: Messwerte der Algorithmen auf verschiedenen Eingabedateien. Die Messungen wurden auf einem alternativen System mit 128 nutzbaren Threads durchgeführt.

Eingabe	Algorithmus	Laufzeit[s]	Speicher	FR	CR*
proteins	LZ77	30.65	14.88	9.95%	70.92%
	Approx.LZ77	64.46	9.94	15.34%	63.95%
	Approx.LZ77Par	4.75	9.20	15.34%	63.95%
sources	LZ77	28.38	13.44	5.50%	39.20%
	Approx.LZ77	62.35	6.42	10.05%	40.14%
	Approx.LZ77Par	3.86	5.51	10.05%	40.14%
english	LZ77	33.59	13.44	6.66%	47.45%
	Approx.LZ77	81.21	7.06	10.42%	43.39%
	Approx.LZ77Par	4.30	5.95	10.42%	43.39%
dna	LZ77	28.66	13.44	6.66%	47.46%
	Approx.LZ77	46.06	8.38	10.71%	45.53%
	Approx.LZ77Par	3.52	6.13	10.71%	45.53%
xml	LZ77	27.89	12.72	3.35%	23.89%
	Approx.LZ77	49.88	3.46	6.62%	26.78%
	Approx.LZ77Par	2.91	3.28	6.62%	26.78%

Abbildung A.13: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für proteins. Die Messungen wurden auf einem alternativen System mit 128 nutzbaren Threads durchgeführt.



Literaturverzeichnis

[1] ARB, OpenMP Architecture Review B.: *OpenMP Application Program Interface*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.2.pdf, 2021. – Version 5.2

[2] Bingmann: *Malloc Count*. https://github.com/ByteHamster-etc/malloc_count.git, 2024. – Accessed: 2024-08-15

[3] Bloom, Burton H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. In: *Commun. ACM* 13 (1970), Nr. 7, 422–426. http://dx.doi.org/10.1145/362686.362692. – DOI 10.1145/362686.362692

[4] Crochemore, Maxime ; Ilie, Lucian: Computing Longest Previous Factor in linear time and applications. In: *Inf. Process. Lett.* 106 (2008), Nr. 2, 75–80. http://dx.doi.org/10.1016/J.IPL.2007.10.006. – DOI 10.1016/J.IPL.2007.10.006

[5] Fischer, Johannes ; Gagie, Travis ; Gawrychowski, Pawel ; Kociumaka, Tomasz: Approximating LZ77 via Small-Space Multiple-Pattern Matching. In: *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, 2015, 533–544

[6] Grebnov, Ilya: *libsais*. https://github.com/IlyaGrebnov/libsais.git, 2024. – Accessed: 2024-08-15

[7] Intel: *Intel oneAPI Threading Building Blocks*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html, 2024. – Accessed: 2024-08-15

[8] JáJá, Joseph F.: *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. – 9–11 S. – ISBN 0–201–54856–9

[9] Karp, Richard M. ; Rabin, Michael O.: Efficient Randomized Pattern-Matching Algorithms. In: *IBM J. Res. Dev.* 31 (1987), Nr. 2, 249–260. http://dx.doi.org/10.1147/RD.312.0249. – DOI 10.1147/RD.312.0249

[10] Leitner-Ankerl, Martinus: *Unordered Dense Hash Map*. https://github.com/martinus/unordered_dense.git, 2024. – Accessed: 2024-08-15

[11] McCool, M. ; Reinders, J. ; Robison, A.: *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012. – 58 S. https://books.google.de/books?id=2hYqeoO8t8IC. – ISBN 9780123914439

[12] Navarro, Gonzalo: *The Pizza& Chili Corpus*. https://pizzachili.dcc.uchile.cl/texts.html, 2024. – Accessed: 2024-08-15

[13] Nong, Ge ; Zhang, Sen ; Chan, Wai H.: Linear Suffix Array Construction by Almost Pure Induced-Sorting. In: Storer, James A. (Hrsg.) ; Marcellin, Michael W. (Hrsg.): *2009 Data Compression Conference (DCC 2009), 16-18 March 2009, Snowbird, UT, USA*, IEEE Computer Society, 2009, 193–202

[14] Ohlebusch, E.: *Lempel-Ziv Factorization: LZ77 without Window*. https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Lehre/SS14/Datenkompression/ScriptLZ.pdf. Version: 2016

[15] Palanga, Etienne: *Sharded Map*. https://github.com/Skadic/sharded_map.git, 2024. – Accessed: 2024-08-15

[16] Sanders, Peter ; Mehlhorn, Kurt ; Dietzfelbinger, Martin ; Dementiev, Roman: *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. http://dx.doi.org/10.1007/978-3-030-25209-0. http://dx.doi.org/10.1007/978-3-030-25209-0. – ISBN 978–3–030–25208–3

[17] STL: *STL Execution Policy*. https://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/execution_policy_tag&oldid=150256, 2024. – Accessed: 2024-08-15

[18] Storer, James A. ; Szymanski, Thomas G.: Data compression via textual substitution. In: *J. ACM* 29 (1982), Nr. 4, 928–951. http://dx.doi.org/10.1145/322344.322346. – DOI 10.1145/322344.322346

[19] Ziv, Jacob ; Lempel, Abraham: A universal algorithm for sequential data compression. In: *IEEE Trans. Inf. Theory* 23 (1977), Nr. 3, 337–343. http://dx.doi.org/10.1109/TIT.1977.1055714. – DOI 10.1109/TIT.1977.1055714

Eidesstattliche Versicherung

(Affidavit)

Sivarajah, Gajann

Name, Vorname
(surname, first name)

168246

Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Parallelisierung einer speichereffizienten Approximation der LZ77-Faktorisierung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Witten, 29.08.2024

Ort, Datum
(place, date)

Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Witten 29.08.2024

Ort, Datum
(place, date)

Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**