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 zugeordnet werden, wobei diese sich in Statistiken, wie der Laufzeit, der Speicheranforderung oder Kompressionrate unterscheiden [14]. In [11] wird eine Variante der LZ77-Faktorisierung beschrieben, die über drei Phasen eine 2-Approximation einer exakten LZ77-Faktorisierung [10] 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 Potential zur 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 konfliktfrei auf Datenstrukturen zugreifen und eine korrekte Ausgabe liefern können. Im Rahmen der praktischen Evaluation der beschriebenen Konzepte wird eine Implementierung in C++ herangezogen. Die Parallelisierung wird hauptsächlich über OpenMP-Instruktionen [5] realisiert. Im Rahmen dieser Arbeit wird insbesondere die parallele Generierung einer Suchtabelle von 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[10] 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 [6].

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 $S_1 + S_2$ die Konkatenation der beiden Teilfolgen.

2.2 Ausgabe → Faktorisierung

Ein charakteristisches Merkmal der Familie der Lempel-Ziv-Kompressionsverfahren ist die Repräsentation der Ausgabe in Form einer Faktorisierung. Für eine Eingabe $S = e_1...e_n$ wird eine Faktorisierung $F = f_1...f_z$ mit $z \leq n$ derart erzeugt, dass die Eingabe S durch die Faktorisierung in eine equivalente Folge von nichtleeren Teilfolgen zerlegt wird. Dabei ist jeder Faktor f_i mit $1 \leq i \leq z$ als nichtleerer Präfix von $S[|f_1...f_i - 1| + 1..n]$ definiert, der bereits in $S[1..|f_1...f_i|]$ vorkommt, oder als einzelnes Zeichen. Die im Folgenden betrachteten Algorithmen können speziell der Klasse der LZ77-Kompressionsverfahren zugeordnet werden, dessen Faktoren im Schema des Lempel-Ziv-Storer-Szymanski [13] repräsentiert werden sollen.

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

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 2.1 beschreibt eine mögliche Implementierung der Dekompression für eine Faktorisierung $F = f_1...f_z$ zu der Eingabe $S = e_1...e_n$. Der beschriebene Algorithmus iteriert durch alle Faktoren und fügt die referenzierten Zeichen einzeln in die Ausgabe 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)[11] durch-

Algorithmus 2.1 DECOMP$_{LZ77}$

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

```
S ← ∅
for i = 1 to z do
  (len, ref) ← f_i
  if len = 0 then
    S ← S + ref
  else
    for j = 0 to len − 1 do
      S ← S + S[ref + j]
    end for
  end if
end for
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[11]. Insbesondere kann die Wahrscheinlichkeit durch die passende Wahl von p und b minimiert werden.

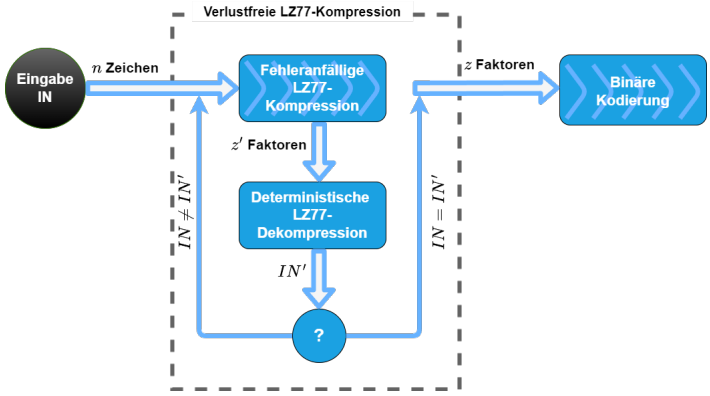
Rabin-Karp-Fingerprints erlauben verschiedene Operationen auf Zeichenfolgen, die im Rahmen der approximativen LZ77-Faktorisierung effizient genutzt werden können. Zum Einen kann ein beschränktes Fenster $S_W = S(j..j+w)$ der Länge $w < n$ leicht verschoben werden. Sei $RFP(S_W)$ der Fingerabdruck des Fensters, so kann der Fingerprint 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. Desweiteren seien zwei Teilfolgen S_1 und S_2 der gleichen Länge n gegeben. Der RFP der Konkatenation,$S_1 + S_2$, der beiden Teilfolgen kann durch den Ausdruck,

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

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



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 Parametern wiederholt werden kann, können wir einen verlustfreien Las-Vegas-Algorithmus konstruieren.

In Abbildung 2.1 wird eine Regelsteuerung illustriert. Der Algorithmus wird solange wiederholt, bis eine korrekte Faktorisierung erzeugt wurde. Dass die Anzahl der Wiederholungen beschränkt ist, werden wir in der Analyse des Approximationsalgorithmus und der praktischen Evaluation zeigen.

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, oder 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 $K_{OUT}^{-1} : \{0,1\}^* \rightarrow F$ 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

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

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 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.[12] 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 benötigt. 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 aufgrund des Amdahl'schen Gesetzes[1] 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 [10] beschrieben ist. Der Algorithmus erzeugt zunächst ein Suffix-Array, welches allen Suffixen der Eingabe eine lexikographische Ordnung zuweist. Mithilfe der lexikographischen Ordnung können Kandidaten für Referenzen effizient gefunden werden. Hierfür werden mit Hilfe des Suffix-Arrays zwei Arrays, das Next Smaller Value(NSV) und das Previous Smaller Value(PSV) erzeugt. Sei die aktuelle Position in der Eingabe k, so muss aufgrund von positionellen und lexikographischen Einschränkungen die Position ref der längsten vorherigen Referenz entweder $NSV[k]$ oder $PSV[k]$ sein. 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. Das Ergebnis dieser Berechnung bestimmt den Faktor (len, ref), welcher in der Eingabe an Position k beginnt. Der Algorithmus terminiert, wenn die gesamte Eingabe abgearbeitet wurde.

3.1.2 Theoretisches Laufzeit- und Speicherverhalten

Die Berechnung des Suffix-Arrays und die folgende Berechnung der NSV- und PSV-Arrays können mithilfe von Algorithmen aus der Literatur(siehe [10]) in $O(n)$ Laufzeit durch-

Algorithmus 3.1 COMP_{LZ77}

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

$SA \leftarrow SuffixArray(S)$
$(NSV, PSV) \leftarrow (NSVArray(S, SA), PSVArray(S, SA))$
$F \leftarrow \emptyset$
$k \leftarrow 1$
while $k \leq n$ **do**
  $(len, ref) \leftarrow (0, 0)$
  $l_{nsv} \leftarrow LCP(S(NSV[k]..n), S(k..n))$
  $l_{psv} \leftarrow LCP(S(PSV[k]..n), S(k..n))$
  if $l_{nsv} > l_{psv}$ **then**
    $(len, ref) \leftarrow (l_{nsv}, NSV[k])$
  else if $l_{nsv} < l_{psv}$ **then**
    $(len, ref) \leftarrow (l_{psv}, PSV[k])$
  else
    $(len, ref) \leftarrow (0, S[k])$
  end if
  $F \leftarrow F + (len, ref)$
  $k \leftarrow k + len + 1$
end while
return F

geführt werden. 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 sich die Größe des Suffix-Arrays und der NSV- und PSV-Arrays linear zur Eingabelänge verhalten. Es sollte jedoch angemerkt werden, dass die Linearität des Speicherbedarfs einen hohen konstanten Faktor hat und unabhängig von der Beschaffenheit der Eingabe und der Anzahl der Faktoren ist.

3.2 Approximation der LZ77-Faktorisierung(Approx. LZ77)

3.2.1 Konzept

Im Rahmen dieser Arbeit wird die erste Phase einer speichereffizienten Approximation des LZ77-Algorithmus betrachtet, welche auch eine verlustfreie Kompression darstellt. Wie in [11] beschrieben, kann die Kombination aller drei Phasen des Algorithmus eine

2-Approximation bezüglich der Faktorrate ermöglichen. Die resultierenden Faktoren entsprechen jedoch ebenfalls dem LZSS-Schema2.1, sodass eine verlustfreie Dekompression mit 2.1 möglich ist. Im weiteren Verlauf werden wir die sequenzielle erste Phase des Algorithmus als Approx.LZ77 bezeichnen. 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. Die Approximation wird in mehreren Runden durchgeführt, wobei in jeder Runde die noch unverarbeitete Eingabe in Blöcke gleicher Größe eingeteilt wird. Innerhalb einer Runde werden für Zeichenfolgen, die durch die jeweiligen Blöcke repräsentiert werden, Referenzen, also vorherige Vorkommen, gesucht. Im Erfolgsfall wird ein entsprechender Faktor extrahiert und der Block gilt als markiert. Die verbleibenden Blöcke werden im Übergang zur nächsten 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. Die Zeichenfolgen der Blöcke, die in der letzten Runde nicht markiert wurden, werden als einzelne Zeichen interpretiert und faktorisiert. Der Ablauf des Algorithmus wird in 3.2 dargestellt. Initial wird die Eingabe in 2^r Blöcke gleicher Größe eingeteilt, wobei $r \in \mathbb{N}$ mit $1 \leq r \leq log_2(|S|)$ die initiale Runde beschreibt. Daraufhin wird eine Schleife über alle Runden gestartet, wobei in jeder Iteration die aktuelle Menge der Blöcke auf Referenzen untersucht und ggf. markiert werden. Die Menge der markierten Blöcke,die auch zur Menge der Faktoren F hinzugefügt werden, wird in der Routine ProcessRound bestimmt. Nachdem die Menge der Blöcke um die markierten Blöcke bereinigt wurde, werden die verbleibenden Blöcke halbiert und die nächste Runde initiiert.

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 einzigartigen Rabin-Karp-Fingerprint(RFP), der über die Menge aller Blöcke erzeugt wird, den linkesten Block zu, welcher diesen Wert aufweist. In diesem Rahmen vernachlässigen wir explizit Kollisionfälle 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 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

Algorithmus 3.2 $\text{COMP}_{ApproxLZ77}$

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

1: $F \leftarrow \emptyset$
2: $r \leftarrow 1$
3: $Blocks[1..2^r] \leftarrow InitBlocks(S, 2^r)$ // Split S into 2^r equal blocks
4: **while** $r \leq log_2(|S|)$ **do**
5: $\quad markedBlocks[1..z_r] \leftarrow ProcessRound(r, S, Blocks, F)$
6: $\quad Blocks \leftarrow NextNodes(Blocks \setminus markedBlock[1..z_r])$ // Halve unmarked blocks
7: $\quad r \leftarrow r + 1$
8: **end while**
9: **return** F

wird, wenn der RFP noch nicht vorhanden ist. Als Konsequenz wird für jeden RFP, der eine einzigartige Zeichenfolge repräsentiert, der entsprechende linkeste Block gespeichert. Die RefTable wird analog initialisiert. Falls der RFP eines Blocks bereits in der RFPTable vorhanden ist, so kann dem Block ein vorheriges Vorkommen zugeordnet werden. Andernfalls ist der Block das linkeste Vorkommen seiner Zeichenfolge und dessen Position wird entsprechend gespeichert. Es ist zu betonen, dass die InitTables-Routine durch ihre Aktualisierungen der RefTable bereits markierte Blöcke bzw. Faktoren implizit erzeugt.

Im zweiten Schritt werden zusätzliche Referenzen innerhalb der gesamten Eingabe S gesucht. Dazu wird der RFP eines Fensters der Größe $\frac{|S|}{2^r}$ über die Eingabe berechnet und mit den Einträgen der RFPTable verglichen. Im Falle eines Treffers wird die Position des Fensters in der RefTable eingetragen, falls er den vorherigen Wert unterbietet. Im Rahmen des Scans wird ein Rolling-Hash2.4 verwendet, um die RFPs der Fenster effizient um eine Position zu verschieben. Schließlich wird im dritten Schritt die RefTable genutzt, um die implizit markierten Blöcke zu extrahieren. Ein Block wird markiert, wenn die Position des linkesten Vorkommens seiner Zeichenfolge in der RefTable kleiner als die aktuelle Position des Blocks ist. Die markierten Blöcke werden in die Menge der Faktoren F eingefügt, wobei die Reihenfolge der Faktoren entweder während des Einfügens oder durch eine nachfolgende Sortierung sichergestellt werden muss.

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 innerhalb der Blöcke 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

Algorithmus 3.3 ProcessRound

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** $markedBlocks$

Algorithmus 3.4 InitTables

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** $RFPTable, RefTable$

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 $O(n \cdot log_2(n))$ für die Generierung der approximativen LZ77-Faktorisierung. Der Speicherbedarf des Algorithmus wird durch die Größe der RFPTable und der RefTable bestimmt, die wiederum durch die Anzahl der Blöcke bestimmt wird. 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 in der Runde übersteigt damit nie die Anzahl der endgültigen Faktorfolge. Der Speicherbedarf kann damit konservativ auf $O(z)$ abgeschätzt werden.

Algorithmus 3.5 ReferenceScan

Eingabe: $S, Blocks, RFPTable, RefTable$

blockSize $\leftarrow \frac{|S|}{|Blocks|}$
$RFP \leftarrow RFP(S[1..blockSize])$
for $i \leftarrow 1$ **to** $|S| - blockSize$ **do**
  if $RFPinRFPTable$ **and** $i < RefTable[RFPTable[RFP]]$ **then**
    $RefTable[RFPTable[RFP]] \leftarrow i$
  end if
  $RFP \leftarrow RFP(S[i+1..i+blockSize])$ // Rolling Hash
end for

3.3 Parallelisierung von Approx. LZ77(Approx. LZ77Par)

3.3.1 Konzept

Im Folgenden beschreiben wir das verwendete Schema zur Parallelisierung des Approx. LZ77-Algorithmus. Die Parallelisierung zielt auf eine schrittweise Bearbeitung der Eingabe, die jeweils auf die verschiedenen Prozessoren bzw. Threads verteilt werden. In 3.1 wird die

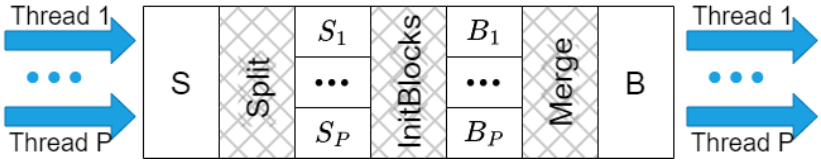


Abbildung 3.1: Parallele Generierung der initialen Blöcke

parallele Generierung der initialen Blöcke dargestellt. Hierbei wird die Eingabe S in P Teile aufgebrochen, sodass die Routine InitBlocks auf jedem Prozessor die zugehörigen Blöcke erzeugen kann. Da die chronologische Ordnung erhalten bleibt, können die erzeugten Blöcke ohne zusätzlichen Aufwand kombiniert werden.

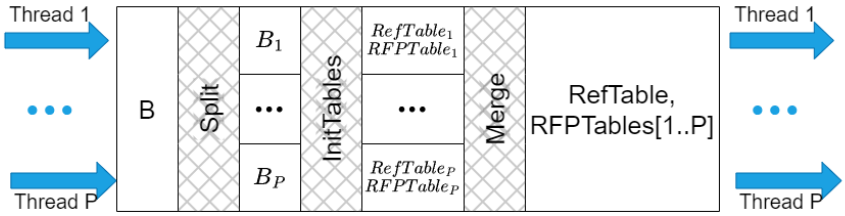


Abbildung 3.2: Parallele Initialisierung der RFP-Tabelle und der Referenztabelle

In 3.2 wird die parallele Initialisierung der RFP-Tabelle und der Referenztabelle dargestellt. Die 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 Aufteilungsschlüssel eine Überlappung der Zugriffe vermieden. Das Ergebnis ist eine konsistente RefTablee und eine einer P-elementigen Menge von RFPTable, die jeweils eine klar definierte Teilmenge aller RFPs enthalten.

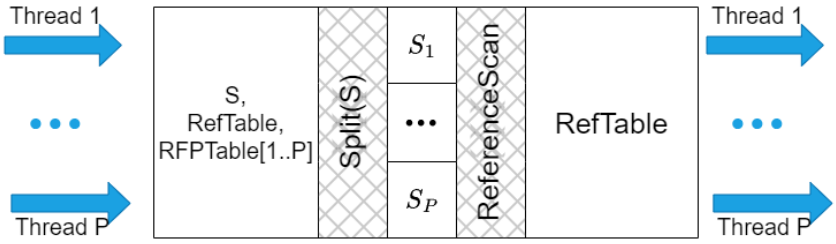


Abbildung 3.3: Paralleler Scan der Eingabe nach zusätzlichen Referenzen

In 3.3 wird der parallele Scan der Eingabe S nach zusätzlichen Referenzen dargestellt. Die Eingabe S wird in P Teile aufgeteilt. Auf jede Teilmenge 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 Arbeit und wird daher nicht weiter betrachtet. Eine mögliche Implementierung ist in [12] 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 $O(\frac{n \log n}{P})$ erreicht werden, wobei P die Anzahl der Prozessoren ist. 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 Kodierung K für die Übersetzung der erzeugten Faktorenfolge F gegeben. Der Wert,

$$Min_{Bin}^{Ref} = min\{|K(f)||f \in F, f \text{ ist Referenz}\} \tag{3.1}$$

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

$$Max_{Bin}^{Lit} = max\{|K(f)||f \in F, f \text{ ist Zeichen}\} \tag{3.2}$$

die maximale Anzahl an Bits, die für die Kodierung eines einzelnen 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}:

$$\begin{aligned} |K(f_{ref})| &\geq Min_{Bin}^{Ref} \\ &\geq |f_{ref}| \cdot Max_{Bin}^{Lit} \\ &\geq \sum_{i=1}^{|f_{ref}|} |K((0, S_{ref}(i)))|. \end{aligned} \tag{3.3}$$

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 berechnet werden. Gibt uns die Kodierung eine Grenzlänge l_{min}^{ref} vor, so kann der Algorithmus in Runde $r = \lceil log_2|S| - log_2 l_{min}^{ref} \rceil$ 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 sinkender Kompressionsrate.

$$CR_{DynEnd}^{Approx.LZ77} \leq CR^{Approx.LZ77} \tag{3.4}$$

$$FR_{DynEnd}^{Approx.LZ77} \geq FR^{Approx.LZ77} \tag{3.5}$$

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 Menge der Blöcke werde 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 Blockhalbierung 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 insbesonere auch auf die markierten Blöcke anwenden, sodass die folgende Beziehung hergeleitet werden kann:

$$|B^{init2}_{marked}| \geq 2^{r_{init2}-r_{init1}} \cdot |B^{init1}_{marked}|. \tag{3.6}$$

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,

$$longestChain(B^{init2}_{marked}) < 2^{r_{init2}-r_{init1}} \Rightarrow B^{init1}_{marked} = \emptyset, \tag{3.7}$$

wobei $LongestChain(B^{init2}_{marked})$ die längste 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_{Start} derart, dass vorherige Runden garantiert keine markierten Blöcke erzeugen und damit ausgelassen werden können. Der Wert r_{Start} ergibt sich wie folgt,

$$r_{Start} = r_{init} - \begin{cases} -1, & \text{falls } longestChain(B^{init}_{marked}) = 0 \\ \lfloor log_2 longestChain(B^{init}_{marked}) \rfloor, & \text{sonst} \end{cases} \tag{3.8}$$

In Abhängigkeit von der Beschaffenheiit der Eingabe, können 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 weiteren 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 $B^{marked}_{prematch}$. Weiterhin speichern wir uns den RFP aller Blöcke, die im Rahmen der Runde erzeugt werden. 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 entsprechend 2.5 in eine konstante Operation auf der Basis des RFP übersetzt werden. Gegeben sei eine Runde r_m mit $1 \leq r_m \leq log_2|S|$. 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, die die gleiche Zeichenfolge repräsentieren. So ergibt sich für den zugehörigen RFP,

$$RFP(b) = RFP(b_1) \oplus RFP(b_2) \oplus ... \oplus RFP(b_{2^{r_{prematch}-r_m}}) \tag{3.9}$$

, wobei jede Operation in konstanter Zeit durchgeführt werden kann. Die Anzahl der Rechenschritte für die Berechnung des RFP eines Blockes hängt nun nicht mehr von der Länge der repräsentierten Zeichenfolge ab, sondern der Rundendistanz zur vorberechneten Runde. 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 reduzieren. Ein Block $b \in B_m$ kann nur dann markiert werden, wenn die equivalente Sequenz von Blocken $(b_1, b_2, ..., b_{2^{r_{prematch}-r_m}}) \in B_{prematch}$ markiert ist. Die Umkehrung dieser Aussage liefert einen Filter für alle vorherigen Runden. Die zusätzlich gespeicherten Daten erhöhen den Speicherbedarf des Algorithmus. Falls die vorberechnete Runde auf den Wert $k \in \mathbb{N}$ festgelegt wird, so kann der Speicherbedarf mit $O(\max\{2^k, z\})$ 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 implizite Faktoren speichert. Die Anzahl der Blöcke, die im folgenden Schritt in der Referencescan-Routine3.5 markiert werden können, ist durch die Anzahl der Einträge in der RFP-Tabelle beschränkt. Der Wert $k \in [0, 1]$ gebe den Anteil der Einträge in der RFPTable in Relation zur Anzahl 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 sinkt 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. Damit ist die Anzahl der Faktoren, die durch diese Technik nicht direkt ausgegeben werden, stets kleiner als $k_{min} \cdot z$ und weiterhin bei der endgültigen Ausgabe durch $2 \cdot k_{min} \cdot z$ beschränkt. In der Gesamtheit aller Runden ergibt sich dadurch eine obere Schranke für die relative Verschlechterung der Faktorrate um den Faktor $k_{min} * log_2|S|$, wobei diese Schätzung sehr konservativ ist.

Kapitel 4

Praktische Evaluation

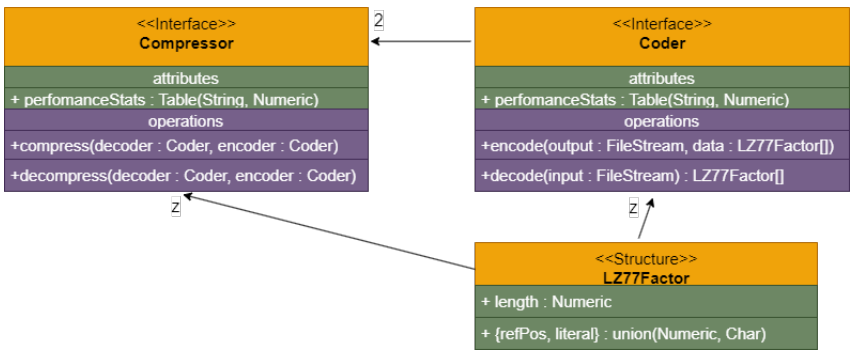
4.1 Testumgebung

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 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[4] 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 des Ausführung der Algorithmen aus.

Unordered-Dense-Map

Die Unordered-Dense-Map[8] stellt eine hochperformante 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[2][7] als effizienter herausgestellt.

LibSaiS

Für die Implementierung der exakten LZ77-Faktorisierung, die als Referenzalgorithmus fungiert, wurde die Bibliothek LibSaiS[3] zu Hilfe genommen. Diese Bibliothek stellt eine effiziente Implementierung für die Konstruktion des Suffix-Arrays bereit.

4.2.3 Parametrisierte Einstellung

Falls nicht anders ausgewiesen wurden die Approximationsalgorithmen mit allen prakischen Optimierungen aus Kapitel 3.4 ausgeführt. Im Falle von Approx.LZ77Par wurde die Ausführung standardmäßig mit 16 Threads durchgeführt. Unter Berücksichtigung der verwendeten Eingabedaten wurde folgende Parameter für die Optimierungen festgelegt:

Tabelle 4.1: Parameter und Einstellung für die Approximationsalgorithmen

Einstellung	**Wert**
DynEnd	Aktiv
DynStart	Aktiv
PreMatching	$r_{prematching} = r_{DynEnd} - 3$
ScanSkip	$k_{min} = 3\%$

4.3 Messung

4.3.1 Eingabedaten

Die folgenden Algorithmen wurden auf verschiedenen Dateien aus dem Pizza & Chili-Corpus 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

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.

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 der 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.

$$|K_{LZ77}(f)| = 1 + \begin{cases} 2\log_2(n) & \text{falls } |f| > 1 \\ 9 & \text{sonst} \end{cases} \tag{4.1}$$

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 ASCII-Kodierung des Zeichens verwendet, die 8 Bits benötigt.

$$|K_{Approx.LZ77}| = 1 + \begin{cases} \log_2(n) + \log_2(\log_2(n)) & \text{falls } |f| > 1 \\ 9 & \text{sonst} \end{cases} \tag{4.2}$$

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.

4.3.3 Messwerte

Tabelle 4.3: Messwerte der Algorithmen auf verschiedenen Eingabedateien

Eingabe	Algorithmus	Laufzeit[s]	Speicher[Byte]	FR	CR*
proteins	LZ77	15.40	20.00	9.98%	70.83%
	Approx.LZ77	44.06	9.94	15.34%	63.95%
	Approx.LZ77Par	6.99	10.21	15.34%	63.95%
sources	LZ77	13.47	20.00	5.60%	38.75%
	Approx.LZ77	40.43	6.42	10.05%	40.14%
	Approx.LZ77Par	6.49	5.90	10.05%	40.14%
english	LZ77	15.36	20.00	6.70%	47.24%
	Approx.LZ77	51.00	7.06	10.42%	43.39%
	Approx.LZ77Par	7.46	6.16	10.42%	43.39%
dna	LZ77	14.89	20.00	6.66%	47.46%
	Approx.LZ77	30.10	8.38	10.71%	45.53%
	Approx.LZ77Par	4.80	6.66	10.71%	45.53%
xml	LZ77	13.02	20.00	3.42%	23.61%
	Approx.LZ77	29.38	3.46	6.62%	26.78%
	Approx.LZ77Par	4.91	3.46	6.62%	26.78%

4.4 Auswertung

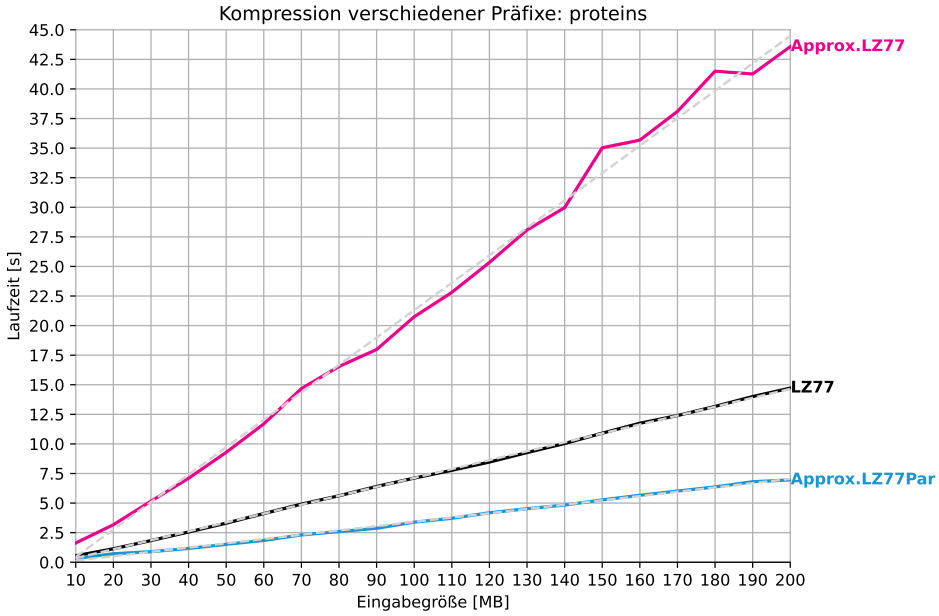
4.4.1 LZ77

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 4.2 deutlich, wo die Laufzeitmessung von LZ77 auf verschiedenen Präfixen von proteins dargestellt ist. Die lineare Regression der Kurven verdeutlicht das lineare Verhalten. Auch im Bezug auf die Speichernutzung wird das lineare Verhältnis zur Eingabegröße in Abbildung 4.4 deutlich. Das Verhältnis von allokiertem Speicher zur Eingabegröße beträgt in nahezu allen Präfixen von proteins 20 Byte pro Eingabezeichen. 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. Weiterhin wurde eine schlechtere theoretische Laufzeitkomplexität von $O(n \log n)$ gegenüber $O(n)$

Abbildung 4.2: 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.

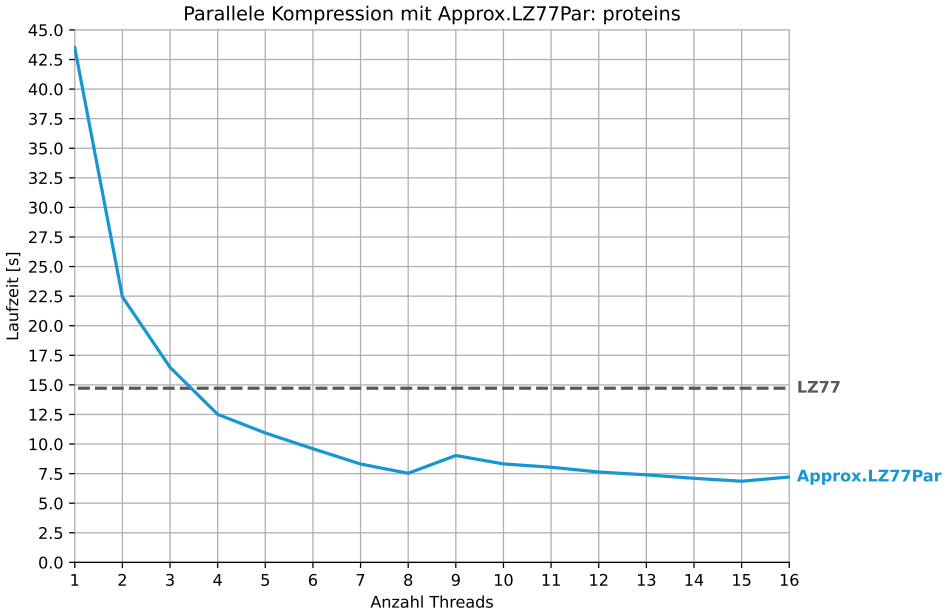


von der exakten LZ77-Faktorisierung festgestellt. Die Messwerte in 4.3 bestätigen diese theoretische Analyse damit. Es ist jedoch zu beachten, dass Approx. LZ77 wie zu erwarten auch eine deutlich bessere Speichernutzung aufweist, wobei diese stark von der Eingabe abhängt. In Abbildung 4.4 wird deutlich, 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 praktische Optimierungen aus Kapitel 3.4 wurden in der Evalutation 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. 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 Pre-

Abbildung 4.3: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für proteins

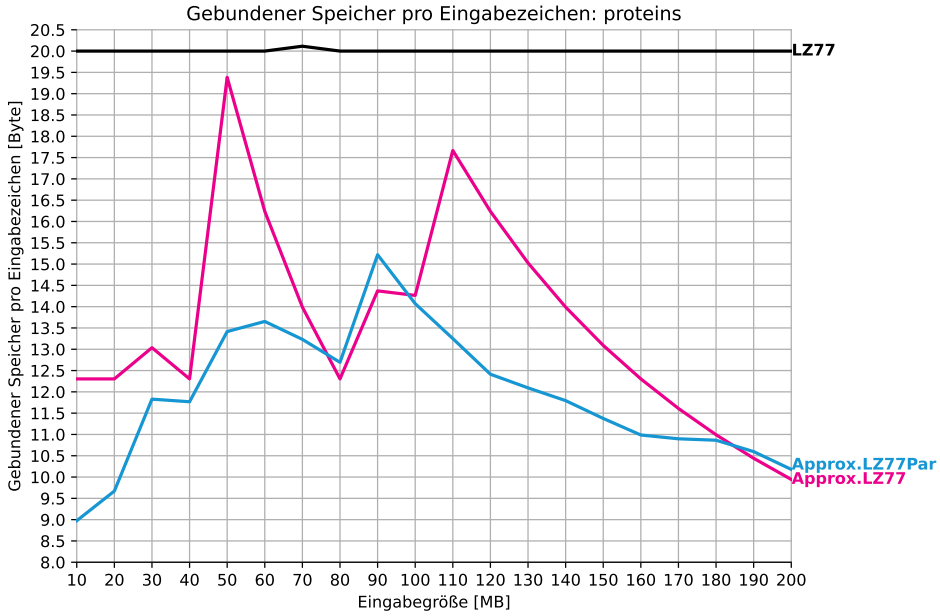


Matching zurückführen, da in InitTables vorgefiltert werden und so weniger Einfügeoperationen stattfinden, sowie in Postprocess die Spaltung der Blöcke durch vorberechnete RFPs beschleunigt wird. Dieser Effekt verfällt enstprechend nach der vorberechneten Runde, hier 24. Schließlich zeigt sich die Wirkung von ScanSkip in den Ausfällen der ReferenceScan-Routine in den ersten vier und der letzten Runde. Die Kombination aus der Filterung durch PreMatching und der Eliminierung von Duplikaten in InitTables führt in beiden Fällen zu einer relativ kleinen RFPTable. 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

Im 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 4.2 zeigt, dass Approx.LZ77Par mit 16 Threads eine deutlich bessere Laufzeit aufweist als Approx.LZ77. Die Laufzeitmessung in 4.3 zeigt, dass die Laufzeit von Approx.LZ77Par von einer kleinen Anzahl an Threads stärker profitiert. Mit zunehmender Anzahl an Threads wird die Beschleunigung bzw. der Speedup geringer. Die resultierende Asymptote in der Laufzeitmessung ist auf externe Faktoren zurückzuführen, wie der Bandbreite der Speicherzugriffe und Symptomen von False-Sharing. Der Speicherverbrauch von Approx.LZ77Par ist in 4.4 ebenfalls aufgezeichnet. Es fällt auf, dass der Speicherverbrauch von Approx.LZ77Par im Vergleich zu Approx.LZ77 in den meisten

Abbildung 4.4: Speicherverbrauch von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von proteins. Aufgezeichnet wurde das Verhältnis von allokiertem Speicher zur Eingabegröße.



Fällen leicht geringer ausfällt. Dies ist auf die Verwendung von mehreren Instanzen von Unordered-Dense-Map 4.2.2 zurückzuführen, die aufgrund ihrer inhärenten Struktur einen geringen gesamten Speicherverbrauch aufweisen.

Abbildung 4.5: Darstellung der Verteilung der Laufzeit der Subroutinen innerhalb jeder Runde im Falle einer Ausführung von Approx.LZ77 ohne jegliche Optimierungen 3.4

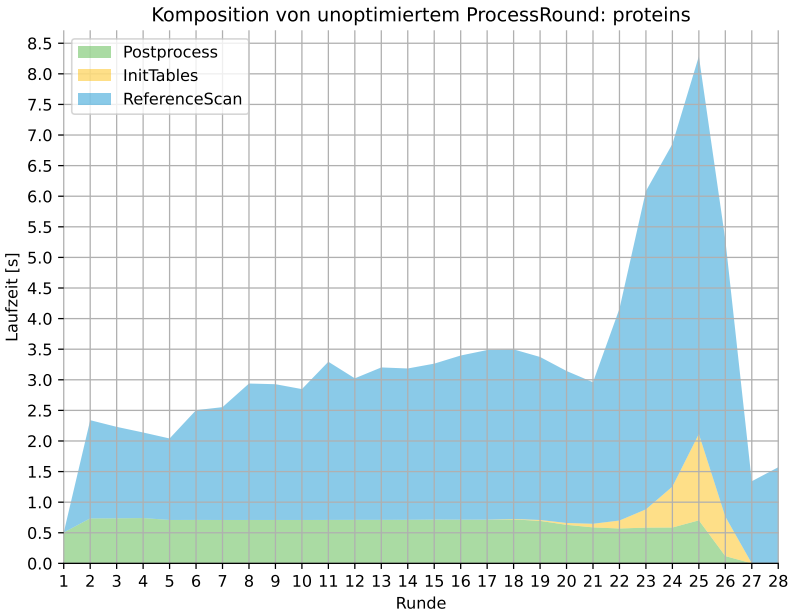
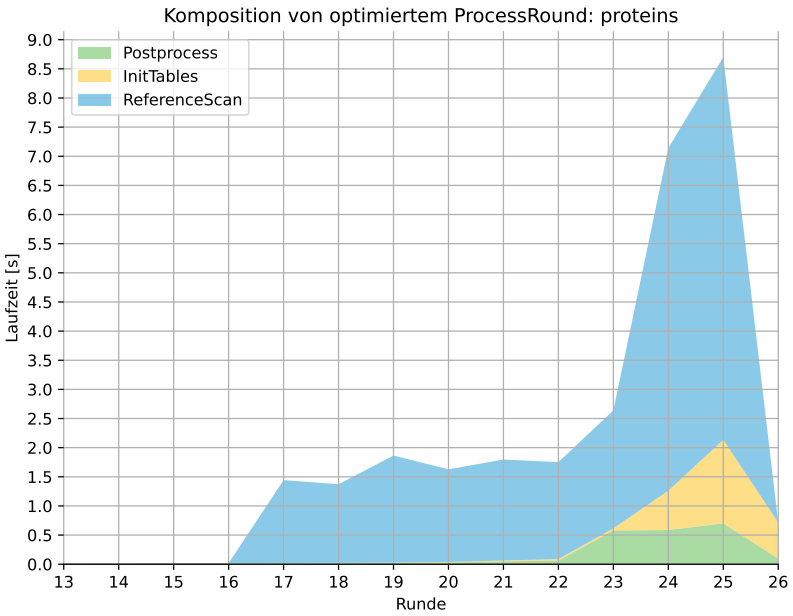


Abbildung 4.6: Darstellung der Verteilung der Laufzeit der Subroutinen innerhalb jeder Runde im Falle einer Ausführung von Approx.LZ77 mit allen Optimierungen 4.1



Kapitel 5

Fazit

5.1 Zusammenfassung und Einordnung

Im Rahmen dieser Arbeit haben wir die erste Phase eines approximativen LZ77- Algorithmus, der in erster Linie auf die Minimierung 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 Optimierung der Parallelisierung, die auf einer hardwarenahen Steuerung der Speicherzugriffe basieren, die Grenzen der Bandbreite des Speichersystems besser ausnutzen. Im Laufe der Ausarbeitung dieser 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 [9] das Volumen der Suchoperationen von Referenzen reduzieren. Weitehrin wäre eine Paralleliserung der weiteren zwei Phasen des approximativen LZ77-Kompressionsalgorithmus eine sinnvolle Erweiterung.

Anhang A

Weitere Informationen

A.1 Alternative Eingabedaten

Im Folgenden sind die vollständigen Laufzeitmessungen für die Eingabedaten, sources, english, dna und xml, aufgeführt. Während die absoluten Werte der Laufzeit für die verschiedenen Eingabedaten variieren, zeigen die Kurven der Laufzeitmessungen eine ähnliche Tendenz.

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.

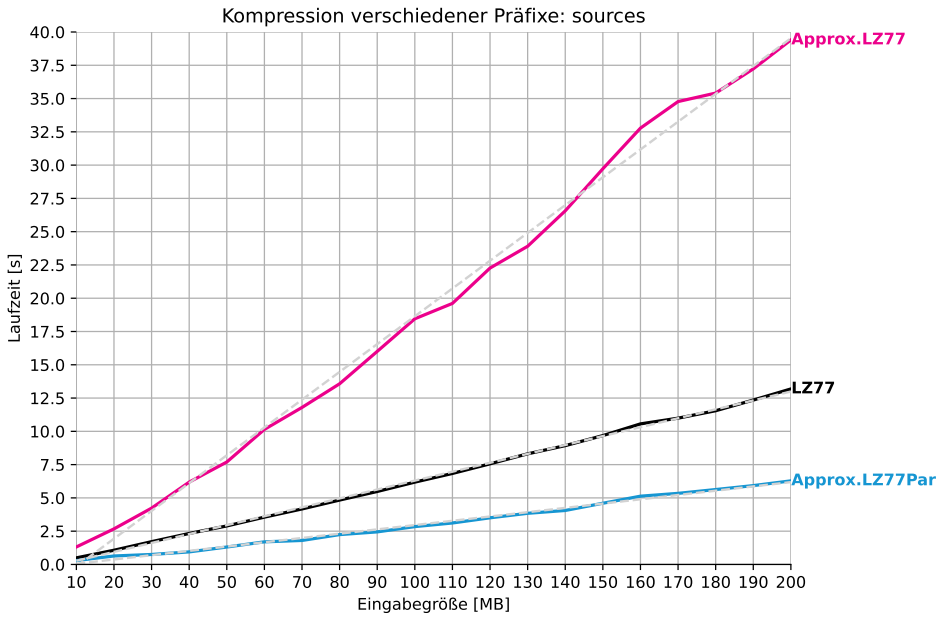
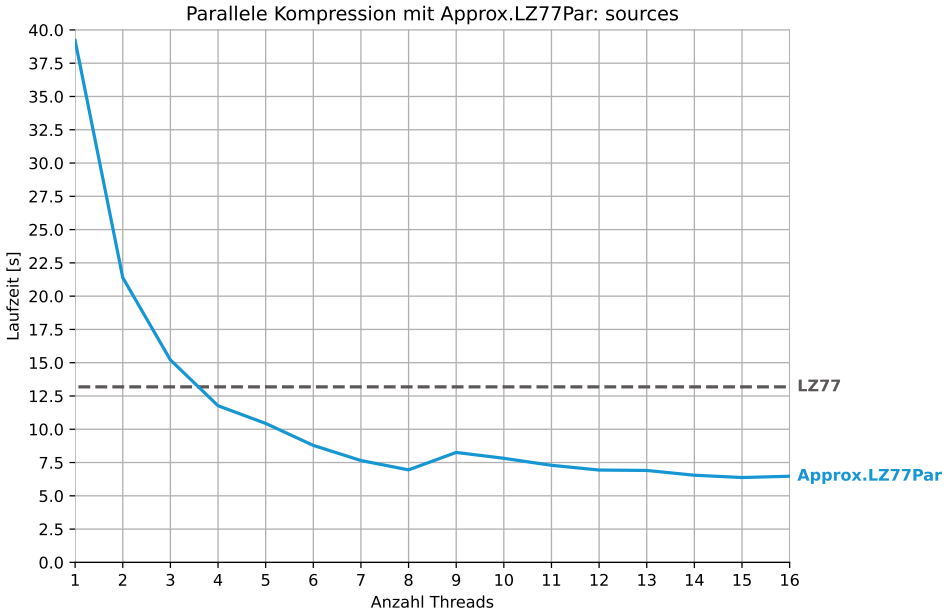


Abbildung A.2: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für sources



A.2 Alternative Testumgebung

Für die folgenden Messwerte wurden die Algorithmen auf einer Rechner mit einem AMD EYPC 32-Core Prozessor mit 128 nutzbaren Threads ausgeführt. Die Einstellungen der Algorithmen, die in 4.1 etabliert wurden, wurden beibehalten. ToDo: Tabelle einfügen

Abbildung A.3: 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.

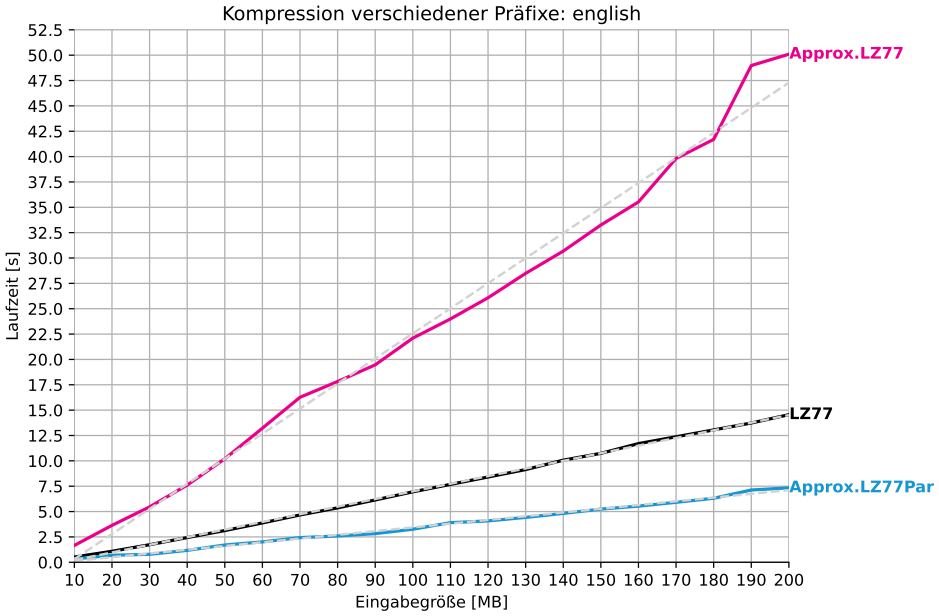


Abbildung A.4: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für english

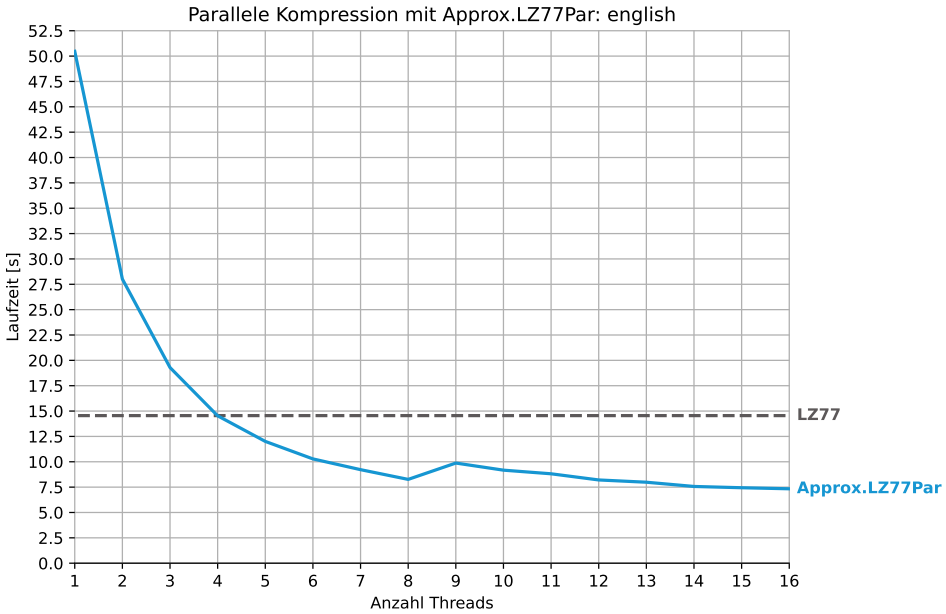


Abbildung A.5: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von dna. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

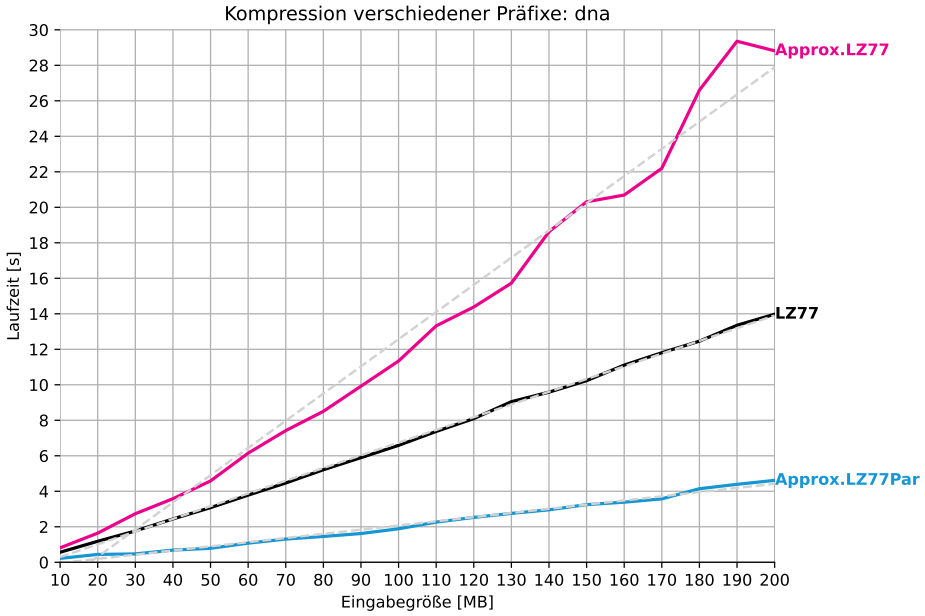


Abbildung A.6: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für dna

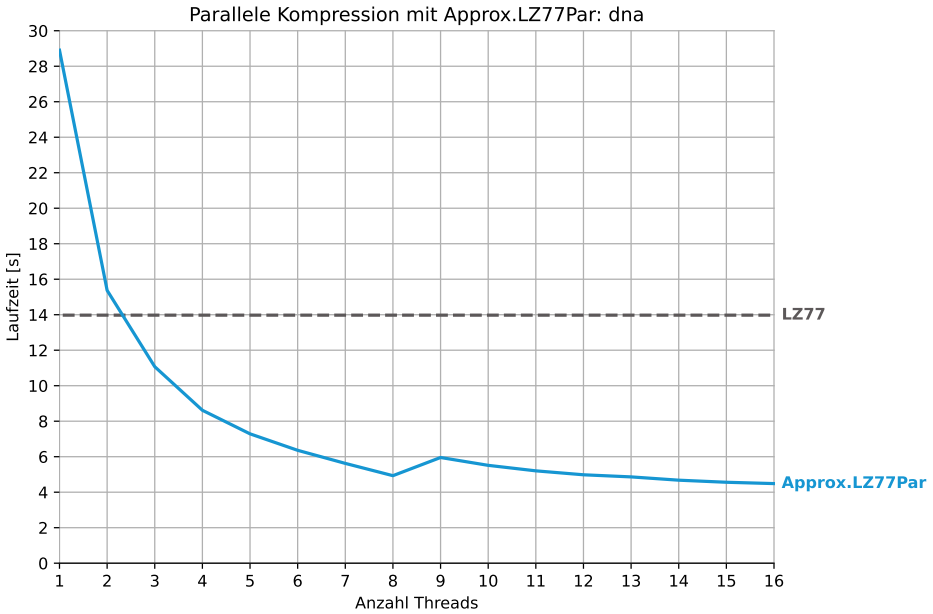


Abbildung A.7: Laufzeitmessung von LZ77, Approx.LZ77 und Approx.LZ77Par(16 Threads) auf verschiedenen Präfixen von xml. Als Vergleichsmaß wurde die lineare Regression der Kurven gestrichelt eingezeichnet.

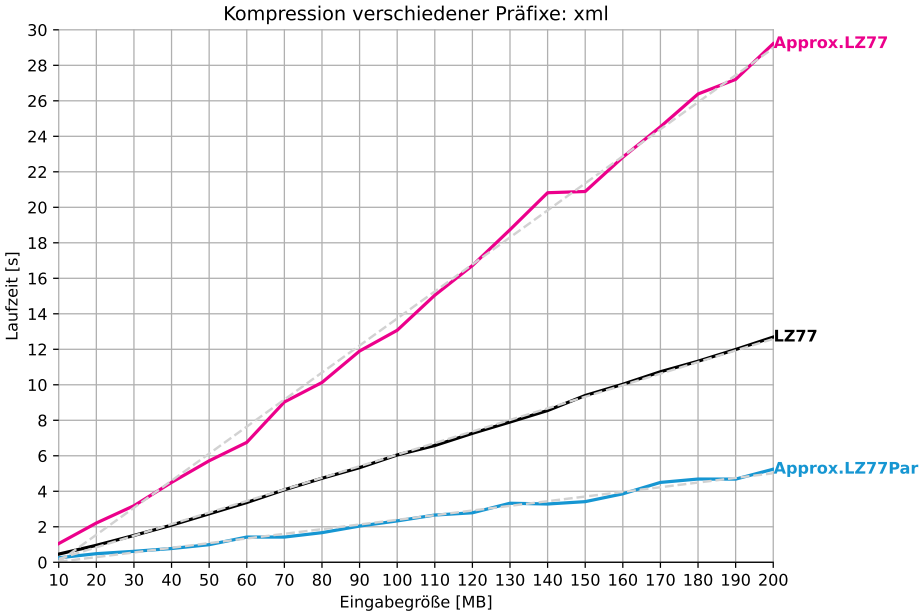
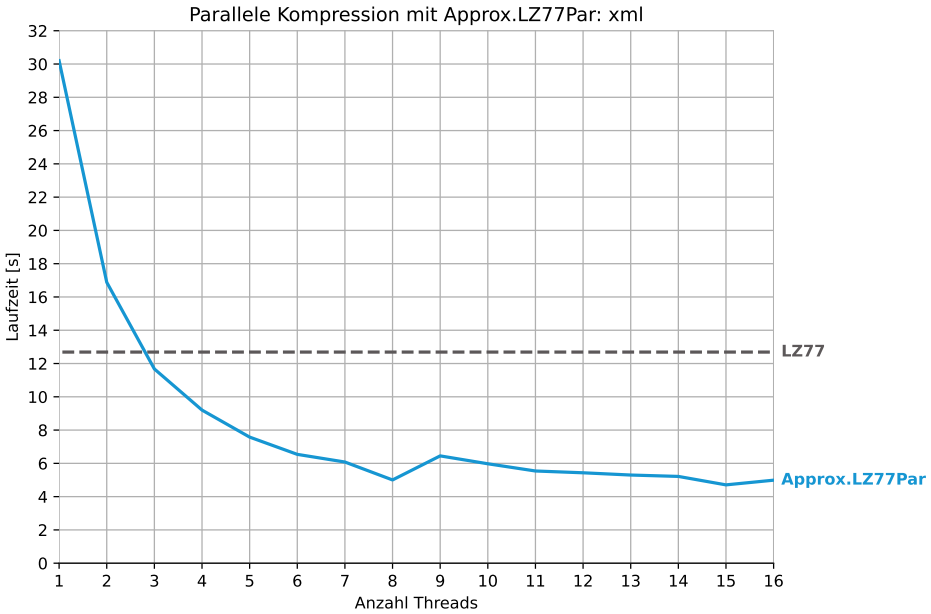


Abbildung A.8: Laufzeitmessung von Approx.LZ77Par mit verschiedener Anzahl an Threads für xml



Literaturverzeichnis

[1] *Amdahls Law*. https://en.wikipedia.org/wiki/Amdahl%27s_law.

[2] *Intel oneAPI Threading Building Blocks*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

[3] *libsais*. https://github.com/IlyaGrebnov/libsais.git.

[4] *Malloc Count*. https://github.com/ByteHamster-etc/malloc_count.git.

[5] *OpenMP ARB*. https://www.openmp.org.

[6] *The Pizza& Chili Corpus*. https://pizzachili.dcc.uchile.cl/texts.html.

[7] *Sharded Map*. https://github.com/Skadic/sharded_map.git.

[8] *Unordered Dense Hash Map*. https://github.com/martinus/unordered_dense.git.

[9] Bloom, Burton H.: *Space/time trade-offs in hash coding with allowable errors*. Commun. ACM, 13(7):422–426, jul 1970.

[10] E., Ohlebusch: *Lempel-Ziv Factorization: LZ77 without Window*, 2016.

[11] Fischer, Johannes, Travis Gagie, Paweł Gawrychowski und Tomasz Kociumaka: *Approximating LZ77 via small-space multiple-pattern matching*. In: *23rd European Symposium on Algorithms (ESA)*, Band 9294, Seiten 533–544. Springer, 2015.

[12] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger Roman Dementiev: *Sequential and Parallel Algorithms and Data Structures*. https://doi.org/10.1007/978-3-030-25209-0, 2020.

[13] Storer, James A. und Thomas G. Szymanski: *Data compression via textual substitution*. J. ACM, 29(4):928–951, oct 1982.

[14] Ziv, J. und A. Lempel: *A universal algorithm for sequential data compression*. IEEE Transactions on Information Theory, 23(3):337–343, 1977.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 12. August 2024

Muster Mustermann