TECHNISCHE UNIVERSITÄT DRESDEN

ZENTRUM FÜR INFORMATIONSDIENSTE
UND HOCHLEISTUNGSRECHNEN
PROF. DR. WOLFGANG E. NAGEL

Komplexpraktikum "Paralleles Rechnen"
A - Stringmanipulationen mit Intrinsics

Bengt Lennicke

Dresden, 8. Januar 2024

Inhaltsverzeichnis

1 Aufgabenstellung

Implementieren Sie eine sequentielle und eine SIMD-parallele (mittels Intrinsics für einen Prozessor, der AVX2, AVX und FMA unterstützt) Variante für folgende String-Funktionen:

```
/* turns string "string" (with length len_string) to uppercase */
/* returns 1 if there has been an error, 0 if there has been no error */
int toUppercase(char* string, int len_string)

/* turns string "string" (with length len_string) to lowercase */
/* returns 1 if there has been an error, 0 if there has been no error */
int toLowercase(char* string, int len_string)

/* counts the appearences of character "c" in string "string" */
/* (with length len_string) */
/* returns -1 if there has been an error, and the number of appearences*/
/* if there has been no error */
int countChar(char* string, int len_string, char c)
```

- Beschreiben Sie für diese Funktionen die asymptotische Zeitkomplexität.
- Messen und Vergleichen Sie die Ausführungszeiten für sequentielle und SIMD-parallele Ausführung für Strings der Länge 10.000, 100.000, 1.000.000 und 100.000.000 .
- Nutzen Sie dafür die "romeo"Partition von taurus.
- Führen Sie jeweils 20 Messungen durch und analysieren Sie die Ergbenisse mit geeigneten statistischen Mitteln.

2 Umsetzung

2.1 string_manipulation.c

Das Programm ist aufgeteilt in Dateien für den sequentiellen und parallelen Ansatz und einer 'main'-Datei, welche die Laufzeitmessungen für beide Umsetzungen durchführt. Die 'main'-Datei ist 'string_manipulation.c'. Der Anfang der 'main'-Funktion darin ist im folgenden zu sehen.

```
1   int main()
2   {
3       int len_string;
4       FILE *file;
5
6       init_register();
7
8       // 10000
9       file = fopen("../evaluation/data/string_times_10000.csv", "w");
10      if (measurement(file, 100, 10000))
11      {
12          return 1;
13      }
14      fclose(file);
15
16      // 100.000
17      ...
18  }
```

Zunächst werden einige Register initialisiert (2.1 line 6), welche für die parallelen Berechnungen notwendig sind. Näheres dazu im Kapitel 2.3. Anschließend wird für die jeweiligen Stringlängen (10.000, 100.000,...) eine csv-Datei geöffnet (line 9) und die "measurementFunktion aufgerufen.

```
1  int measurement(FILE *file, int iterations, int len_string)
2  {
3          char *string;
4          int i;
5          string = rand_string_alloc(len_string);
6          fprintf(file,
7          "count_par,count_seq,upper_par,upper_seq,lower_par,lower_seq\n")
8          for(i = 0; i < iterations; i++)
9          {
10
11                 if (measure(file, string, len_string))
12                 {
13                         return 1;
14                 }
15         }
16         free(string);
17         return 0;
18 }
```

In der 'measurement'-Funktion wird ein zufälliger String mit gegebener Länge wird erstellt. Das Ergebnis der Messung wird in Form einer csv-Datei festgehalten. Daher wird die Header-Zeile mit den Spaltennamen in die übergebene Datei geschrieben (line 6-7). Die Spaltenamen sind in der Reihenfolge der Messungen in der 'measure' Funktion. Anschließend wird 'iterations'-mal mit dem gebildeten String die 'measure'-Funktion aufgerufen.
Die 'measure'-Funktion führt für den gegebenen String die sequentiellen und parallelen Funktionen für 'toUppercase', 'toLowercase' und 'countChar' aus und schreibt die gemessenen Laufzeiten in die gegebene Datei im csv Format.

```
1  int measure(FILE *file, char *string, int len_string)
2  {
3          int par_count, seq_count;
4          struct timespec start, end;
5
6          // string gets changed so we work with duplicates to be able to
7          char *seq_string, *par_string;
8          seq_string = malloc(len_string * sizeof(char));
9          par_string = malloc(len_string * sizeof(char));
10         strncpy(seq_string, string, len_string);
11         strncpy(par_string, string, len_string);
```

Da die jeweiligen Stringverarbeitungsfunktionen mit Pointern auf den String arbeiten und den gegebenen String inplace verarbeitet wird, wird in line 8-11 der String in neu-allokiertem Speicher kopiert. Es wird für sequentiell und parallel jeweils ein String vorbereitet, damit die Ergebnisse der Funktionen vergleicht werden können, um die Korrektheit der Ansatze zu versichern.

```
1          // start with count as no reset necessary after
2          // count
3          clock_gettime(CLOCK_MONOTONIC, &start);
4          par_count = countCharPar(par_string, len_string, 'c');
5          clock_gettime(CLOCK_MONOTONIC, &end);
```

```
6                   fprintf(file, "%d,", time_diff_in_ns(start, end));
7
8                   clock_gettime(CLOCK_MONOTONIC, &start);
9                   seq_count = countCharSeq(seq_string, len_string, 'c');
10                  clock_gettime(CLOCK_MONOTONIC, &end);
11                  fprintf(file, "%d,", time_diff_in_ns(start, end));
12                  if (par_count != seq_count)
13                  {
14                          fprintf(stderr, "Counting␣does␣not␣match␣up.\n");
15                          return 1;
16                  }
```

Die erste Messung ist für die 'countChar' Funktion. Diese Funktion verändert den übergebenen String nicht, daher können 'seq_string' und 'par_string' für die Messung danach nochmal verwendet werden. Die Zähl-Messung zu Beginn spart daher einmal das erneuern der Strings.
Für die Messung der Zeit wird mit der 'clock_gettime'-Funktion aus der 'time.h' Bibliothek vor und nach dem Funktionsaufruf verwendet. Ich habe die clockid 'CLOCK_MONOTONIC' gewählt, weil ich diese im Zusammenhang mit 'Laufzeitmessungen in C' am meisten gefunden habe und es funktioniert hat. Inhaltlich würden hier auch andere möglich seien.
Der Unterschied zwischen den beiden Zeitstempeln in Nanosekunden wird dann in die übergebene Datei geschrieben, jeweils mit einem Komma dazu für das csv-Format.
Es wird zunächst die Laufzeit der 'countCharPar'-Funktion in line 3-6 gemessen und geschrieben; analog in line 8-11 für die Funktion 'countCharSeq'.
Anschließend werden die Ergebnisse verglichen und die Funktion bricht ab, wenn beide Ansätze nicht zum gleichen Ergebnis gekommen sind.

```
1                   // uppercase
2                   clock_gettime(CLOCK_MONOTONIC, &start);
3                   toUppercasePar(par_string, len_string);
4                   clock_gettime(CLOCK_MONOTONIC, &end);
5                   fprintf(file, "%d,", time_diff_in_ns(start, end));
6
7                   clock_gettime(CLOCK_MONOTONIC, &start);
8                   toUppercaseSeq(seq_string, len_string);
9                   clock_gettime(CLOCK_MONOTONIC, &end);
10                  fprintf(file, "%d,", time_diff_in_ns(start, end));
11                  if (strcmp(par_string, seq_string))
12                  {
13                          fprintf(stderr, "toUppercase␣does␣not␣match␣up.\n");
14                          return 1;
15                  }
16
17                  // reset
18                  strncpy(seq_string, string, len_string);
19                  strncpy(par_string, string, len_string);
```

Die Messung für die 'toUppercase' Funktionen verläuft analog zum Fall darüber. Allerdings werden hier die übergebenen Strings verändert. Damit die 'toLowercase' Funktionen nicht mit reinen Großbuchstaben-Strings arbeiten sondern ebenfalls mit dem in der 'measurement'-Funktion Bestimmten, werden 'seq_string' und 'par_string' zum Schluss resettet.

```
1                   // lowercase
2                   clock_gettime(CLOCK_MONOTONIC, &start);
3                   toLowercasePar(par_string, len_string);
```

```
4                   clock_gettime(CLOCK_MONOTONIC, &end);
5                   fprintf(file, "%d,", time_diff_in_ns(start, end));
6
7                   clock_gettime(CLOCK_MONOTONIC, &start);
8                   toLowercaseSeq(seq_string, len_string);
9                   clock_gettime(CLOCK_MONOTONIC, &end);
10                  fprintf(file, "%d\n", time_diff_in_ns(start, end));
11                  if (strcmp(par_string, seq_string))
12                  {
13                          fprintf(stderr, "toLowercase␣does␣not␣match␣up.\n");
14                          return 1;
15                  }
16
17                  free(par_string);
18                  free(seq_string);
19                  return 0;
20          }
```

Anschließend findet noch die Messung für die 'toLowercase' Funktionen statt. Hier wird in line 10 anstelle des Kommas ein Zeilenumbruch in die csv-Datei geschrieben, da die Ergebnisse der nächsten Messung in die nächste Zeile gehören.
Die Ergebnisse sind in Form von csv-Dateien im 'evaluation/data' Ordner in 'Aufgabe_A' zu finden und können mit einem Python-Skript 'evaluation.py' (2.5) ausgewertet werden.

2.2 string_manipulation_seq.c

Die Funktionen 'countCharSeq', 'toUppercaseSeq' und 'toLowercaseSeq' sind in 'string_manipulation_seq.c' definiert.

```
1           int toUppercaseSeq(char *string, int len_string)
2           {
3                   while(*string)
4                   {
5                           *string = toupper(*string);
6                           *string++;
7                   }
8                   return 0;
9           }
```

Im sequentiellen Ansatz wird der String sowie die Stringlänge übergeben. Die Übergabe der Stringlänge war in der Aufgabenstellung gefordert, wird aber nicht benötigt.
In der Funktion gibt es einen while-loop, welcher durchläuft solange der Pointer '*string' nicht '\0' ist. Dieses Zeichen markiert das Ende von Strings, sodass der Loop arbeitet bis der Pointer auf das Ende des Strings zeigt.
Im Loop wird das Zeichen worauf der Pointer aktuell zeigt ersetzt, durch das Ergebnis der 'toupper'-Funktion aus der 'ctype.h' Bibliothek (line5). Diese Funktion nimmt ein Zeichen und wenn es ein lowercase Charakter ist, dann wird der passende uppercase Charakter zurückgegeben. Anschließend wird der Pointer auf das nächste Zeichen im String bewegt und der Loop beginnt erneut (line 6).
Die Funktion 'toLowercaseSeq' funktioniert exakt analog und unterscheidet sich nur in line 5; dort wird 'tolower' anstelle von 'toupper' verwendet.

```
1           int countCharSeq(char *string, int len_string, char c)
2           {
3                   int count = 0;
```

```
4                   while(*string)
5                   {
6                           if (strncmp(string, &c, 1) == 0)
7                           {
8                                   count++;
9                           }
10                          *string++;
11                  }
12                  return count;
13          }
```

Die 'countCharSeq'-Funktion arbeitet mit dem gleichen Loop-System wie die anderen beiden Funktionen. Anstelle von 'toupper' und 'tolower' wird allerdings überprüft, ob das Zeichen am Ziel des aktuellen Pointers dem übergebenen, zu zählendem Zeichen entspricht. Falls 'strncmp' aus 'string.h' eine Übereinstimmung feststellt wird ein Zähler um 1 erhöht. Der Wert dieser Variable ist am Ende des Loops die Anzahl wie oft das gesuchte Zeichen im gegebenen String vorkommt und wird als Ergebnis zurückgegeben.

2.3 string_manipulation_par.c

Die Funktionen 'countCharPar', 'toUppercasePar' und 'toLowercasePar' sind in 'string_manipulation_par.c' definiert.
Die parallele Umsetzung erfolgt mit SIMD dh. unter der Verwendung von Vektorregistern. Für die jeweiligen Funktionen sind 5 Register mit bestimmten konstanten Werten notwendig. Daher sind diese zu Beginn des Programms global definiert.

```
1           __m256i upper_low_limit;
2           __m256i lower_low_limit;
3           __m256i upper_up_limit;
4           __m256i lower_up_limit;
5           __m256i register_of_32;
```

Diese globalen Variablen sind nicht im Header-File definiert, da Register nicht auf diese Weise deklariert werden können, sondern direkt definiert werden, also Speicher besetzt wird. Da das Header-File sowohl in 'string_manipulation_par.c' als auch in 'string_manipulation.c' importiert wird, würde es dadurch zu Problemen kommen.
Um vor den Berechnungen die richtigen Werte in diese Register zu kommen wird dir 'init_register()'-Funktion verwendet.

```
1           void init_register()
2           {
3                   // register with chars '<' than a
4                   lower_low_limit = _mm256_set1_epi8('`');
5                   // register with chars '>' than z
6                   upper_low_limit = _mm256_set1_epi8('{');
7                   // register with chars '<' than A
8                   lower_up_limit = _mm256_set1_epi8('@');
9                   // register with chars '>' than Z
10                  upper_up_limit = _mm256_set1_epi8('[');
11                  // register with the 8-bit values '32'
12                  register_of_32 = _mm256_set1_epi8(' ');
13          }
```

Um später Zeichen finden zu können, welche zu den Groß- bzw. Kleinbuchstaben gehören, sind für den Vergleich die Grenzen im Zahlenraum der ASCII Zeichen notwendig. Im Detail: Kleinbuchstaben gehen von 96-123 (in Zeichen '‘' bis '{') und Großbuchstaben von 64-91 ('@' bis '[').
Ein 'char' in C ist 8 Bits groß. Daher wird im gesamten Programm in den Registern mit 8Bit Bereichen gearbeitet.
Das Verarbeiten der Strings mit Vektorregistern teilt sich in 2 Schritte: Ein-/Auslesen des Strings in die Register und das Verarbeiten der einzelnen Register.
Das Ein- und Auslesen in die Register läuft für 'lowercase', 'uppercase' und 'countChar' analog. Im folgenden wird beispielhaft die Umsetzung für 'toLowercase' betrachtet.

```
1   int toLowercasePar(char *string, int len_string)
2   {
3       int i, filler_size;
4       char *filler_string;
5       __m256i xmm;
6
7       // a register can hold 32 chars (8bit ints)
8       // so we work on 32 chars of the string at a time
9       for ( i=0; i<=len_string-32; i+=32)
10      {
11          xmm = _mm256_loadu_si256((__m256i*) string);
12          regToLowercase(&xmm);
13          _mm256_storeu_si256((__m256i*) string, xmm);
14          string+=32;
15      }
16
17      // to avoid naughty memory access last chars treated different
18      filler_size = len_string % 32;
19      if (filler_size != 0)
20      {
21          filler_string = (char*) malloc(32*sizeof(char));
22          // remaining chars into allocated 32 bytes memory
23          strncpy(filler_string, string, filler_size);
24          xmm = _mm256_loadu_si256((__m256i*) filler_string);
25          regToLowercase(&xmm);
26          _mm256_storeu_si256((__m256i*) filler_string, xmm);
27          // load the relevant chars back into original string
28          strncpy(string, filler_string, filler_size);
29          free(filler_string);
30      }
31      return 0;
32  }
```

In line 9-15 wird der String in 32-Char Schritten durchgegangen, solange noch 32 Zeichen im String sind. Anschließend werden die restlichen Zeichen bearbeitet (line 18-30). Der Grund für die Aufteilung ist, dass am Ende des Strings ansonsten die '__m256_loadu_si256()'-Funktion auf unallokierten Speicher zugreift bzw. versucht zuzugreifen. Diese Funktion läd 256 Bit, auf die der Pointer '*string' im Argument zeigt, in ein Register. Dieses Register wird dann an die Funktion 'regToLowercase()' übergeben. Die 256 Bits am Ziel von '*string' werden mit '__mm256_store_si256()' mit dem verarbeitetem Inhalt des Registers überschrieben.
Für die restlichen Zeichen werden nochmal 32 Bytes allokiert. Ansonsten ist die Verarbeitung analog.

```
1   int regToLowercase(__m256i *string)
```

```
2           {
3           __m256i is_lower_char = _mm256_and_si256(
4           _mm256_cmpgt_epi8(*string, lower_up_limit),
5           _mm256_cmpgt_epi8(upper_up_limit, *string));
6
7           *string = _mm256_add_epi8(*string,
8           _mm256_and_si256(register_of_32, is_lower_char));
9           return 0;
10          }
```

Die 'regToLowercase' Funktion übernimmt einen Pointer auf ein Register. Jeweils 8 Bit große Bereiche werden verglichen mit den Grenzen für Großbuchstaben verglichen, sodass am Ende 'is_lower_char' Einsen hat wo die 8 Bit zu einem Großbuchstaben korrespondieren und Nullen wo nicht.

Dieses Register wird anschließend 'und' verknüpft. Das daraus resultierende Register wird auf das übergebene Register mit String addiert in 8 Bit Blöcken (d.h. ersten 8 Bit auf die ersten 8 Bit usw.). Somit werden nur die 8 Bit Bereiche, welche einen Großbuchstaben beinhalten, um 32 erhöht. Das entspricht einer Umwandlung von Groß- zu Kleinbuchstaben.

Die Umwandlung zu Uppercase funktioniert analog.

Beim Zählen der Zeichen in einem String gibt es einige Unterschiede. Die Funktion, welche das einzelne Register verabreitet, gibt eine Zahl zurück. Diese Zahlen werden aufaddiert anstatt, dass das Register zurück in den String geschrieben wird. Dementsprechen sieht auch 'regCountChar' anders aus als beim Umwandeln.

```
1           int regCountChar(__m256i *string, char c)
2           {
3                   __m256i char_register = _mm256_set1_epi8(c);
4                   return bitCount(_mm256_movemask_epi8(_mm256_cmpeq_epi8(*string,
5           }
```

Zunächst wird ein Register gefüllt mit den gesuchten Zeichen. Anschließend wird das Register des Eingabestrings damit verglichen. Der 8-Bit-Vergleich ergibt ein Register mit Einsen bei Gleichheit und Nullen sonst. Mit '__mm256_movemask_epi8()' wird aus den ersten Bits der 8 Bit Blöcke eine Zahl gebildet (diese Funktion gewählt, da theoretisch schneller seien sollte als die '__mm256_extract..' Funktionen). Die Einsen in der Binärdarstellung werden gezählt in der Funktion 'bitCount' und das Ergebnis wird von 'regCountChar' zurückgegeben.

Die bitCount Funktion habe ich so wie sie ist im Internet gefunden bei der Suche nach einer effizienten Möglichkeit aus dem extrahierten Integer die Anzahl der Einsen zu gewinnen. Dabei bin ich auf den 'MIT HAKMEM Count' [1] gestoßen. Hier ist die Bearbeitung in konstanter Zeit möglich und hat daher keinen entscheidenden Einfluss auf die Zeitkomplexität.

```
1  int bitCount(unsigned int u)
2  {
3           unsigned int uCount;
4           uCount = u - ((u >> 1) & 033333333333) - ((u >> 2) & 011111111111);
5           return ((uCount + (uCount >> 3)) & 030707070707) % 63;
6  }
```

Ich empfehle die Erklärung im verlinkten Blog-Eintrag [1]. Die Funktion basiert auf Bit-Shifts. Eine 32 bit Zahl n kann wie folgt dargestellt werden: $n = a_{31} * 2^{31} + a_{30} * 2^{30} + ... + a_0$
Wobei 'bitCount' $a_{31} + a_{30} + ... + a_0$ berechnen soll. Der nächste Schritt ist demnach das Entfernen der

Zweierpotenzen.

$$n = a_{31} * 2^{31} + a_{30} * 2^{30} + ... + a_0 \quad (1)$$
$$n >> 1 = a_{31} * 2^{30} + a_{30} * 2^{29} + ... + a_1 \quad (2)$$
$$... \quad (3)$$
$$n >> 31 = a_{31} \quad (4)$$
$$(5)$$

Mit $2^k - 2^{k-1} = 2^{k-1}$ folgt:

$$n - (n >> 1) - (n >> 2) - ... - (n >> 31) = a_{31} + a_{30} + ... + a_0 \quad (6)$$

Da z.B. für die Faktoren von a_{31} gilt damit

$$a_{31} * (2^{31} - 2^{30} - ... - 2^0) = a_{31} \quad (7)$$

.

Das kann durch die Verwendung von Bit-Masken noch optimiert werden, da dadurch erst 3er Blöcke zusammengefasst und dann aufaddiert werden können. Damit wird die Bearbeitung weiter parallelisiert. In 2.3 in line 4 werden zunächst die Bits in 3er Blöcken (octal) gezählt. Die Ergebnisse werden dann in line 5 zusammengezählt.
Zur Veranschaulichung seien die Zahlen erneut als Polynom dargestellt:

$$\begin{aligned} u &= a_0 * 2^0 + a_1 * 2^1 + a_2 * 2^2 + a_3 * 2^3 + a_4 * 2^4 + a_5 * 2^5 \ ... \\ u >> 1 &= a_1 * 2^0 + a_2 * 2^1 + a_3 * 2^2 + a_4 * 2^3 + a_5 * 2^4 + a_6 * 2^5 \ ... \\ \& \ 033333333333 &= a_1 * 2^0 + a_2 * 2^1 + \quad a_4 * 2^3 + a_5 * 2^4 + \quad ... \end{aligned}$$

$$\begin{aligned} u >> 2 &= a_2 * 2^0 + a_3 * 2^1 + a_4 * 2^2 + a_5 * 2^3 + a_6 * 2^4 + a_7 * 2^5 \ ... \\ \& \ 011111111111 &= a_2 * 2^0 + \quad a_5 * 2^3 + \quad ... \end{aligned}$$

Damit ergibt sich in Kombination:

$$\begin{aligned} u - ((u >> 1)\&033333333333) - ((u >> 2)\&011111111111) = \\ a_0 * 2^0 + a_1 * (2^1 - 2^0) + a_2 * (2^2 - 2^1 - 2^0) \\ a_3 * 2^3 + a_4 * (2^4 - 2^3) + a_5 * (2^5 - 2^4 - 2^3) \\ ... \\ = (a_0 + a_1 + a_2) * 2^0 + (a_3 + a_4 + a_5) * 2^3 + ... \end{aligned}$$

Es sind also immer 3 Bits aufaddiert. Die Variable 'uCount' beinhaltet also die Information der "gezählten"Bits und muss anschließend in eine Integer zahl umgewandelt werden. Das passiert in Zeile 5, für welche eine ähnliche Visualisierung möglich ist:

$$\begin{aligned} uCount &= a_0 * 2^0 + a_1 * 2^1 + a_2 * 2^2 + a_3 * 2^3 + a_4 * 2^4 + a_5 * 2^5 + \ ... \\ uCount >> 3 &= a_3 * 2^0 + a_4 * 2^1 + a_5 * 2^2 + a_6 * 2^3 + a_7 * 2^4 + a_8 * 2^5 + \ ... \end{aligned}$$

$$uCount + (uCount >> 3) = (a_0 + a_3) * 2^0 + (a_1 + a_4) * 2^1 + (a_2 + a_5) * 2^2 + (a_3 + a_6) * 2^3 + (a_4 + a_7) * 2^4 + (a_5 + a_8) * 2^5 + (a_6 + a_9) * 2^6 + (a_7 + a_{10}) * 2^7 + ...$$

$$\& \; 030707070707 = (a_0 + a_3) * 2^0 + (a_1 + a_4) * 2^1 + (a_2 + a_5) * 2^2 + (a_6 + a_9) * 2^6 + (a_7 + a_{10}) * 2^7 + ...$$

Der Term $(a_6 + a_9) * 2^6$ entspricht dem LSB vom 2. und 3. Block von uCount also eigentlich zu 2^0. Auch bei den folgenden Termen ist die 2er Potenz zu groß, sie sollte zwischen 0 und 2 liegen. Damit erklärt sich das modulo 63 am Ende. Also:

$$((uCount + (uCount >> 3)) \& \; 030707070707) \% 63$$
$$(a_0 + a_3) * 2^0 + (a_1 + a_4) * 2^1 + (a_2 + a_5) * 2^2 + (a_6 + a_9) * 2^0 + (a_7 + a_{10}) * 2^1 + ...$$

Somit werden von uCount jeweils die 3er Blöcke addiert. Da darin die Anzahl der 1-Bits in den 3er Blöcken vom Argument steht, ist das Endresultat die Summe aller Einsen im Argument.

2.4 Makefile

Die C Anwendung wird mit 'make' und 'gcc' gebaut. Relevant ist hier der Tag '-mavx2'. Dadurch werden die Instruktionen von AVX2 eingeschaltet, d.h. die Arbeit mit Vektorregistern ermöglicht.

```
1  all: string_manipulation
2
3  string_manipulation_par.o: string_manipulation_par.c
4  gcc -mavx2 -c string_manipulation_par.c
5
6  string_manipulation_seq.o: string_manipulation_seq.c
7  gcc -c string_manipulation_seq.c
8
9  string_manipulation.o: string_manipulation.c
10 gcc -mavx2 -c string_manipulation.c
11
12 string_manipulation: string_manipulation.o string_manipulation_par.o str
13 gcc -mavx2 -o string_manipulation string_manipulation.o string_manipulat
14
15 clean:
16 rm string_manipulation_par.o string_manipulation.o string_manipulation_s
```

2.5 evaluation.py

Im Evaluierungsskript werden zunächst die csvs aus der Messung eingelesen. Anschließend werden die Daten in 3 Ergebnissen aufbereitet: Dateien mit Tabellen (zum Kopieren in den Report), Diagramme für die Zeitkomplexität und Diagramme zum Vergleichen der Laufzeiten.

```
1 def main():
2     df_10k = pd.read_csv("./data/string_times_10000.csv")
```

```
3         df_100k = pd.read_csv("./data/string_times_100000.csv")
4         df_1M = pd.read_csv("./data/string_times_1000000.csv")
5         df_100M = pd.read_csv("./data/string_times_100000000.csv")
6
7         create_tabulars(df_10k, df_100k, df_1M, df_100M)
8
9         if not os.path.isdir("../report/images"):
10                os.mkdir("../report/images")
11
12        complex_plots(df_10k, df_100k, df_1M, df_100M)
13
14        comparison_plots(df_10k, 10000)
15        comparison_plots(df_100k, 100000)
16        comparison_plots(df_1M, 1000000)
17        comparison_plots(df_100M, 100000000)
```

Mit 'create_tabulars' wird für jede Funktion ('toUppercase', 'toLowercase', 'countChar') eine Tabelle erstellt. Dabei wird für die einzelnen Stringlängen jeweils eine Zeile geschrieben, welche den Mittelwert und die Standardabweichung für den parallelen und sequentiellen Ansatz beinhaltet.

'complex_plots' erstellt 3 Diagramme jeweils für jede Funktion, die gemessen wurde. Im Diagramm werden die Mittelwerte für parallel/sequentiell über den Stringlängen dargestellt. Dadurch wird das Laufzeitverhalten in Abhängigkeit von der Stringlänge gezeigt d.h. die Zeitkomplexität.

Die 'comparion_plots' erstellt für jede Funktion und jeweils jede Stringlänge ein Diagramm d.h. 12 Diagramme. Hier werden einfach die Laufzeiten für jede Messung dargestellt.

Die Funktionen an sich sind nicht besonders interessant, da es primär plot-shenanigans ist.

3 Auswertung

3.1 Zeitkomplexität

Die sequentiellen Funktionen für die 'uppercase'-, 'lowercase'-Umwandlung von Strings und dem Zählen von bestimmte Buchstaben in einem String sind in der Datei ßtring_manipulation_seq.c".

Die Funktionen heißen "toUppercaseSeq", "toLowercaseSeqünd "countCharSeq". Ich bin hier von den Namen der Aufgabenstellung abgewichen, damit ich den sequentiellen und paralleln Ansatz in einer Main Datei gleichzeitig importieren/nutzen kann.

Alle drei Funktionen arbeiten mit einem while loop, in welchem jedes Zeichen bearbeitet wird und anschließend der Pointer auf das nächste Zeichen bewegt wird. Damit hängt die Bearbeitungszeit linear von der Stringlänge ab. Die asymptotische Zeitkomplexität ergibt sich damit zu: O(n).

Die parallelen Funktionen sind in der Datei ßtring_manipulation_par.cünd heißen "toUppercasePar", "toLowercaseParünd "countCharPar". Die Funktionen laden jeweils 32 Zeichen des Eingabe Strings in ein 256-bit Register und bearbeiten dieses; arbeiten also in 32 Charakter Schritten. Damit hängt die Bearbeitungszeit davon ab wie viele 32-Charakter Blöcke existieren bzw. damit von der Länge des Strings. Die asymptotische Zeitkomplexität ist also ebenfalls: O(n).

Das beschriebene Zeitverhalten ist auch in den folgenden drei Grafiken zu erkennen. Hier wird die durchschnittlich benötigte Rechenzeit in Abhängigkeit von der Stringlänge gezeigt. Die logarithmischen Skalen wurden gewählt, weil sonst die Messpunkte von 10k, 100k und einer Million sehr dicht zusammen liegen im Vergleich zu 100 Millionen.

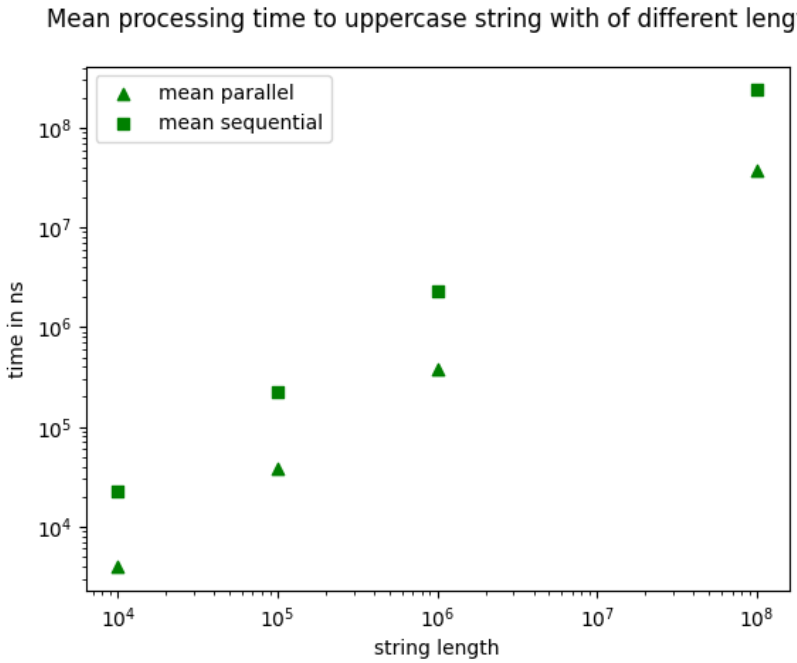

Abbildung 1: Durchschnittliche Durchführungszeit von toUppercase() in Abhängigkeit von der Stringlänge.

3.2 Ausführungszeiten

3.2.1 toUppercase

In der Tabelle 3.2.1 sind die Zeitmessungen für die toUppercase() Funktionen zusammengefasst. Es ist deutlich zu sehen, dass die sequentielle Ausführung in für jede Stringlänge mehr Zeit benötigt als die Umsetzung mit SIMD.

String Länge	parallel in ns		sequentiell in ns	
	Mittelwert	Standardabweichung	Mittelwert	Standardabweichung
10000	3927.74	136.28	22713.70	853.06
100000	38037.63	1796.03	225913.24	1528.48
1000000	376818.89	1996.34	2262666.73	15791.38
100000000	37738184.27	82337.04	239258933.07	14075870.69

3.2.2 toLowercase

In der Tabelle 3.2.2 sind die Zeitmessungen für die toLowercase() Funktionen zusammengefasst. Es ist deutlich zu sehen, dass die sequentielle Ausführung in für jede Stringlänge mehr Zeit benötigt als die Umsetzung mit SIMD.

String Länge	parallel in ns		sequentiell in ns	
	Mittelwert	Standardabweichung	Mittelwert	Standardabweichung
10000	3971.14	458.73	23990.09	474.48
100000	37695.94	294.53	239811.23	1545.62
1000000	378483.88	9016.46	2401516.14	29803.12
100000000	37753168.08	122854.37	247914893.18	10307739.59

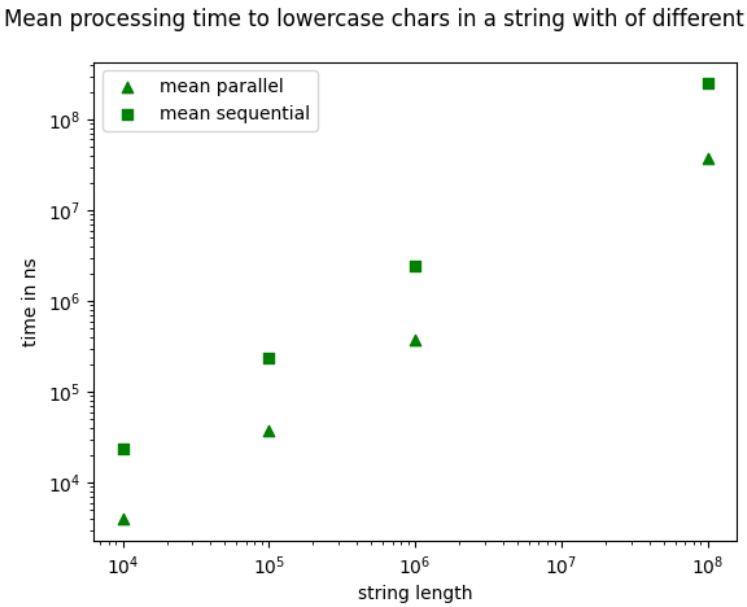

Abbildung 2: Durchschnittliche Durchführungszeit von toLowercase() in Abhängigkeit von der Stringlänge.

3.2.3 countChar

In der Tabelle 3.2.3 sind die Zeitmessungen für die countChar() Funktionen zusammengefasst. Es ist zu sehen, dass die sequentielle Ausführung in für jede Stringlänge mehr Zeit benötigt als die Umsetzung mit SIMD. Hier ist der Unterschied allerdings nicht so deutlich wie bei den vorherigen Funktionen. Hier würde es sich lohnen nach einer effizienteren Lösung zu suchen als dem aktuellen Ansatz.
Desweiteren ist die Standardabweichung auffällig hoch für den sequentiellen Ansatz bei einer Stringlänge von 100 Millionen. Deutlicher zu sehen in Abb. 15.

String Länge	parallel in ns		sequentiell in ns	
	Mittelwert	Standardabweichung	Mittelwert	Standardabweichung
10000	15524.42	374.14	10018.55	724.44
100000	154677.94	1582.74	100867.41	6679.76
1000000	1545701.48	6479.76	1074906.42	297715.72
100000000	154723840.88	219245.90	163612321.04	68948873.63

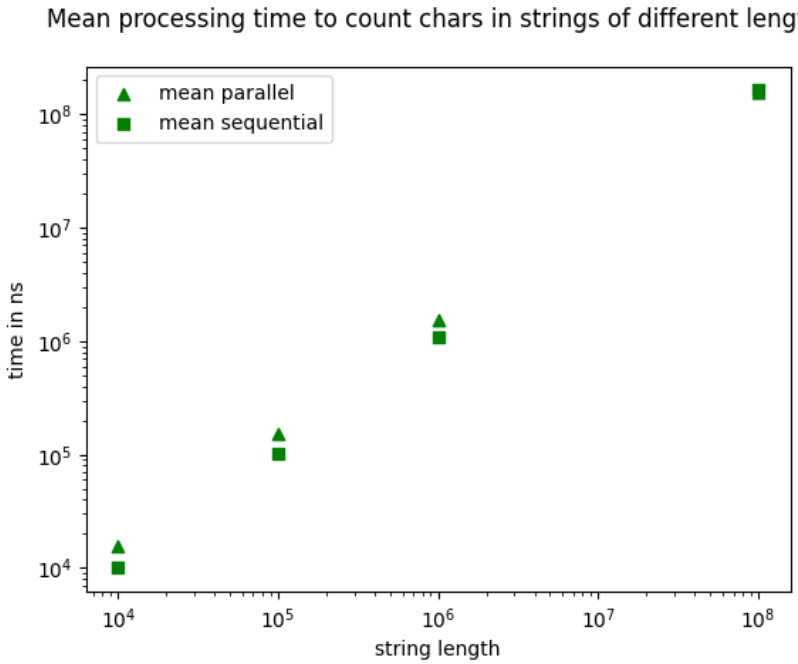

Abbildung 3: Durchschnittliche Durchführungszeit von countChar() in Abhängigkeit von der Stringlänge.

3.3 Vergleich

In den folgenden Abbildungen sind 100 Laufzeiten für das Verarbeiten eines Strings aufgezeigt. Dabei sind pro Funktion 4 Abbildungen für je 10.000, 100.000, 1.000.000 und 100.000.000 Zeichen in den Strings. Die Diagramme veranschaulichen den Vergleich zwischen dem sequentiellen und parallelen Ansatz unter Verwendung von SIMD.

3.3.1 toUppercase

Alle Diagramme zeigen deutlich, dass der parallele Ansatz schneller ist für alle Stringlängen.

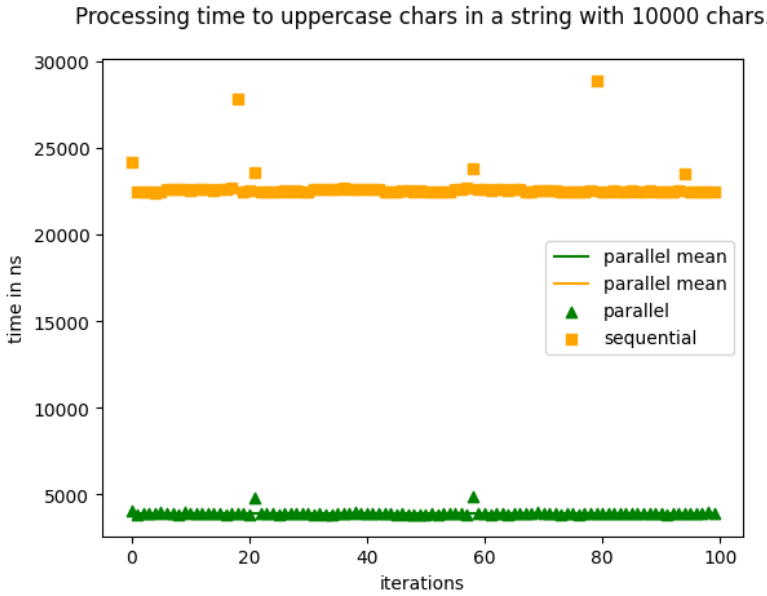

Abbildung 4: Durchführungszeiten von toUppercase für einen String mit 10.000 Zeichen.

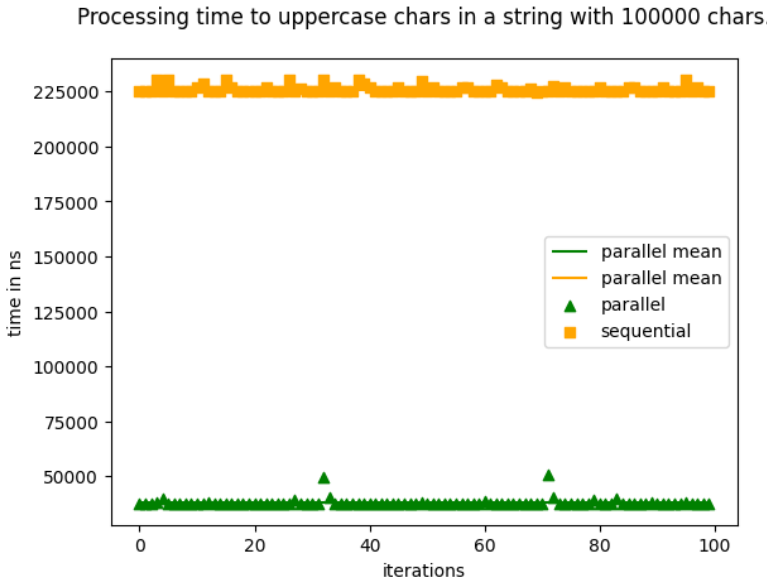

Abbildung 5: Durchführungszeiten von toUppercase für einen String mit 100.000 Zeichen.

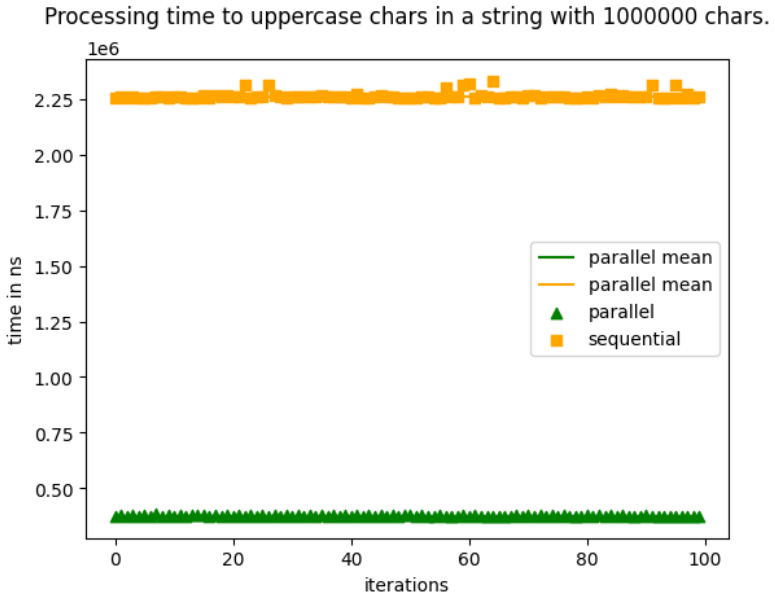

Abbildung 6: Durchführungszeiten von toUppercase für einen String mit 1.000.000 Zeichen.

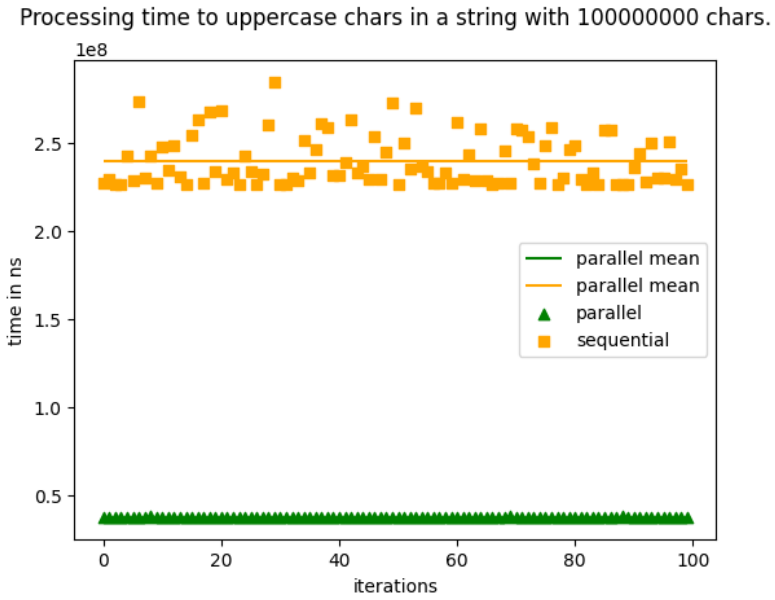

Abbildung 7: Durchführungszeiten von toUppercase für einen String mit 100.000.000 Zeichen.

3.3.2 toLowercase

Alle Diagramme zeigen deutlich, dass der parallele Ansatz schneller ist für alle Stringlängen.

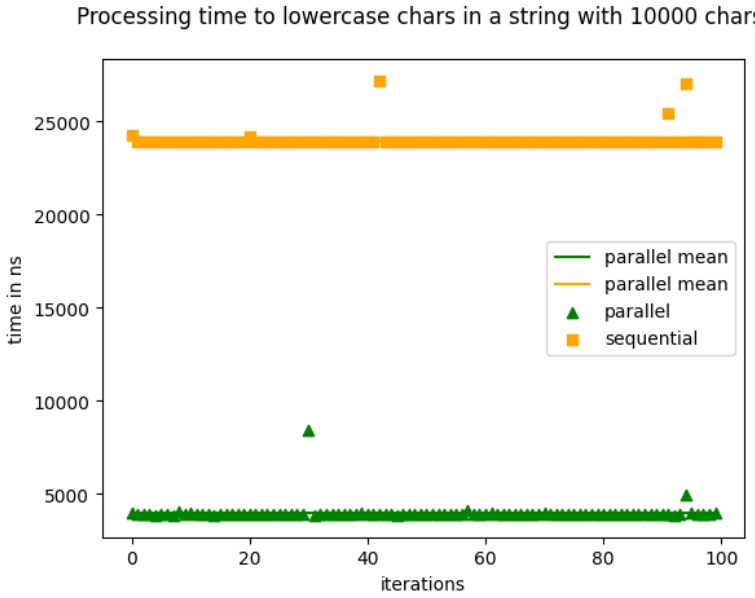

Abbildung 8: Durchführungszeiten von toLowercase für einen String mit 10.000 Zeichen.

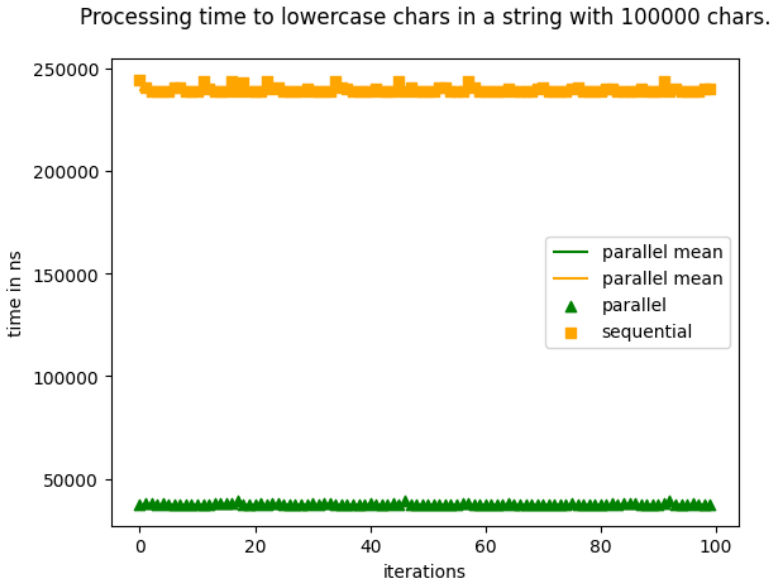

Abbildung 9: Durchführungszeiten von toLowercase für einen String mit 100.000 Zeichen.

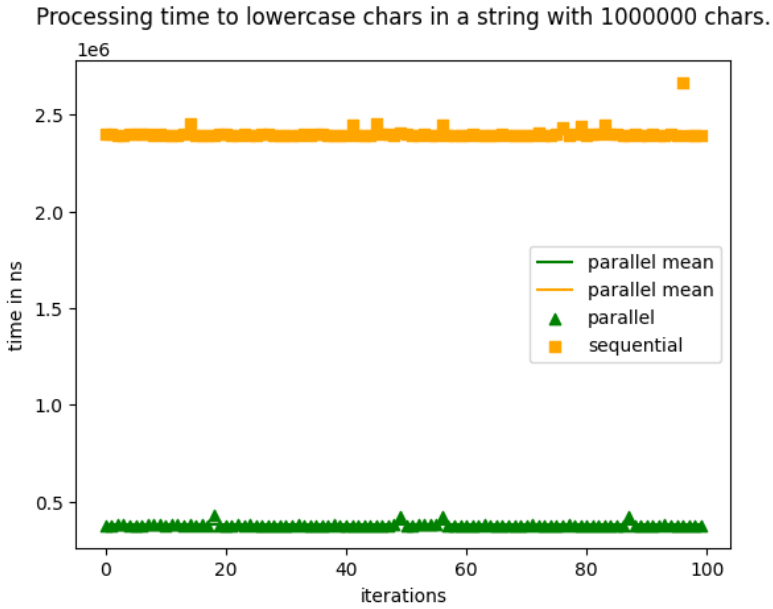

Abbildung 10: Durchführungszeiten von toLowercase für einen String mit 1.000.000 Zeichen.

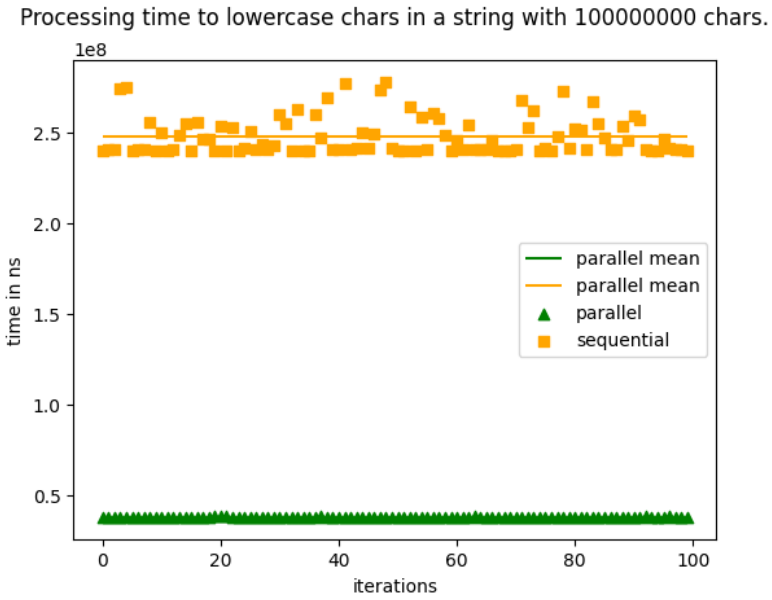

Abbildung 11: Durchführungszeiten von toLowercase für einen String mit 100.000.000 Zeichen.

3.3.3 countChar

Die folgenden Diagramme zeigen jeweils 100 Laufzeiten für einen String mit unterschiedlicher Länge. Gemessen wird die Laufzeit um einen gegebenen Buchstaben ('c') im String zu zählen.
Auch hier ist zu sehen, dass der parallele Ansatz schneller ist, außer bei Strings mit 100 Millionen Zeichen. In Abb. 15 ist ein sehr merkwürdiges Verhalten für die sequentielle Lösung zu sehen. Für den gleichen String scheint die Funktion entweder extrem schnell oder langsam zu sein. Da es der gleiche String ist, kann der Grund nicht sein, dass die Anzahl der Vorkommen von 'c' eine derartigen Unterschied erzeugen. Der Unterschied tritt auch erst mit der hohen Stringlänge auf. Damit sollte der Unterschied auch nicht durch etwaige sonstige Auslastung der Rechnungsnodes verursacht werden. Ich konnte das gleiche Verhalten bei meinem Laptop nicht beobachten und kann es auch nicht sicher erklären.

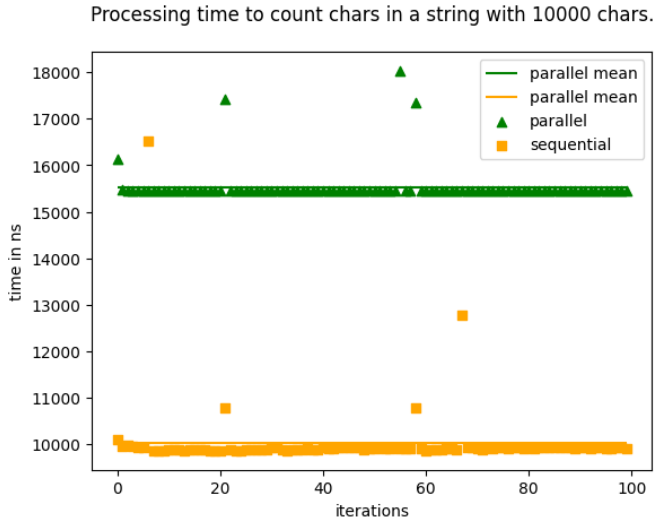

Abbildung 12: Durchführungszeiten von countChars für einen String mit 10.000 Zeichen.

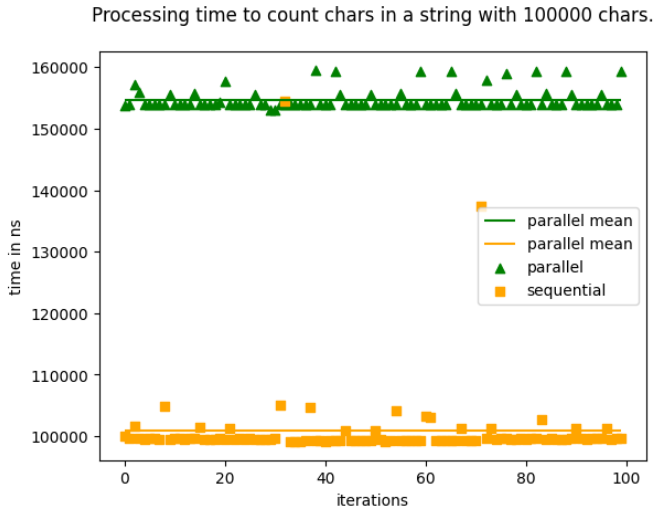

Abbildung 13: Durchführungszeiten von countChars für einen String mit 100.000 Zeichen.

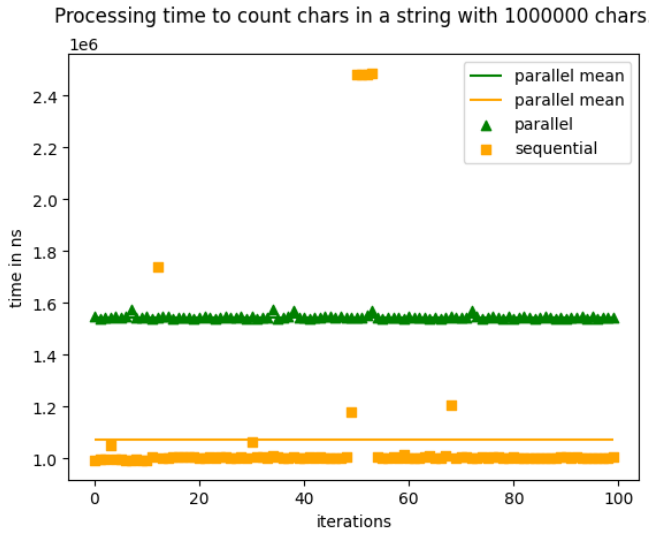

Abbildung 14: Durchführungszeiten von countChars für einen String mit 1.000.000 Zeichen.

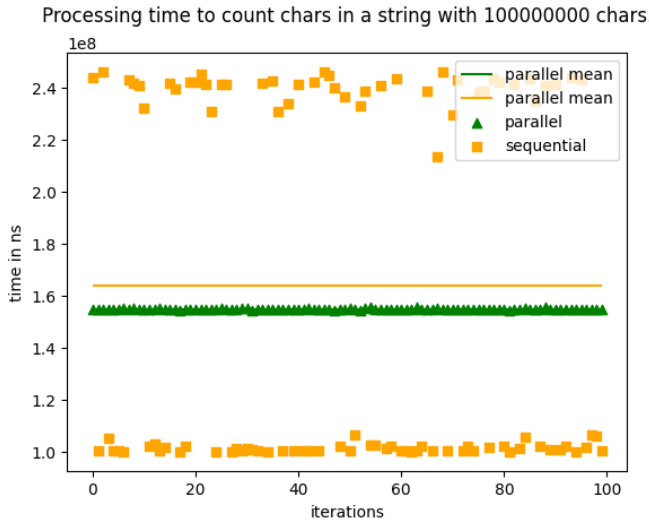

Abbildung 15: Durchführungszeiten von countChars für einen String mit 100.000.000 Zeichen.

Literatur

[1] Jeu George's Blog 'Parallel Counting'
https://web.archive.org/web/20151229003112/http://blogs.msdn.com/b/jeuge/archive/2005/06/08/hakmem-bit-count.aspx