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

1 Einleitung **1**

1.1 Motivation und Hintergrund 1

1.2 Aufbau der Arbeit 1

2 Grundlagen **3**

2.1 Eingabe 3

2.2 Ausgabe → Faktorisierung 3

2.3 Kompression 4

2.3.1 Verlustfreie Kompression 4

2.3.2 Dekompression 4

2.3.3 String-Matching → Rabin-Karp 4

2.3.4 Verlustbehaftete Kompression 6

2.3.5 Binäre (De-)Kodierung 6

2.3.6 Metriken 7

2.4 Parallelität 7

2.4.1 Shared-Memory-Modell 7

2.4.2 Metriken 7

3 Kompressionsalgorithmen **9**

3.1 (exakte) LZ77-Kompression 9

3.1.1 Konzept 9

3.1.2 Theoretisches Laufzeit- und Speicherverhalten 9

3.2 Approximation der LZ77-Faktorisierung(Approx. LZ77) 11

3.2.1 Konzept 11

3.2.2 Theoretisches Laufzeit- und Speicherverhalten 12

3.3 Parallelisierung von Approx. LZ77(Approx. LZ77Par) 12

3.3.1 Konzept 12

3.3.2 Theoretisches Laufzeit- und Speicherverhalten 12

3.4 Praktische Optimierungen 13

3.4.1 Dynamische Endrunde(DynEnd) - Laufzeit vs. Qualität* 13

3.4.2 Dynamische Startrunde(DynStart) - Laufzeit vs. Speicher 14
3.4.3 Vorberechnete Runde(PreMatching) - Laufzeit vs. Speicher 14
3.4.4 Minimale Tabellengröße(ScanSkip) - Laufzeit vs. Qualität 14

4 Praktische Evaluation **15**
4.1 Testumgebung 15
4.2 Implementierung 15
4.2.1 Klassenstruktur 15
4.2.2 Code 15
4.3 Messung 15
4.3.1 Eingabedaten 15
4.3.2 Messgrößen 16
4.3.3 Messwerte 16
4.4 Auswertung 16
4.4.1 LZ77 16
4.4.2 Approx. LZ77 16
4.4.3 Approx. LZ77Par 16

A Weitere Informationen **19**

Literaturverzeichnis **21**

Erklärung **21**

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Eine Referenz [1].

1.2 Aufbau der Arbeit

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 werden. Dabei ist jeder Faktor f_i mit $1 \leq i \leq z$ als 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 ohne vorheriges Vorkommen. Die im Folgenden betrachteten Algorithmen können speziell der Klasse der LZ77-Kompressionsverfahren zugeordnet werden, dessen Faktoren im Schema des Lempel-Ziv-Storer-Szymanski 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 $\rightarrow$ Rabin-Karp

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

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
```

führt. 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 Fingerprints. Im Falle von gleichen Fingerprints, 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.

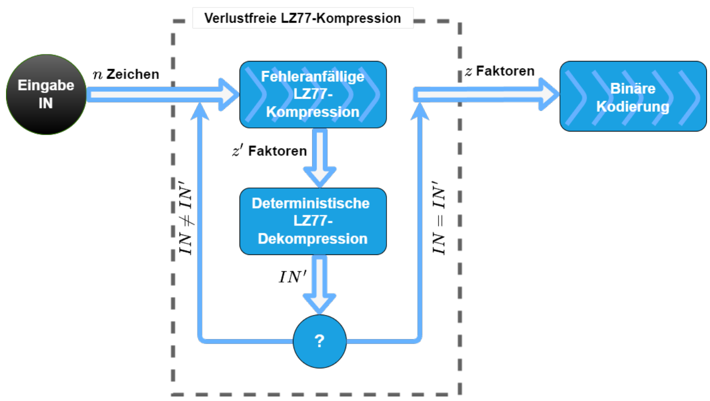
Rabin-Karp-Fingerprints erlauben verschiedene Operationen auf Zeichenfolgen, die im Rahmen der approximativen LZ77-Faktorisierung effizient genutzt werden können. Zum Einen kann ein beschränktes Fenster $S_W = S(j..j+w)$ der Länge $w < n$ leicht verschoben werden. Sei $RFP(S_W)$ der Fingerabdruck des Fensters, so kann der Fingerabdruck 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. 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: Las-Vegas-Algorithmus



berechnet werden. Ähnlich 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 agiere 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 Arbeit und Kommunikation unter den Prozessoren wird man jedoch auf Konflikte bei gleichzeitigen Speicherzugriffen zustoßen. Ein parallel modellierter Algorithmus muss explizit hinsichtlich der Korrektheit und Effizienz Mechanismen zur Synchronisation implementieren.

2.4.2 Metriken

Das Ziel der Parallelisierung eines Algorithmus liegt hauptsächlich in einer Verbesserung der Laufzeit, insbesondere unter Berücksichtigung von 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 Amdahlschen Gesetzes 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 xx 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 beschrieben. Der Algorithmus erzeugt zunächst ein SuffixArray, welches allen Suffixen der Eingabe eine lexikographische Ordnung zuweist. Mithilfe der loxikographischen Ordnung können Kandidaten für Referenzen effizient gefunden werden. Hierfür werden mit Hilfe des SuffixArrays 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 $NSV[k]$ 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 SuffixArrays und die folgende Berechnung der NSV- und PSV-Arrays können mithilfe von Algorithmen aus der Literatur(siehe xx) in $O(n)$ Laufzeit durchgeführt

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

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 SuffixArrays 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

Eine Approximation der LZ77-Faktorisierung ist ein Algorithmus, der ebenfalls eine Faktorisierung einer Eingabe S derart erzeugt, dass eine verlustfreie Dekrompression mit 2.1 möglich ist. Im Gegensatz zur exakten LZ77-Faktorisierung wird jedoch kein Greedy-Ansatz verwendet, um die längsten Referenzen zu finden. Stattdessen wird eine Approximation der optimalen Faktorisierung erzeugt, die einen Tradeoff zwischen der Qualität und der Performanz des Algorithmus darstellt. In dieser Arbeit wird die erste Phase des approximativen LZ77-Algorithmus, im Folgenden Approx. LZ77 genannt, herangezogen. Das Ergebnis von Approx. LZ77 ist eine Faktorisierung, in der Faktoren nur Zweierpotenzen als Länge haben. Im Folgenden gehen wir davon aus, dass die Länge unserer Eingabe eine Zweierpotenz ist. In der Praxis kann eine abweichende Länge durch entsprechendes Padding erreicht werden. Der Algorithmus teilt ihren Ablauf in Runden ein, wobei in jeder Runde die noch unverarbeitete Zeichenfolge in Blöcke gleicher Größe eingeteilt werden. In der ersten Runde entspricht die Blockgröße der Hälfte der Eingabelänge und wird sukzessive halbiert, bis die Zeichenfolge vollständig verarbeitet oder die Blockgröße 1 erreicht wurde. In jeder Runde werden für die erzeugten Blöcke Referenzen gesucht. Im Erfolgsfall wird ein entsprechender Faktor extrahiert und die Zeichenfolge gilt als verarbeitet. In 3.2 wird der Ablauf des Algorithmus illustriert. In der initialen Runde r wird die Eingabe in der Routine InitNodes zunächst in 2^r Blöcke gleicher Größe eingeteilt. Die erzeugten Bläcke repräsentieren die komplette Eingabe und werden in mehreren Runden einer Schleife verarbeitet. In der MatchNodes-Routine wird die Eingabe S auf Referenzen, also früher Vorkommen der Blöcke, durchsucht. Im Falle eines Treffers, werden referenzierte BLöcke markiert und die Position der Referenz abgespeichert. Die Routine gibt die Menge der markierten Blöcke und dessen Referenzpositionen aus. In der Runde r besitzen gefundene Referenzen bzw. Faktoren eine Länge von $\frac{|S|}{2^r}$, jeder markierte Block dem Faktor $(\frac{|S|}{2^r}, Referenzposition)$ entspricht. Schließlich werden die markierten Blöcke aus der Menge der zu verarbeitenden

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

```
Eingabe:S = e_1...e_n Ausgabe:F = f_1...f_z
F ← ∅
r ← 1
Blocks[1..2^r] ← InitNodes(S, 2^r) // Split S into 2^r equal blocks
while r ≤ log_2(|S|) do
   (markedBlock, refPos)[1..z_r] ← MatchNodes(r, S, blocks)
   for i ← 1 to z_r do
      F ← InsertFactor(F, (length = |S|/2^r, ref = refPos[i]), markedBlock[i])
   end for
   blocks ← NextNodes(blocks \ markedBlock[1..z_r]) // Halve unmarked blocks
   r ← r + 1
end while
return F
```

Blöcke entfernt, die verbleibenden Blöcke halbiert und die nächste Runde gestartet. Da eine Blöckgröße von 1 nicht unterschritten werden kann, terminiert der Algorithmus spätestens nach $log_2(|S|)$ Runden. Für die Extraktion von Referenzen wird die Technik des Rabin-Karp-Fingerprints verwendet. Dabie wird jedem Block, der eine Zeichenfolge repräsentiert, ein Hashwert zugewiesen und in einer Hashtabelle abgespeichert. Im Anschluss kann ein einfacher Durchgang der Eingabe mithilfe eines Rolling-Hashes und der Hashtabelle die Referenzen in linearer Zeit finden.

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 $log_2(|S|) = log_2(n)$, wobei in jeder Runde ein einfacher Durchgang der Eingabe S notwendig ist, um ggf. Referenzen zu finden. Damit kann die Laufzeit des Algorithmus mit $O(n \log n)$ abgeschätzt werden.

3.3 Parallelisierung von Approx. LZ77(Approx. LZ77Par)

3.3.1 Konzept

3.3.2 Theoretisches Laufzeit- und Speicherverhalten

Eine theoretische Laufzeit von $O(\frac{n \log n}{p})$ kann erreicht werden, wobei p die Anzahl der Prozessoren ist. Der Speicherbedarf des Algorithmus beträgt $O(z)$. 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 verbessern können auf Kosten von anderen Metriken. Jede einzelne Technik ist 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 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

3.4.3 Vorberechnete Runde(PreMatching) - Laufzeit vs. Speicher

3.4.4 Minimale Tabellengröße(ScanSkip) - Laufzeit vs. Qualität

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 Ausführung der Algorithmen mit einer spezifischen Anzahl von Threads wurde softwareseitig über OpenMP-Instruktionen realisiert.

4.2 Implementierung

4.2.1 Klassenstruktur

4.2.2 Code

Die konkrete Implementierung erfolgte in der Programmiersprache C++ im C++20 - Standard und dem Build - System CMake in der Version 3.28. Die Parallelisierung wurde über Präprozessor - Instruktionen von OpenMP realisiert.

4.3 Messung

4.3.1 Eingabedaten

Die folgenden Algorithmend wurden auf verschiedenen Dateien aus dem Pizza & Chili-Corpus getestet. Die verwendeten Dateien decken verschiedene Kontexte und damit Kompressionspotentiale ab. In der Tabelle 4.1 sind die verwendeten Dateien aufgelistet. Die Größe der Dateien wurde auf 100MB beschränkt, um einen angemessenen Rahmen für die Laufzeitmessung zu erhalten.

Abbildung 4.1: Auflistung der verwendeten Eingabedaten

Datei	Größe	Alphabet	Beschreibung
`dna`	100MB	4	DNA-Sequenzen
`english`	100MB	256	Englische Texte
`proteins`	100MB	20	Proteinsequenzen
`sources`	100MB	256	Quellcode
`xml`	100MB	256	XML-Dateien

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 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.

4.3.3 Messwerte

Laufzeiten

Speicherverbrauch

FR und CR*

4.4 Auswertung

4.4.1 LZ77

4.4.2 Approx. LZ77

4.4.3 Approx. LZ77Par

Abbildung 4.2: Laufzeit der Algorithmen auf Präfixen verschiedener Größen der Datei Proteins

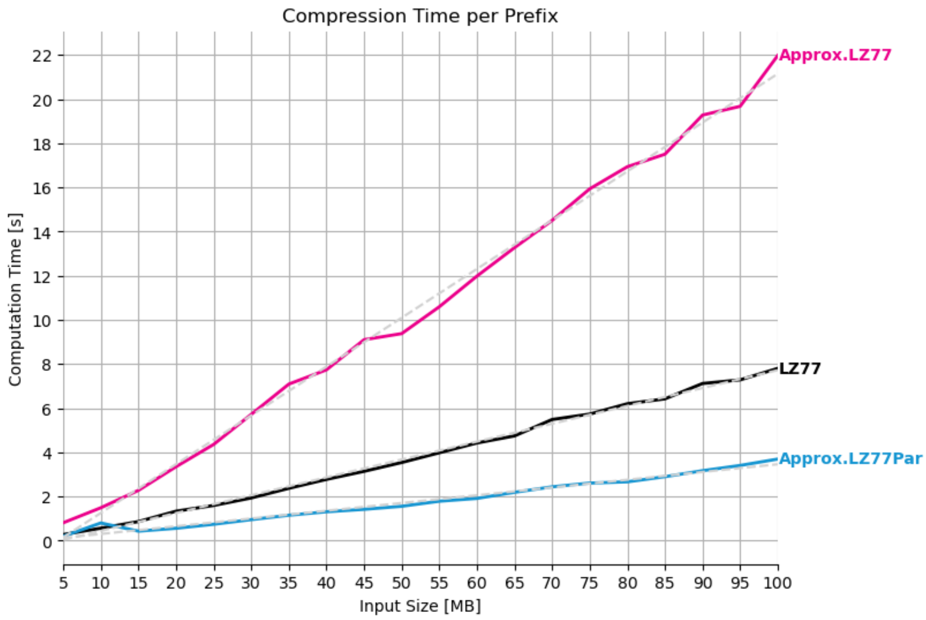
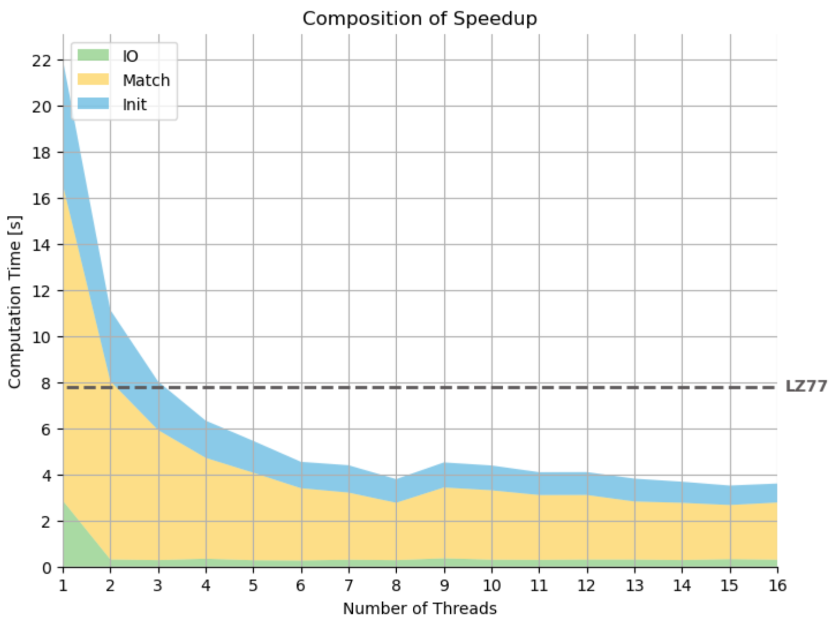


Abbildung 4.3: Laufzeit der parallelen Approximation von LZ77 mit verschiedenen Anzahlen von Threads



Anhang A

Weitere Informationen

Literaturverzeichnis

[1] Aggarwal, Alok und Jeffrey Scott Vitter: *The Input/Output Complexity of Sorting and Related Problems.* Communications of the ACM, 31(9):1116–1127, 1988.

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 23. Juli 2024

Muster Mustermann