SIMD-Accelerated Regular Expression Matching

Seminar - Implementierungstechniken für MMDBS

Stefan Lachnit

29. Oktober 2018

Einleitung

- Reguläre Ausdrücke ermöglichen komplexe Abfragen bei Strings
- RE muss auf alle Einträge angewandt werden
- Beschleunigung durch vektorisierten Code möglich

```
select count(*) from employees
where email regexp
'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4}$'
```

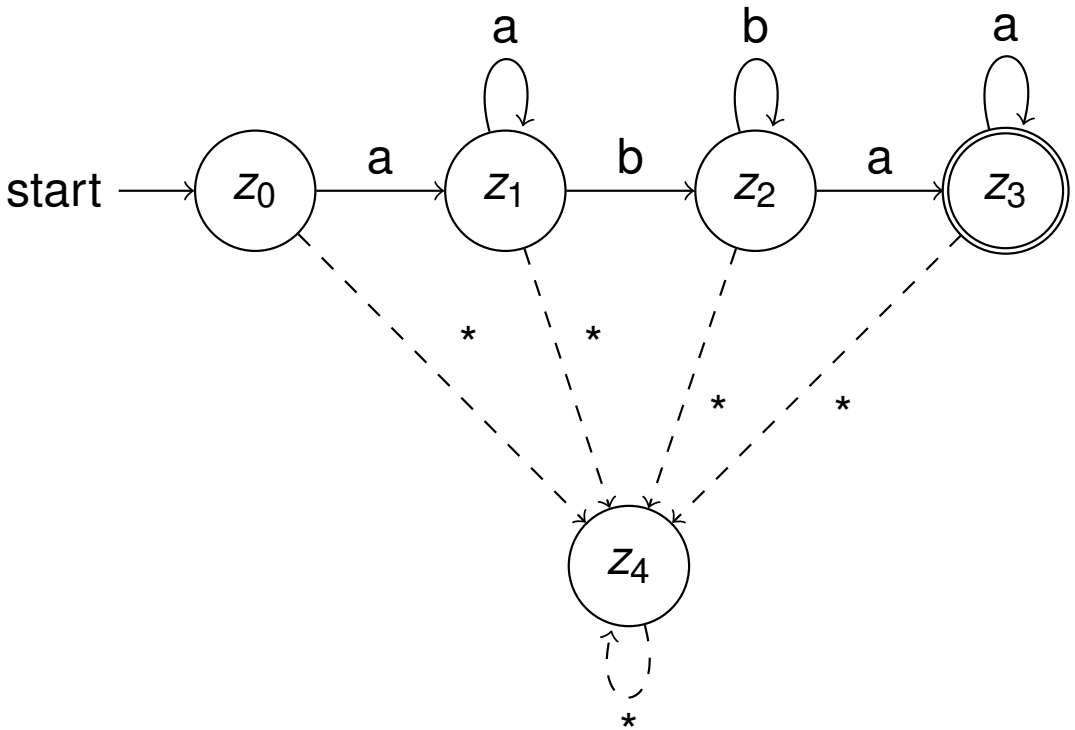
Abfrage: Anzahl der Einträge mit gültigen Mailadressen[1]

Einleitung

1. Umwandlung: RE $\rightarrow$ DFA
2. Datenstruktur DFA
3. AVX
4. SIMD-Beschleunigte Algorithmen
5. Benchmarks

Umwandlung: RE $\rightarrow$ DFA

RE: aa*bb*aa*



Umwandlung: RE $\rightarrow$ DFA

- RE $\rightarrow$ NFA
- NFA $\rightarrow$ DFA
- DFA minimieren

Umwandlung: RE $\rightarrow$ DFA

- RE $\rightarrow$ NFA $O(n)$
- NFA $\rightarrow$ DFA $O(2^n)$
- DFA minimieren $O(n \log n)$

Umwandlung: RE $\rightarrow$ DFA

- RE $\rightarrow$ NFA $O(n)$
- NFA $\rightarrow$ DFA $O(2^n)$
- DFA minimieren $O(n \log n)$

```
^(ht|f)tp(s)?://([!$&'()*+,;=A-Za-z0-9:-]+@)?
((((([_~!$&'()*+,;=A-Za-z0-9-]|(%[0-9A-F]{2}))+.)+[a-zA-Z]{2,4})
|((([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]).){3}
([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]))(:[0-9]*)?
(/([._~!$&'()*+,;=A-Za-z0-9:@-]|(%[0-9A-F]{2}))+)*
(?([?._~!$&'()*+,;=A-Za-z0-9:@/-]|(%[0-9A-F]{2}))*)?
(#([?._~!$&'()*+,;=A-Za-z0-9:@/-]|(%[0-9A-F]{2}))*)?$
```

URL-DFA (90 Zustände)[1]

Datenstruktur DFA

- DFA als eindimensionales Integer-Array
- 8-Bit Darstellung der Eingabe $\rightarrow$ alle Möglichkeiten für Zustandsübergänge von einem Zustand in einem 256 Elementebereich
- Eingabe c, aktueller Zustand a:

$$Folgezustand = DFA[a * 256 + zahlenwert(c)]$$

- Sortierung: Ablehnende Zustände zuerst
- Fehlerzustand als Zustand mit Offset 0
- Wort akzeptiert $\Leftrightarrow$

$$Endzustand > Anzahl\ ablehnender\ Zustände - 1$$

AVX

- Advanced Vector Extensions
- x86-Erweiterung
- Verfügbar in CPUs ab ca. 2013
- 16 256-Bit Register
- Befehle, um Operationen auf mehreren Elementen in einem Vektor gleichzeitig auszuführen (SIMD)
- Aufrufbar in C++ über intrinsische Funktionen

AVX

Wichtige Befehle

- _mm256_add_epi32 (__m256i a, __m256i b)
  Addiert die 8 Integer-Werte in den beiden Vektoren und gibt den Ergebnisvektor zurück

- _mm256_i32gather_epi32 (int const* a, __m256i i, const int s)
  Lädt Integer an den Indices im Vektor i, beginnend von Adresse a, aus dem Speicher. i wird dabei mit s skaliert.

- _mm256_maskstore_epi32 (int* m, __m256i mask, __m256i a)
  Speichert Daten aus Vektor a in den Speicher an Adresse m, aber nur die 32-Bit Elemente an denen das höchstwertige Bit in mask gesetzt ist

Skalarer Algorithmus

- Schleife über alle Strings
- Jeden String solange bearbeiten, bis der letzte Buchstabe oder der Zustand 0 erreicht ist
- Zustand mit der Anzahl abzulehnender Zustände vergleichen und ggf. in das Ergebnisarray speichern

SIMD-Beschleunigte Algorithmen

Algorithmus 1

- Laden der ersten Zeichen von 8 Strings in 8 Vektoren
- Zusammenfügen der jeweils ersten Bytes aus den Vektoren zu einem Vektor
- Berechnen und Laden (gather) der neuen Zustände für alle 8 Strings gleichzeitig
- Die 8 Vektoren um 1 Byte verschieben, damit die nächsten Buchstaben an erster Stelle stehen
- Wiederholen, bis das Ende der Strings erreicht ist und auf akzeptierende Zustände prüfen
- Wiederholen, bis alle Strings bearbeitet wurden

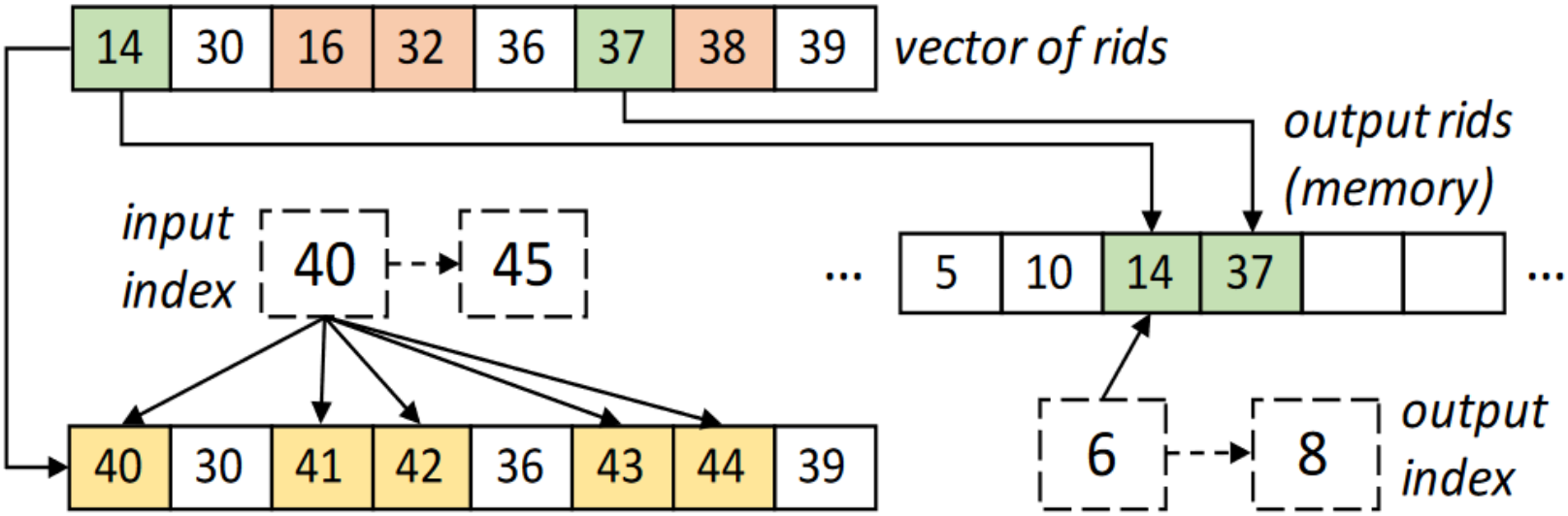
SIMD-Beschleunigte Algorithmen

Algorithmus 2

- Vektoren zum Speichern der String-Indices, der Offsets in den Strings und der aktuellen Zustände
- Laden von Bytes (4 Bytes) jedes Strings am entsprechenden Offset
- Berechnen und Laden (gather) der neuen Zustände
- Auf Ende des Strings und akzeptierenden Zustand prüfen und ggf. speichern
- Auf Ende des Strings oder Fehlerzustand prüfen, String-Indices austauschen und Zustände zurücksetzten
- Schleife über die 4 geladenen Bytes
- Schleife, bis alle Strings bearbeitet wurden

SIMD-Beschleunigte Algorithmen

Algorithmus 2

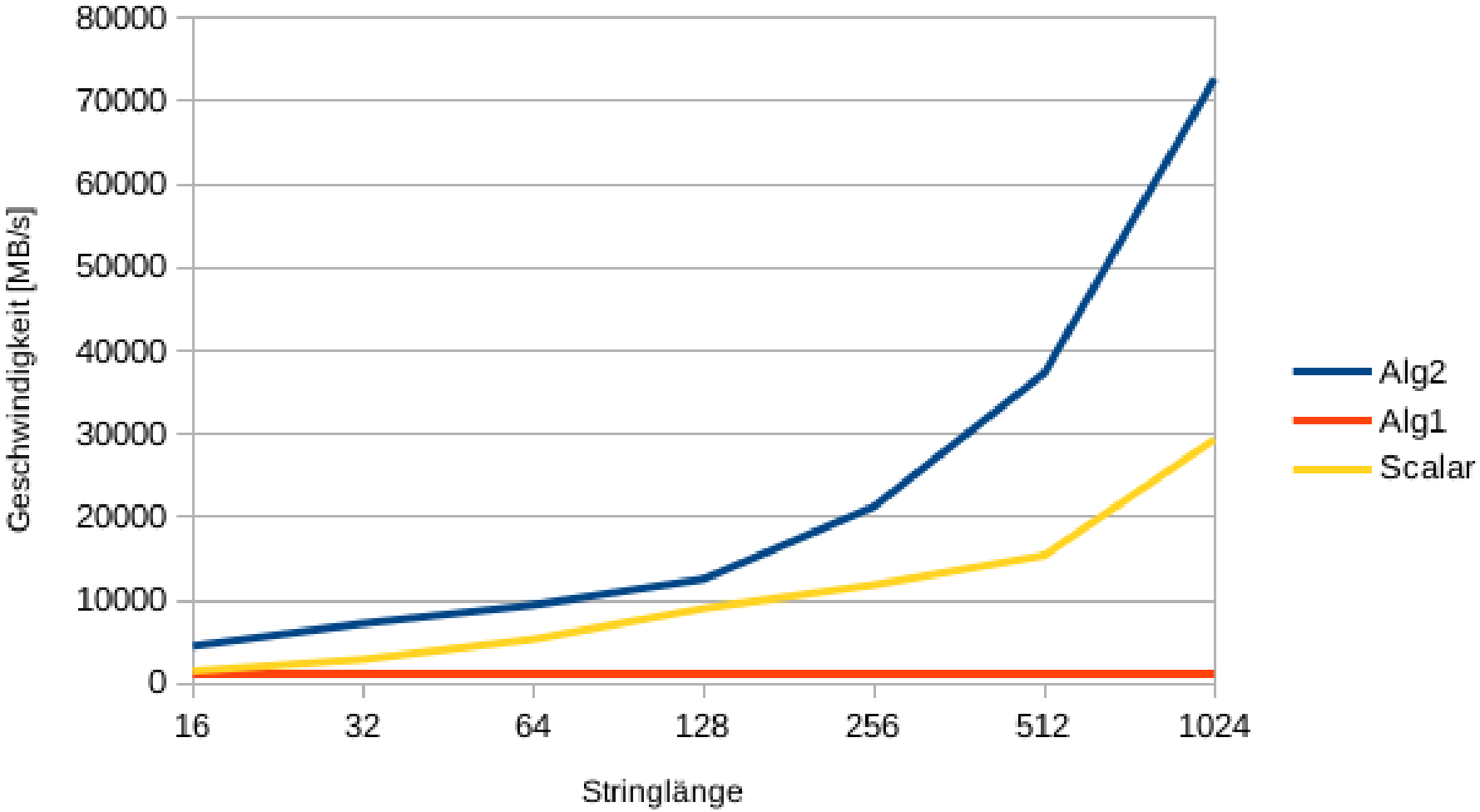


[1]

Benchmarks

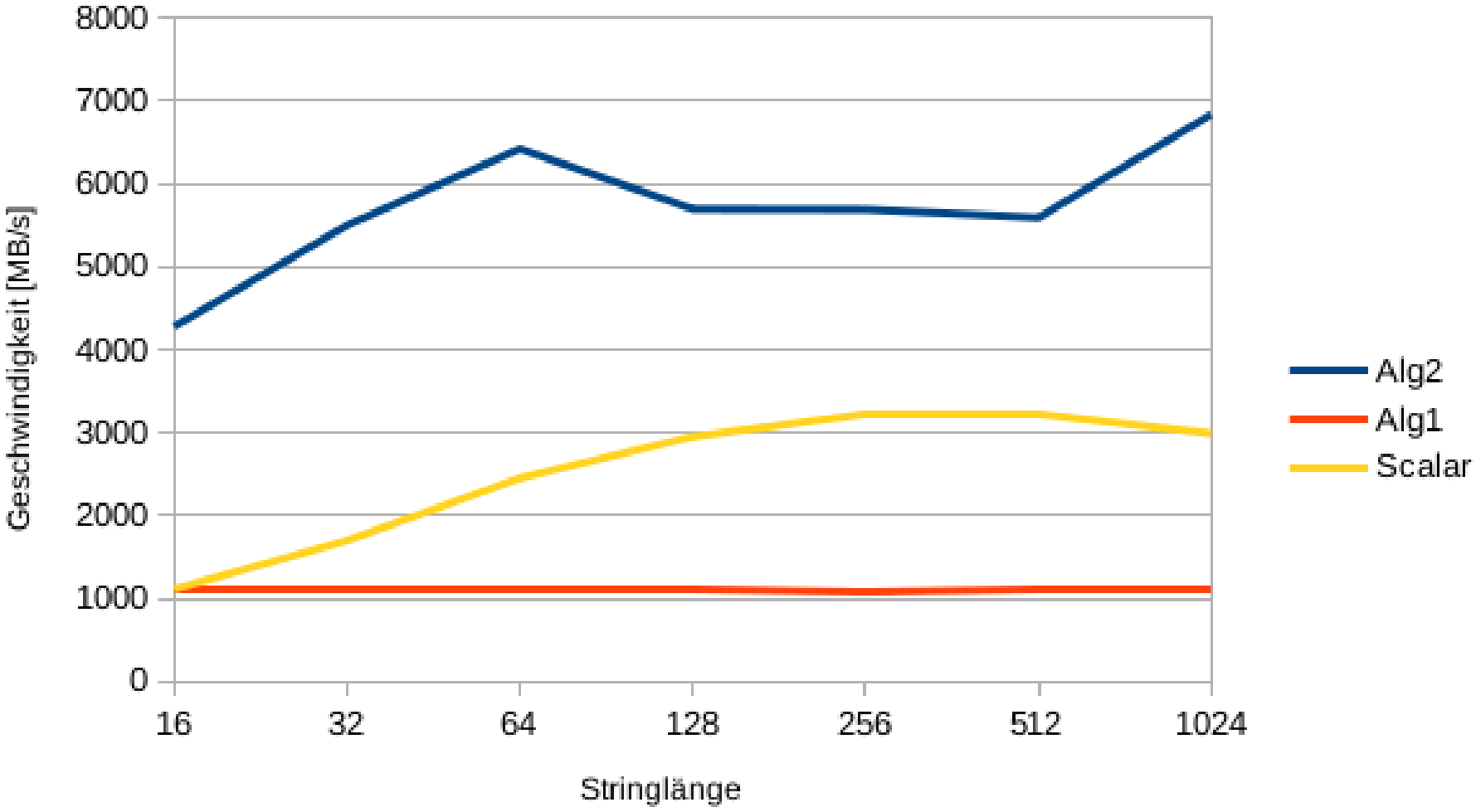
- Eigene Tests durchgeführt mit der Google-Benchmark-Library aus dem Template
- Getestet in einer Linux-VM mit 3GB RAM und 2 Haswell-Kernen
- Geschwindigkeit bezieht sich auf die gesamte Stringlänge (auch wenn vorzeitig abgelehnt wird)

Benchmarks



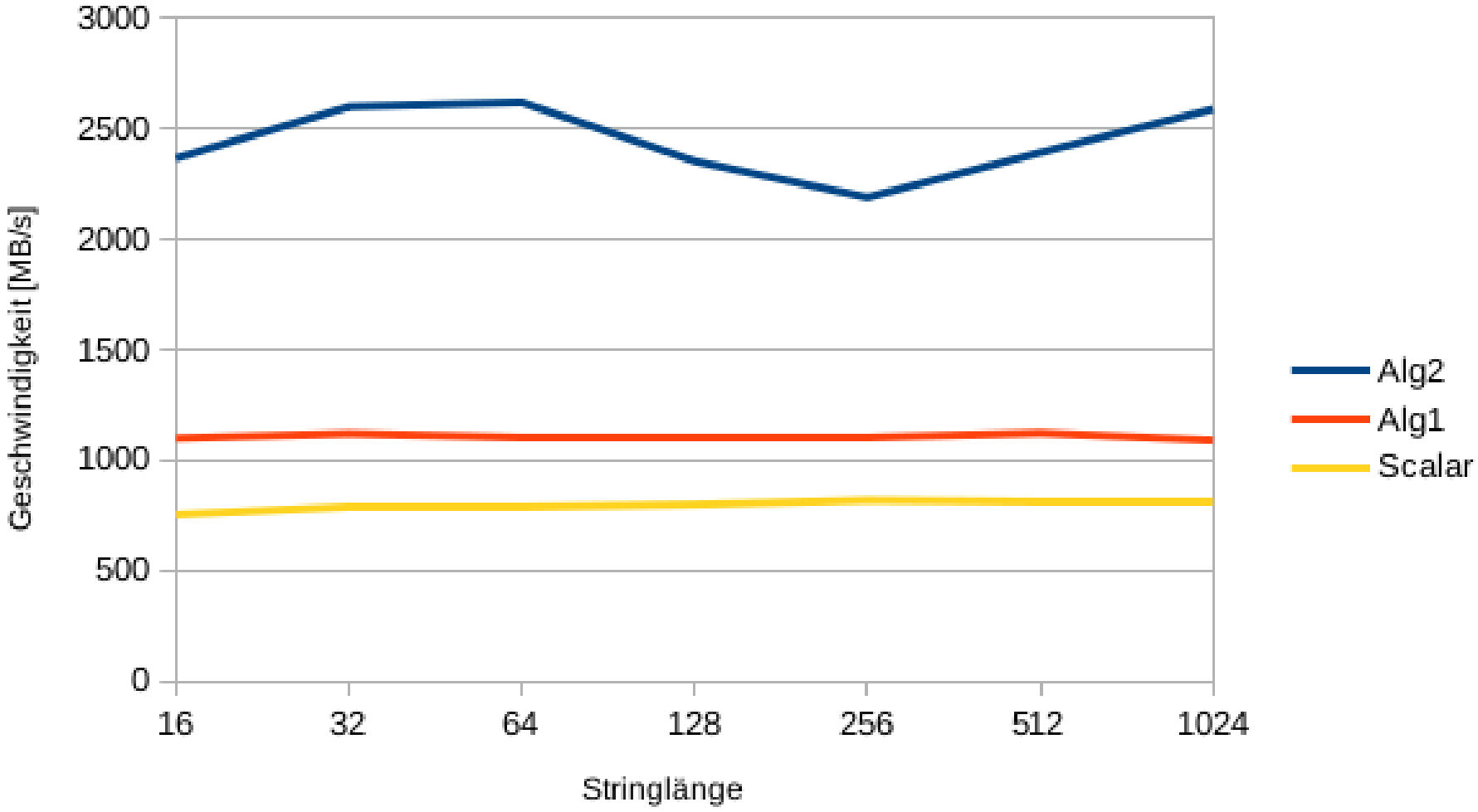
Durchschnittlich 0% bearbeitet

Benchmarks



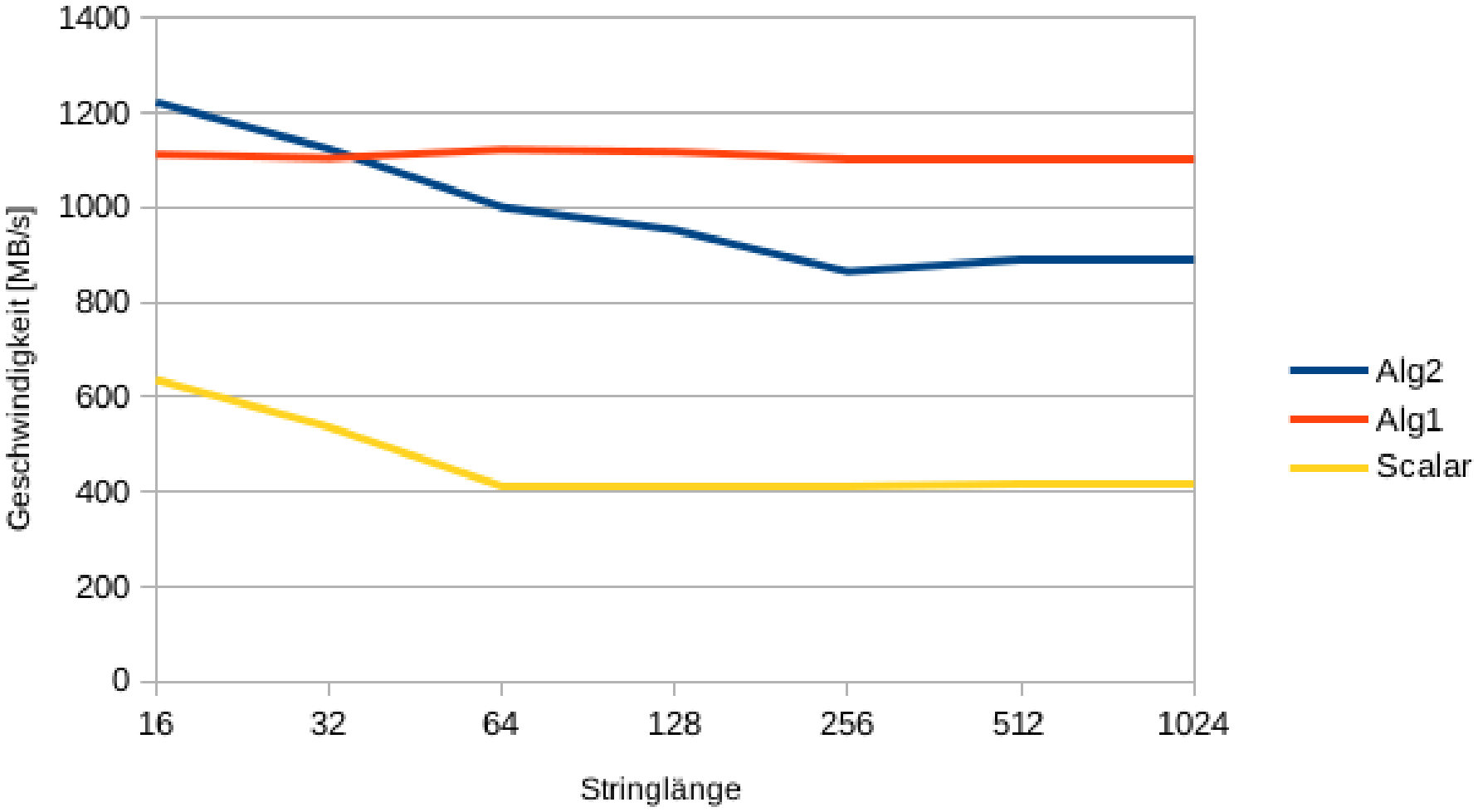
Durchschnittlich 10% bearbeitet

Benchmarks



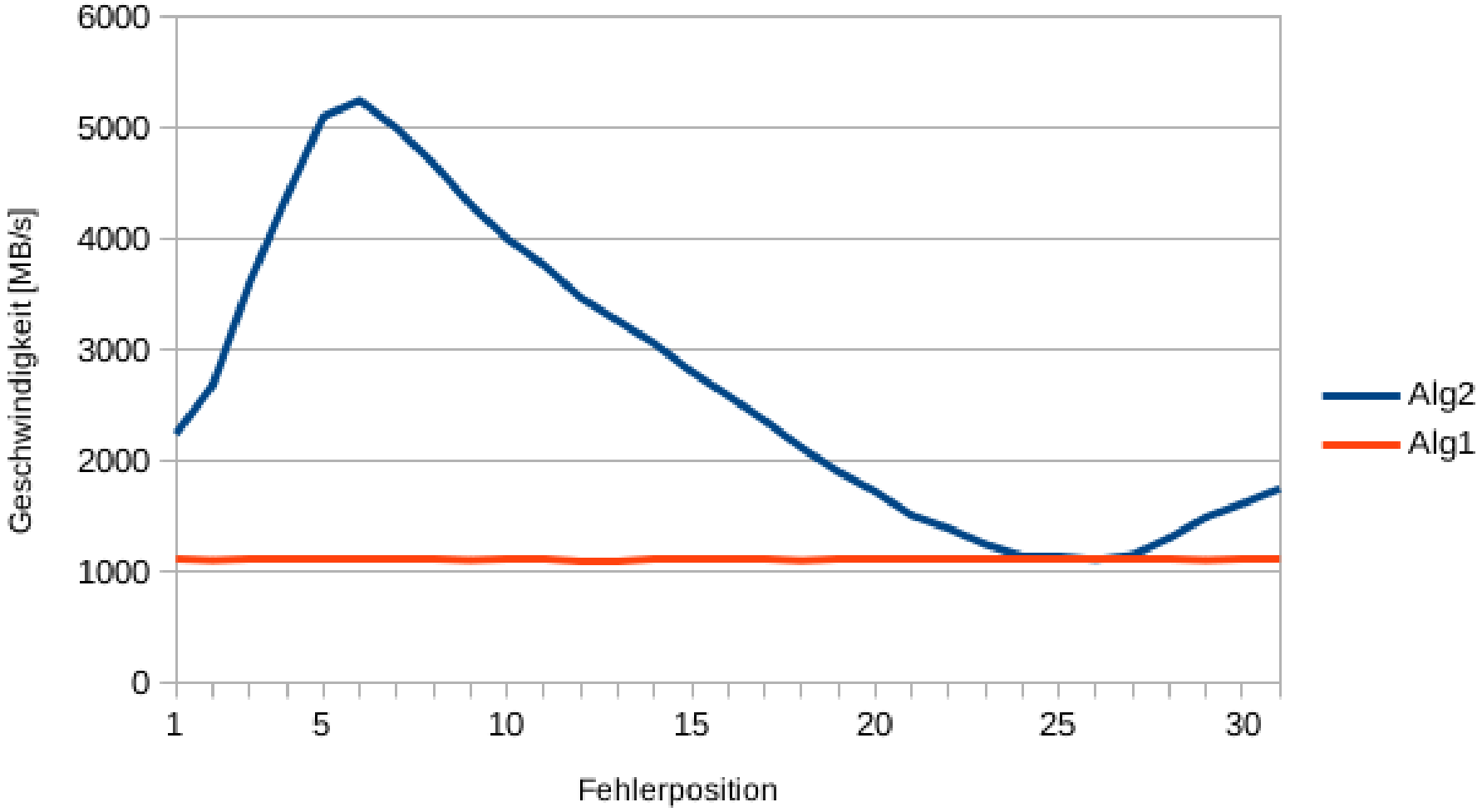
Durchschnittlich 50% bearbeitet

Benchmarks



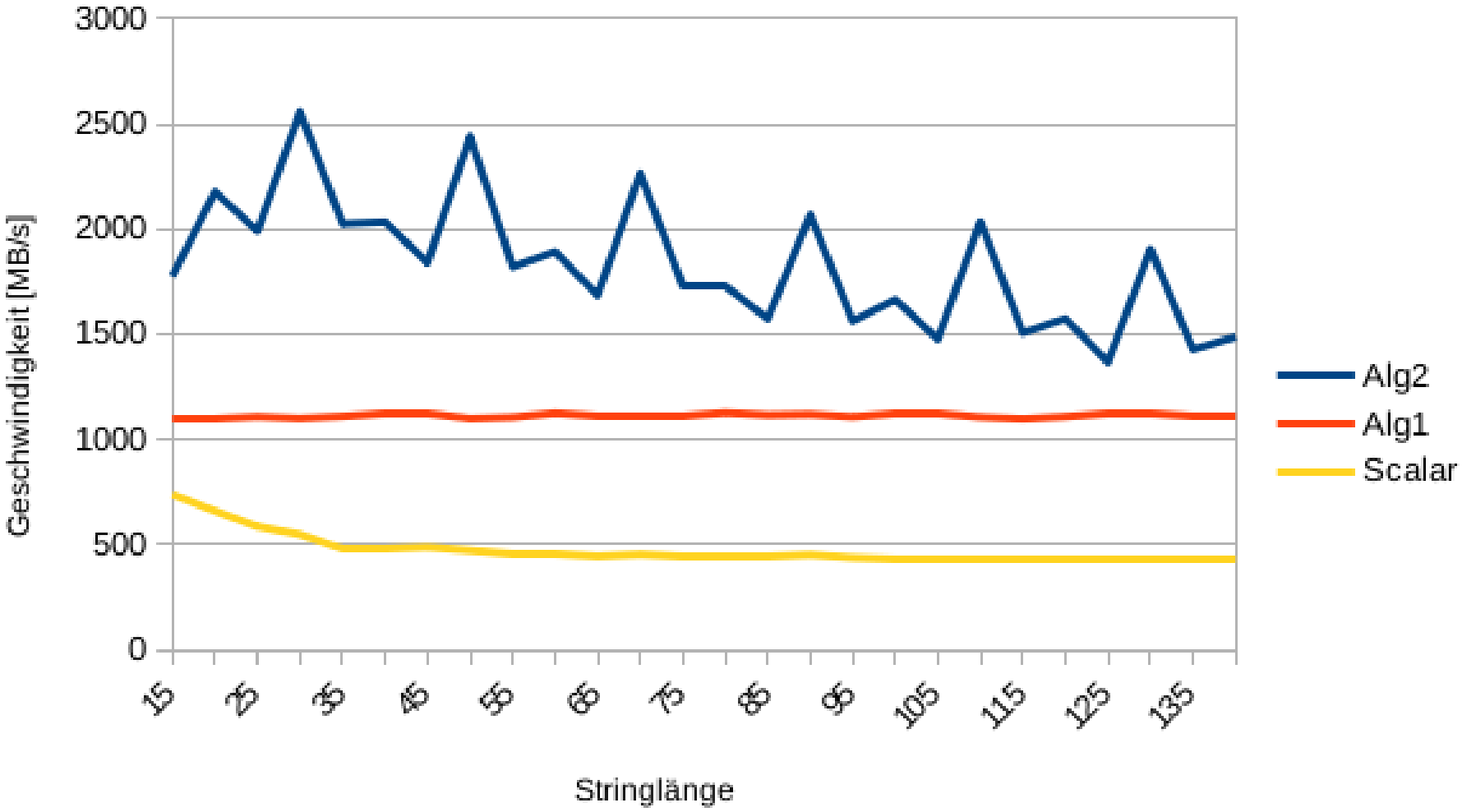
Durchschnittlich 100% bearbeitet

Benchmarks



Stringlänge 32; x = Durchschnittliche Bearbeitungslänge

Benchmarks



ungerade Stringlängen

Zusammenfassung

- Algorithmus 1 ermöglicht es, gleichzeitig bis zu 8 Strings zu bearbeiten
- Algorithmus 2 ist durch vorzeitiges Austauschen nicht akzeptierter Strings ca. 2-3 mal schneller als der skalare Algorithmus
- Mit AVX-512 können 16 Strings gleichzeitig betrachtet werden $\rightarrow$ ca 5-6 mal schneller

Quellen

- [1] E. Sitaridi, O. Polychroniou, K. A. Ross. SIMD-Accelerated Regular Expression Matching
- [2] https://software.intel.com/sites/landingpage/IntrinsicsGuide
- [3] https://www7.in.tum.de/um/courses/theo/ss2018/materials/folien1.pdf