technische universität
dortmund

Expose

Parallelisierung einer speichereffizienten Approximation der LZ77-Faktorisierung

Gajann Sivarajah

Gutachter:
Prof. Dr. Johannes Fischer
Patrick Dinklage

Technische Universität Dortmund
Fakultät für Informatik
LS-11
https://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 [9] wird eine Variante der LZ77-Faktorisierung beschrieben, die über drei Phasen eine 2-Approximation einer exakten LZ77-Faktorisierung [8] 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 zunächst und hauptsächlich 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 die OpenMP-Schnittstelle [3] realisiert, wobei alternative Konzepte, wie execution policies [6] aus der Standardbibliothek auch in Erwägung gezo-

gen werden können. Die parallele Manipulation von Daten in Datenstrukturen wird eine besondere Schwierigkeit sein. Im Rahmen des approximierten LZ77-Algorithmus werden wir Hash-Werte über partitionierte Datenblöcke generieren und in eine Hashtabelle einfügen [12] [9]. Die gängigsten Datenstrukturen aus der Standardbibliothek sind nicht für parallele Schreiboperationen geeignet. Beispielsweise kann die Einfügeoperation in einen std::unordered_map nicht parallel durchgeführt werden, weil einer dieser Operationen ein Rehashing initiieren kann und damit einen Synchronisierungskonflikt erzeugt [7]. Daher werden wir verschiedene Techniken dahingehend untersuchen und einen Blick auf bereits vorhandene Lösungen [2] [4] geworfen. Als Bewertungsmaßstab werden verschiedene Statistiken, unter Anderem die Laufzeit, die Speichenanforderung, sowie die Qualität der Kompression gemessen. Da hier nur die erste Phase berücksichtig wird, soll die Qualität der Kompression anhand der Zahl der Faktoren bemessen werden, die aus der Eingabe extrahiert wird. Wir werden grundlegend voraussetzen, dass jegliche Anpassungen am Algorithmus keinen negativen Einfluss auf die Kompressionsqualität bzw. der Anzahl der Faktoren haben darf. Weiterhin soll die Laufzeit in Relation zu der Anzahl der verwendeten nebenläufigen Threads beurteilt werden. Neben dem sequenziellen approximativen LZ77-Algorithmus sollen die Statistiken mit Messungen anderer Kompressionsalgorithmen, wie LZW [13] und dem exakten LZ77, verglichen werden. Weiterhin wäre ein Vergleich mit anderen parallelen Kompressionsalgorithmen von Interesse [10]. Für die Testeingabe wird der Pizza&Chili Corpus [5] herangezogen.

Kapitel 2

Arbeitsplan

2.1 Gliederung

1. Einleitung
2. Theoretische Grundlagen
    - Definitionen und Terminologie
    - Referenzalgorithmen
        - (exakte) LZ77-Faktorisierung
        - Approximative LZ77-Faktorisierung(Sequenziell)
    -
3. Parallele approximative LZ77-Faktorisierung - Konzept
    - Beschreibung des Algorithmus
    - Theoretisches Laufzeitverhalten und Speicheranforderung
4. Praktische Evaluation
    - Definition der Testumgebung
    - Beschreibung der Implementation
    - Grafische Aufarbeitung von Metriken(Laufzeit, Speicher, Faktorzahl)
    - Bewertung der Ergebnisse
5. Fazit und Ausblick

2.2 Zeitplan

Der Zeitplan sieht 4 Phasen vor, die wiederum einen Kreislauf aus **Konzept** ⇒ **Implementierung** ⇒ **Evaluation** ⇒ **Konz...** darstellen. Im Folgenden ist der chronologische Zeitplan(Annahme: 16 Wochen stehen zur Verfügung) zu sehen.

Zeitraum	**Phase**	**Treffen mit Betreuer**
W1-W2	Framework mit Referenz-Algorithmen	W2
W3-W4	Approx. LZ77(seq)	W3, W4
W5-W8	Approx. LZ77(par)	W5, W7, W8
W9-W12	Dokumentation und Optimierung	W10, W12
W13-W15	Präsentationsvorbereitung	W14, W15

Kapitel 3

Textbaustein - LZ77

3.1 Familie der LZ77-Algorithmen

Im Jahr 1977 beschrieben Abraham Lempel und Jacob Ziv erstmals eine Methode zur Textindizierung, die es erlaubt, Folgen von Datenzeichen dynamisch zu komprimieren, ohne ihren Kontext vorher zu kennen. Der Grundbaustein des Algorithmus ist die Generation von Faktoren, welche Teilfolgen sind, die bereits vorher in der Zeichenfolge aufkommen. Diese Faktoren können anschließend durch Referenzen auf ihre vorherigen Vorkommen ersetzt werden. Bis heute findet man zahlreiche Varianten und Weiterentwicklungen dieser Grundidee, wie z.B. im GZIP-Format, DEFLATE [1].

3.2 Konzept

3.2.1 Definitionen

Unsere Eingabe sei eine n-elementige Zeichenfolge $S = e_1...e_n$, die über einem beschränkten numerischen Alphabet Σ konstruiert wurde. Für eine Zeichenfolge S gibt der Ausdruck $|S|$ die Länge, hier n, an. Der Ausdruck $S[i..j] \in \Sigma^{j-i+1}$ mit $i, j \in [1, n]$ und $i \leq j$ beschreibt die Teilfolge $e_i...e_j$, wobei im Falle, dass $i = j$ gilt, das einzelne Zeichen e_i referenziert wird. In diesem Fall kann der Ausdruck $S[i..i]$ durch $S[i]$ ersetzt werden. Ein Präfix von S ist jede Teilfolge, die durch den Ausdruck $S[1..k]$ mit $k \leq n$ referenziert wird. Eine LZ77-Faktorisierung von S ist eine Partitionierung von $S = e_1...e_n$ zu $S = f_1...f_z$ mit $z \in N$ und $z \leq n$. Jeder Faktor f_i mit $1 \leq i \leq z$ stellt dabei entweder den größten Präfix in $S[|f_1...f_i - 1|..n]$ mit einem Vorkommen in $S[1..|f_1...f_i|]$ oder ein einzelnes neues Zeichen dar.

3.2.2 Algorithmus - Kompression

In 3.1 wird der sequenzielle und deterministische Prozess der Faktorisierung beschrieben. Ein referenzierender Faktor ist eindeutig durch seine Zielposition, der Länge der Über-

1: $S \leftarrow input$
2: $output \leftarrow EmptyQueue$
3: $i \leftarrow 1$
4: **while** $i \leq n$ **do**
5: $(pos, len) \leftarrow longestPreviousMatch(S[pos..pos + len] = S[i..i + len])$
6: $Append(pos, len, S[i + len])$ to output
7: $i \leftarrow i + 1$
8: **end while**
9: **return** output

Algorithmus 3.1: LZ77-Compression

einstimmung und dem darauf folgenden Zeichen bestimmt. Das Folgezeichen erhält seine Bedeutung aus dem Szenario, dass das aktuelle Zeichen kein vorheriges Vorkommen hat. In diesem Fall werden die Position und die Länge auf den Wert 0 normiert. Bei der Extraktion der längsten übereinstimmenden Zeichenfolge entsprechend der Zeile 5 kann der sichtbare Bereich vor und nach dem aktuellen Zeichen in seiner Größe beschränkt werden. Führt man diese Beschränkung nicht ein, so bezeichnet man die resultierende Faktorisierung als **exakt**. In [8] wird ein Algorithmus zur Generation einer exakten LZ77-Faktorisierung in $O(n)$ Laufzeit beschrieben. Für die Extraktion von Übereinstimmungen nutzt der Algorithmus ein Suffix-Array, welches jedoch zusätzlich zur Eingabe $O(n)$ an Speicherplatz belegt.

3.2.3 Algorithmus - Dekompression

1: $input \leftarrow (f_1, ...f_z)$
2: $output \leftarrow$ Empty String
3: $i \leftarrow 1$
4: **for** i from 1 to z **do**
5: $(pos, len, char) \leftarrow f_i$
6: **for** j from 0 to len-1 **do**
7: Append $output[pos...pos + j]$ to output
8: **end for**
9: Append char to output
10: **end for**
11: **return** output

Algorithmus 3.2: LZ77-Decompression

In 3.2 wird die Dekompression einer Faktorenfolge in die ursprüngliche Zeichenfolge beschrieben.Für jeden Faktor wird chronologisch entweder ein einzelnes Zeichen oder eine

referenzierte Teilfolge angefügt.Es kann jedoch sein, dass eine referenzierte Teilfolge über die bisher dekomprimierte Zeichenfolge geht. In diesem Fall muss der verfügbare Anteil durch Wiederholungen auf die geforderte Größe gestreckt werden.

3.3 Beispiel

3.3.1 Uniforme Zeichenfolge

Als Eingabe dient die uniforme Zeichenfolge S=aaaaaaaaa$.

1.

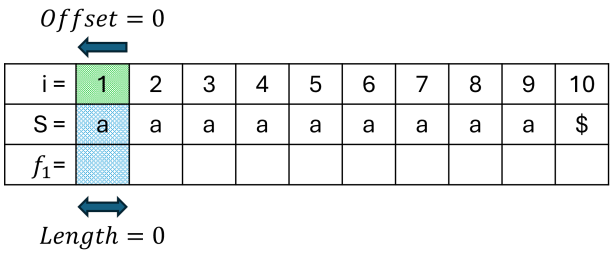

Abbildung 3.1: $i = 1$

2.

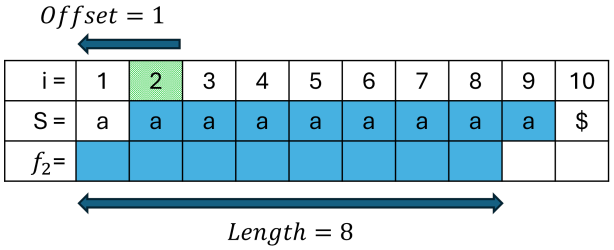

Abbildung 3.2: $i = 2$

In 3.1 ist zu sehen, dass das erste Zeichen 'a' kein vorheriges Vorkommen haben kann, sodass dieses Zeichen ohne Referenz gespeichert werden kann. Anschließend wird der Zeiger um eine Einheit erhöht. Das vorher gespeicherte Zeichen kann nun bis 9.ten Zeichen wiedergefunden werden, sodass eine Referenz auf eine 8-elementige Zeichenfolge gespeichert werden kann. Diese enthält die Verschiebung zum aktuellen Zeichen, die Länge der Teilfolge und das folgende Zeichen, hier $ 3.2 Das Ergebnis: $S = f_1 f_2$mit $f_1 = a$ und $f_2 = (pos = -1, len = 8, char = \$)$

3.3.2 Zeichenfolge mit Varianz

Als Eingabe dient die Zeichenfolge S=abaababaab$.

1.

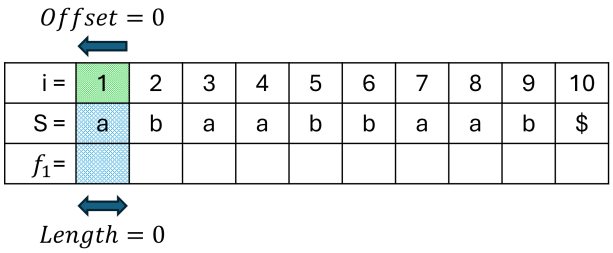

Abbildung 3.3: $i = 1$

2.

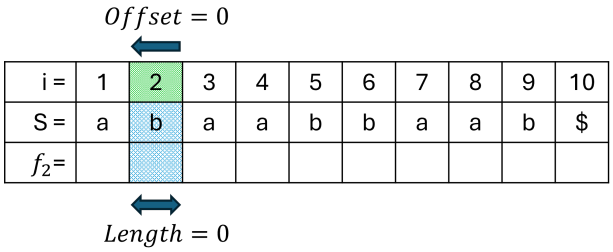

Abbildung 3.4: $i = 2$

3.

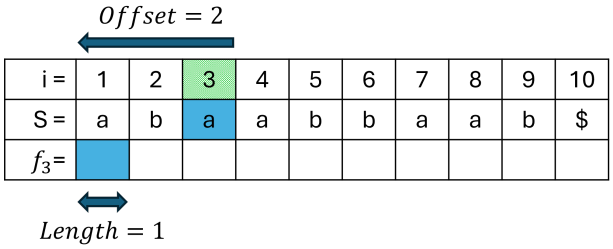

Abbildung 3.5: $i = 3$

4.

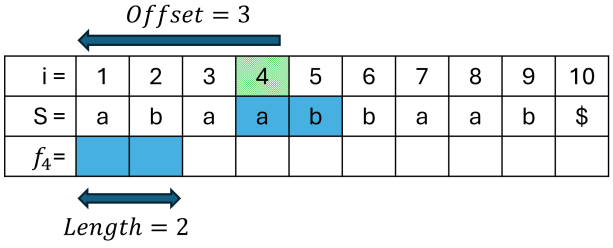

Abbildung 3.6: $i = 4$

5.

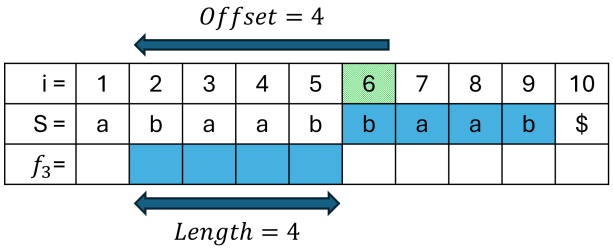

Abbildung 3.7: $i = 6$

In Anlehnung an das vorherige Beispiel können gerade am Anfang kaum Referenzen gefunden werden, sodass Zeichen einzeln gespeichert werden müssen, wie in 3.3 zu sehen ist. In 3.4 können anschließend mehrere übereinstimmende Teilfolgen in teils zunehmender Länge gefunden werden. Ergebnis: $S = f_1 f_2 f_3 f_4 f_5$ mit $f_1 = a, f_2 = b, f_3 = (-2, 1), f_4 = (-3, 2), f_5 = (-4, 4),\ f_6 = \$$

Literaturverzeichnis

[1] *DEFLATE Compressed Data Format Specification version 1.3.* https://www.rfc-editor.org/rfc/rfc1951.html.

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

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

[4] *Parallel Concurrent Datastructures.* https://github.com/TooBiased/growt.

[5] *The PizzaChili Corpus.* https://pizzachili.dcc.uchile.cl/texts.html.

[6] *STL Execution Policy.* https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t.

[7] *Thread Safety of STD Containers.* https://en.cppreference.com/w/cpp/container#Thread_safety.

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

[9] 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.

[10] Klein, Shmuel Tomi und Yair Wiseman: *Parallel Lempel Ziv coding.* Discrete Applied Mathematics, 146(2):180–191, 2005. 12th Annual Symposium on Combinatorial Pattern Matching.

[11] Prokopec, Aleksandar, Phil Bagwell, Tiark Rompf und Martin Odersky: *A Generic Parallel Collection Framework.* In: *Euro-Par 2011 Parallel Processing*, Seiten 136–147. Springer, 2011.

[12] Shah, Parth und Rachana Oza: *Improved Parallel Rabin-Karp Algorithm Using Compute Unified Device Architecture.* In: Satapathy, Suresh Chandra und Amit Joshi (Herausgeber): *Information and Communication Technology for Intelligent Systems*, Band 2, Seiten 236–244. Springer International Publishing, 2017.

[13] Welch: *A Technique for High-Performance Data Compression.* Computer, 17(6):8–19, 1984.

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