TECHNISCHE UNIVERSITÄT DRESDEN

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

Komplexpraktikum "Paralleles Rechnen"
A - Stringmanipulationen mit Intrinsics

Bengt Lennicke

Dresden, 17. 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)
```

- 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 'measurement'-Funktion 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);
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);
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 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;
6           __m256i c_register;
7           __m256i one_register;
```

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                  // register with the 8 bit values for 'c'
```

```
14                  c_register = _mm256_set1_epi8('c');
15                  // register with the 8 bit value 00000001
16                  one_register = _m256_set1_epi8(1);
17          }
```

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 '['). Für diese Limits werden die ersten 4 Register gesetzt. Zwischen Klein- und zugehörigem Großbuchstaben ist in ASCII ein Abstand von 32. Um später 32 zu addieren/subtrahieren wird ein Register mit dem Wert 32 alle 8 Bit erstellt. Für das Zählen der Vorkommen von 'c' wird ein Vergleich benötigt; diese Funktion wird das 'c_register' erfüllen. Außerdem wird eine Art Bitmaske 'one_register' erstellt, um damit später die LSBs der 8Bit Blöcke eines anderen Registers auslesen zu können.
Ein 'char' in C ist 8 Bits groß. Daher wird fast 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, damit beim Laden in das Register nicht auf unallokierten Speicher zugegriffen wird. 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 mit den Grenzen für Großbuchstaben verglichen, sodass am Ende 'is_lower_char' Einsen beinhaltet in Bereichen, in denen die 8 Bit zu einem Großbuchstaben korrespondierten und ansonsten Nullen.
Dieses Register wird anschließend 'und' verknüpft mit dem 'register_of_32'. Das daraus resultierende Register hat den Wert 32 wo 'is_lower_char' nicht Null war und wird auf das übergebene Register mit String addiert (auch 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.
Das Zählen von 'c' im gegebenen String unterscheidet sich von den anderen beiden Funktionen, da hier der String nicht verändert wird und eine Zahl zurückerwartet wird.

```
1  int64_t countCharPar(char *string, int len_string)
2  {
3          int i, filler_size;
4          char *filler_string;
5          __m256i xmm, counter_epi8, counter_epi64;
6
7          counter_epi8 = _mm256_setzero_si256();
8          counter_epi64 = _mm256_setzero_si256();
9
10         // a register can hold 32 chars (8bit ints)
11         // so we work on 32 chars of the string at a time
12         for ( i=0; i<=len_string-32; i+=32)
13         {
14                 xmm = _mm256_loadu_si256((__m256i*) string);
15                 counter_epi8 = _mm256_add_epi8(counter_epi8,
16                                                regCountChar(&xmm));
17
18                 // after 255 * 32 = 8160 steps the 8 bit could overflow
19                 if (!(i % 8160))
```

```
20                      {
21                              counter_epi64 = merge_epi8_to_epi64(&counter_epi64,
22                              &counter_epi8);
23                              counter_epi8 = _mm256_setzero_si256();
24                      }
25                      string+=32;
26              }
27
28              // to avoid naughty memory access last chars treated different
29              filler_size = len_string % 32;
30              if (filler_size != 0)
31              {
32                      filler_string = (char*) malloc(32*sizeof(char));
33                      // remaining chars into allocated 32 bytes memory
34                      strncpy(filler_string, string, filler_size);
35                      xmm = _mm256_loadu_si256((__m256i*) filler_string);
36                      counter_epi8 = _mm256_add_epi8(counter_epi8,
37                                              regCountChar(&xmm));
38
39                      free(filler_string);
40              }
41
42              // add remaining results
43              counter_epi64 = merge_epi8_to_epi64(&counter_epi64, &counter_epi8);
44              return sum_up_epi64(&counter_epi64);
45      }
```

Der grobe Verlauf ist wie bei 'toLowerCasePar' und 'toUppercasePar': solange noch 32 Zeichen übrig sind, wird der String in 32-Zeichen-Schritten abgearbeitet und die letzten Zeichen separated in extra dafür allokiertem Speicher.
Um erst ganz zum Schluss die Zahlen aus den Registern in einen Integer umwandeln zu müssen, und dadurch so lange wie möglich nur Vektoroperationen zu verwenden, werden zwei Register mehr benötigt: 'counter_epi8' und 'counter_epi64'. Ziel ist die Anzahl der vorkommenden 'c' in diesen Registern zu zählen und aus dem epi64 Register zum Schluss die Summe zu berechnen. Zunächst werden dafür in line 7-8 beide Register mit 0 gefüllt; auf Null gesetzt.
Die Bearbeitung von 32 Zeichen beginnt ebenfalls damit, dass die Zeichen in ein Register geladen werden. Anschließend wird dieses Register an 'regCountChar' übergeben und das Ergebnis auf das aktuelle epi8 Zählregister addiert.

```
1  __m256i regCountChar(__m256i *string)
2  {
3          return (_mm256_and_si256(_mm256_cmpeq_epi8(*string, c_register),
4                  one_register));
5  }
```

'regCountChar' führt einen Vergleich mit dem 'c_register' durch und verwendet anschließend 'one_register' als Bitmask, um die zusätzlichen durch den Vergleich entstehenden Einsen zu entfernen.
Am Beispiel:

ASCII Werte:	115	99	104	105	99	107	...
in Binär:	01110011	01100011	1101000	01110011	01100011	01101011	...
Nach Vergleich:	00000000	1111111	00000000	00000000	1111111	00000000	...
Bitmaske:	00000001	00000001	00000001	00000001	00000001	00000001	...
Rückgabewert:	00000000	00000001	00000000	00000000	00000001	00000000	...

Durch das aufaddieren dieser Rückgabe werte wird also in den 8bit Bereichen gezählt wie oft in dem Bereich mit diesem Index 'c' gestanden hat. Hierbei entsteht ein Problem sobald ein 8Bit Bereich den Wert 255 erreicht. Würde nun erneut an dieser Stelle ein 'c' stehen käme es durch die Addition zu einem Overflow. Um das zu verhindern wird der aktuelle Zählstand alle 255 Schritte in das epi64 Zählregister übertragen und das epi8 Register zurückgesetzt. Das passiert in 2.3 in line 18-24. Da i immer um 32 erhöht wird sind immer dann 255 Schritte erreicht, wenn i ganzzahlig durch 255*32=8160 teilen lässt. Das übertragen der Ergebnisse passiert in der Funktion 'merge_epi8_to_epi64'.

```
1 __m256i merge_epi8_to_epi64(__m256i *counter_epi64, __m256i *counter_epi8)
2 {
3         return _mm256_add_epi64(*counter_epi64,
4         // this sums up 8bits into the lower 16bits of 64bit Blocks
5         _mm256_sad_epu8(*counter_epi8, _mm256_setzero_si256()));
6 }
```

Die Idee ist, dass mehrere 8Bit Bereiche zusammengefasst werden. Dafür wird hier die Funktion '_mm256_sad_epu8' verwendet. Diese Funktion ist eigentlich dazu da, um die absolute Differenz zwischen den 8Bit Blöcken der beiden gegebenen Register zu Berechnen und jeweils 8 davon horizontal zusammen zu addieren. Das Ergebnis wird dann in die niedrigsten 16Bit der 4 64Bit Blöcke eines 256 Bit Registers geschrieben. Als zweites Register wird ein Register mit nur Nullen verwendet, sodass die absolute Differenz immer der absolute Betrag der 8Bit Sektionen in epi8 Zählregister beträgt. Es werden also die ersten 8 8Bit Bereiche aufaddiert und in die Summe in die ersten 64Bit des Ergebnisregisters geschrieben. Die 2. 8 8Bit Bereiche in den 2. usw. Das Ergebnis dieser Berechnung wird auf das aktuelle epi64 Zählregister addiert, sodass in diesem gezählt werden kann bis die 64Bits nicht mehr aussreichen. Da 'countCharPar' mit einer 'int' Stringlänge arbeitet und nicht weiter arbeitet also diese, ist es nicht möglich, dass unser epi64 Register overflowed.
Auf gleiche Art und Weise werden die restlichen Zeichen, welche kein volles Register füllen, bearbeitet. Zum Schluss bleibt ein epi64 Register mit 4 64Bit Bereichen, welche aufaddiert werden müssen für das finale Ergebnis.

```
1         int64_t sum_up_epi64(__m256i *counter)
2         {
3                 return (_mm256_extract_epi64(*counter, 0) +
4                 _mm256_extract_epi64(*counter, 1) +
5                 _mm256_extract_epi64(*counter, 2) +
6                 _mm256_extract_epi64(*counter, 3));
7         }
```

Dazu wird mit '_mm256_extract_epi64' jeder 64Bit Block separat ausgelesen und aufaddiert. Diese Funktion wird pro String nur einmal ausgeführt, daher fällt es nicht stark ins Gewicht, dass es eine 'sequentielle Addition' ist.

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 string_manipulation_seq.o
13         gcc -mavx2 -o string_manipulation string_manipulation.o string_manipulation_seq.o string_manipulation_par.o
14
```

```
15 clean:
16 	rm string_manipulation_par.o string_manipulation.o string_manipulation_seq.o string_manipulation
```

2.5 evaluation.py

Im Evaluierungsskript werden zunächst die csvs aus der Messung eingelesen. Hierfür wird die Bibliothek 'pandas' genutzt. Die daraus resultierenden 'pandas.DataFrames' werden im restlichen Programm als Variablen für die Messwerte verwendet. 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. Die Werte hierfür werden mit den numpy Funktionen 'numpy.mean' und 'numpy.std' berechnet.
'complex_plots' erstellt 3 Diagramme jeweils für jede Funktion, die gemessen wurde. Im Diagramm werden die Mittelwerte für parallel/sequentiell über den Stringlängen dargestellt mit den Standardabweichungen als Fehlerbalken. 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 die Laufzeiten für jede Messung dargestellt, heißt die Laufzeiten über den Iterationen der Laufzeitmessung. Die Diagramme enthalten außerdem einen horizontale Linie auf Höhe der Mittelwerte und darum einen Bereich in der größe der Standardabweichung.
Die Funktionen an sich sind nicht besonders interessant, da es primär plot-shenanigans ist. Für das Erstellen der Diagramme wird matplotlib (bzw. gezielt pyplot) verwendet.

3 Ausführung

3.1 Hardware

Die Messung für die Bearbeitung der Aufgaben sind auf dem CPU Cluster Romeo der TU Dresden ausgeführt worden. Dieser Cluster bietet 192 nodes mit jeweils [2]:

- 2 x AMD EPYC CPU 7702 (64 cores) @ 2.0 GHz, Multithreading möglich
- 512 GB RAM

- 200 GB SSD Speicher
- Betriebssystem: Rocky Linux 8.7

3.2 Programm-Versionen

Relevant für die Reproduzierbarkeit sind die Versionen der verwendeten Bibliotheken und Programme.

- GNU Make 4.2.1
- gcc (GCC) 10.3.0
- Python 3.9.5
  - numpy 1.24.1
  - pandas 2.0.0
  - matplotlib 3.3.4

3.3 Messung

Die Messung wurde auf dem 'romeo' cluster der TU Dresden durchgeführt mit folgenden Schritten. Zunächst ein Login darauf notwendig.

```
ssh <username>@login1.romeo.hpc.tu-dresden.de
```

Anschließend das Repository geklont:

```
git clone https://github.com/LennickeBe/KP_Paralleles_Rechnen.git
```

Eine Messung wird mit dem sbatch script 'run.sh' im Repository unter 'Aufgabe_A/measurement' gestartet.

```
cd KP_Paralleles_Rechnen/Aufgabe_A/measurement
sbatch run.sh
```

3.4 Auswertung

Die Daten wurden durch ein python Programm ausgewertet mit folgendem Setup:

```
module load release/23.04
module load GCCcore/10.3.0
module load Python/3.9.5
cd ~/KP_Paralleles_Rechnen/Aufgabe_A/evaluation
python3 -m venv .venv
source .venv/bin/activate
pip install numpy==1.24.1 pandas==2.0.0 matplotlib==3.3.4
```

Für eine lokale Auswertung muss das eigene Setup entsprechend angepasst werden, dass die Versionen eingehalten werden.
Gestartet wurde die Auswertung dann mit:

```
cd ~/KP_Paralleles_Rechnen/Aufgabe_A/evaluation
source .venv/bin/activate
python3 evaluation.py
```

Die Resultate sind unter ' /KP_Paralleles_Rechnen/Aufgabe_A/report/images' und ' /KP_Paralleles_Rechnen/Aufgabe_A gespeichert worden. Die Bilder werden beim compilieren der tex-Datei direkt verwendet. Die Tabellen mussten vorher aus den jeweiligen Dateien in den Report kopiert werden. (ist bereits syntaktisch aufgearbeitet, aber ich habe ein automatisches einfügen nicht hinbekommen.)

4 Auswertung

4.1 Zeitkomplexität

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

Die Funktionen heißen 'toUppercaseSeq', 'toLowercaseSeq' und '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 'string_manipulation_par.c' und heißen 'toUppercasePar', 'toLowercasePar' und '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. Die eingezeichneten Fehlerbalken (nur für String Länge von 10^8 zu sehen) sind die Standardabweichungen der jeweiligen Mittelwerte.

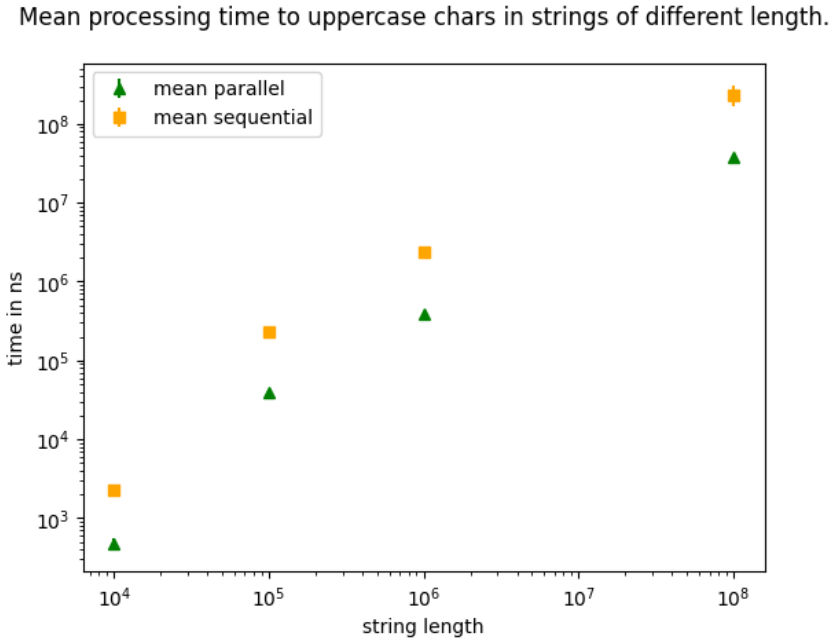

Abbildung 1: Durchschnittliche Durchführungszeit von toUppercase() in Abhängigkeit von der Stringlänge.

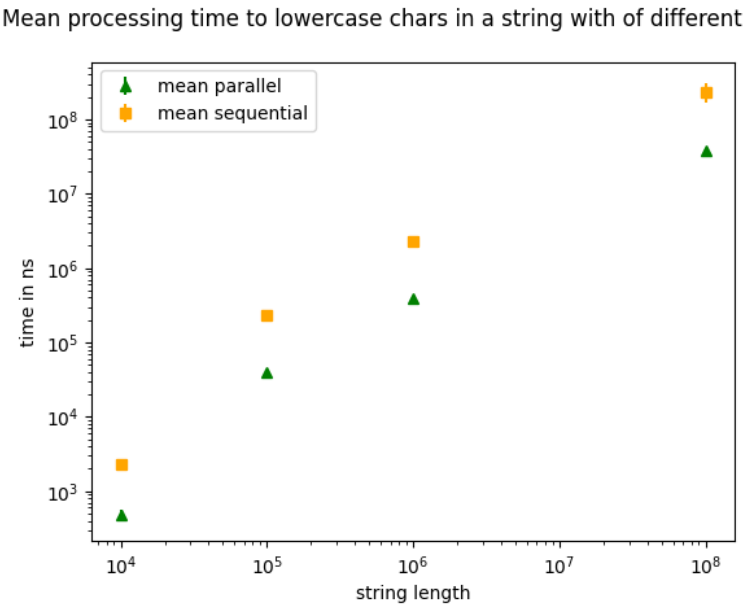

Abbildung 2: Durchschnittliche Durchführungszeit von toLowercase() in Abhängigkeit von der Stringlänge.

4.2 Ausführungszeiten

4.2.1 toUppercase

In der Tabelle 4.2.1 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	481.99	41.55	2312.98	173.74
100000	38941.19	2862.02	233782.84	11024.82
1000000	390209.25	16001.90	2333485.31	29096.43
100000000	38565800.52	358167.95	234261960.43	8838591.01

4.2.2 toLowercase

In der Tabelle 4.2.2 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	486.48	31.04	2303.06	39.46
100000	38923.30	2731.60	235405.92	13489.79
1000000	389663.96	17655.14	2330971.68	29481.62
100000000	38668967.04	418400.19	234386070.75	8881648.84

4.2.3 countChar

In der Tabelle 4.2.3 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

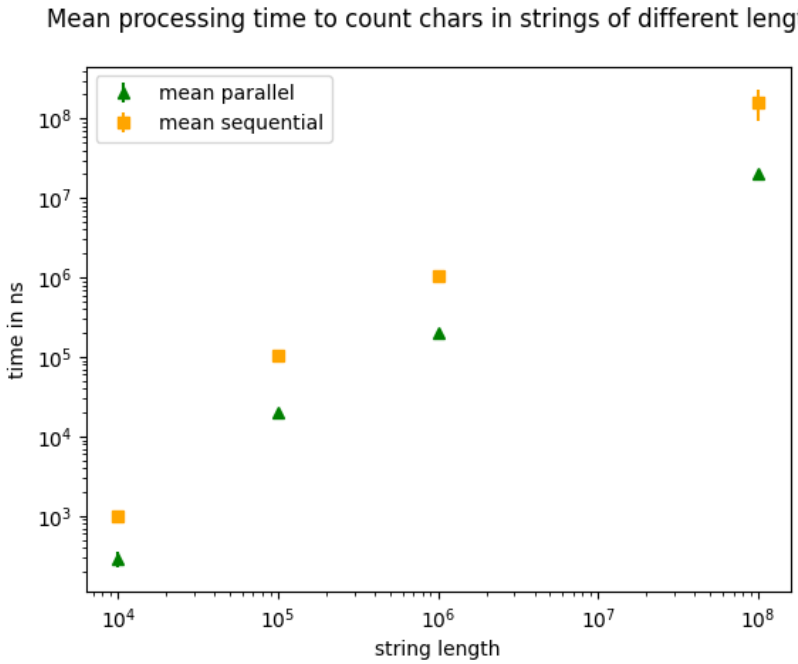

Abbildung 3: Durchschnittliche Durchführungszeit von countChar() in Abhängigkeit von der Stringlänge.

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	289.80	66.57	1010.79	64.76
100000	20327.01	1503.70	104617.29	6654.95
1000000	205360.44	13806.94	1053384.32	20812.50
100000000	20547499.90	158011.15	159285715.48	67981063.92

4.3 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.

4.3.1 toUppercase

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

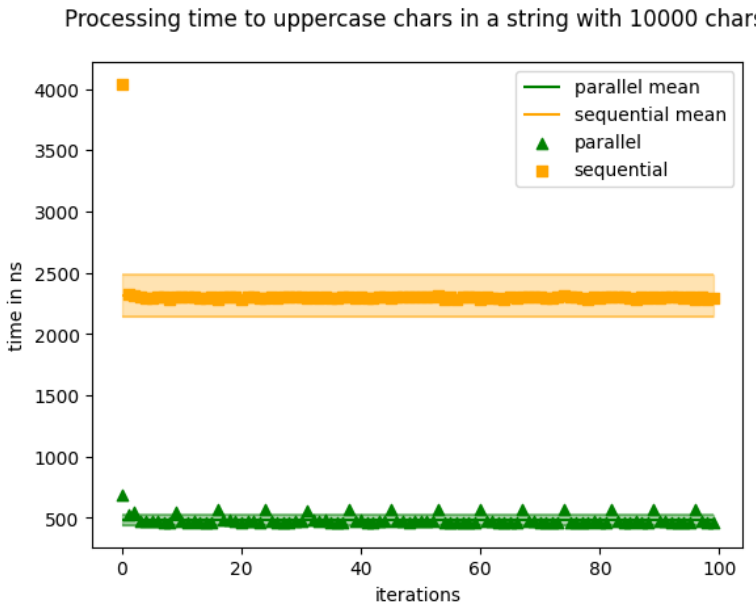

Abbildung 4: Durchführungszeiten von toUppercase für einen String mit 10.000 Zeichen.

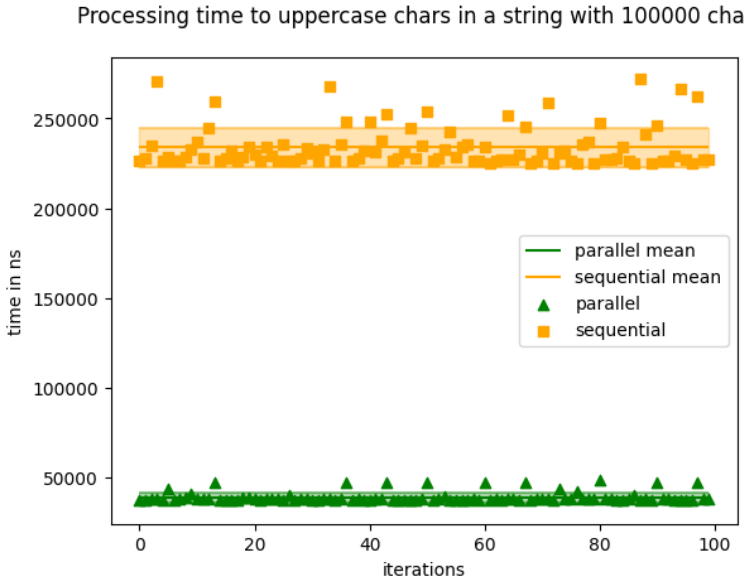

Abbildung 5: Durchführungszeiten von toUppercase für einen String mit 100.000 Zeichen.

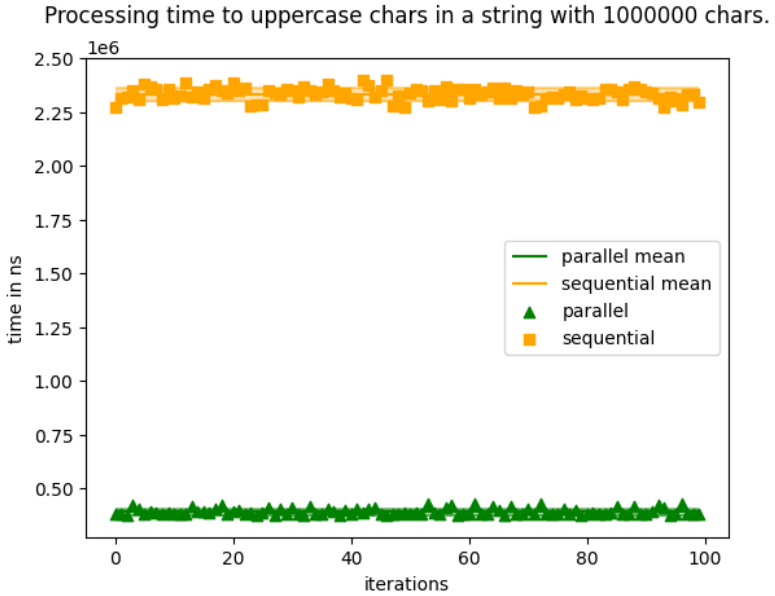

Abbildung 6: Durchführungszeiten von toUppercase für einen String mit 1.000.000 Zeichen.

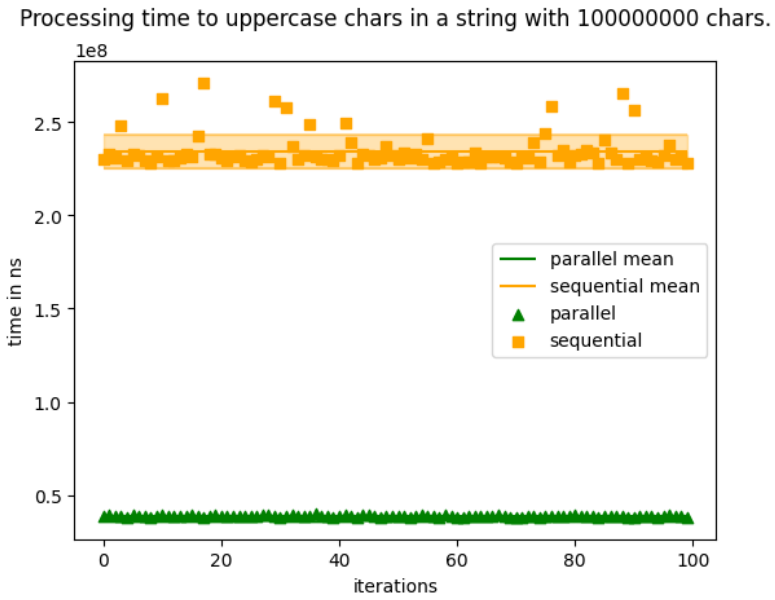

Abbildung 7: Durchführungszeiten von toUppercase für einen String mit 100.000.000 Zeichen.

4.3.2 toLowercase

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

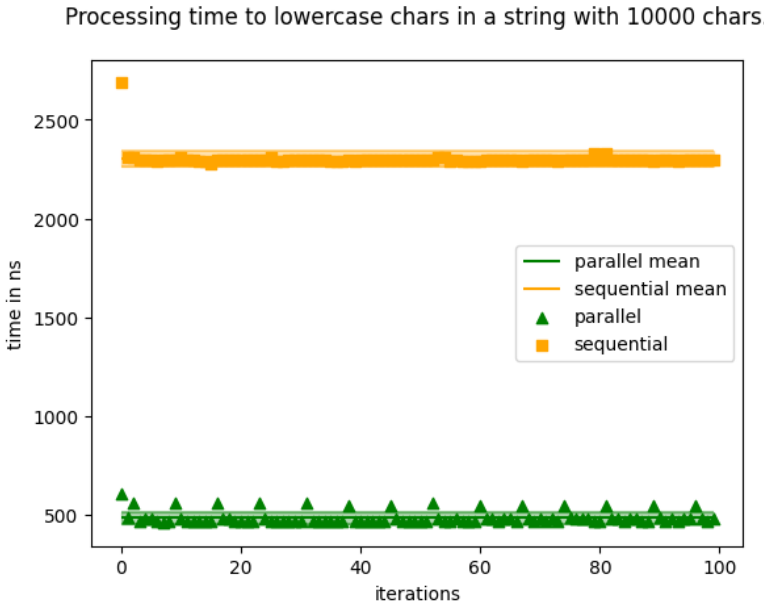

Abbildung 8: Durchführungszeiten von toLowercase für einen String mit 10.000 Zeichen.

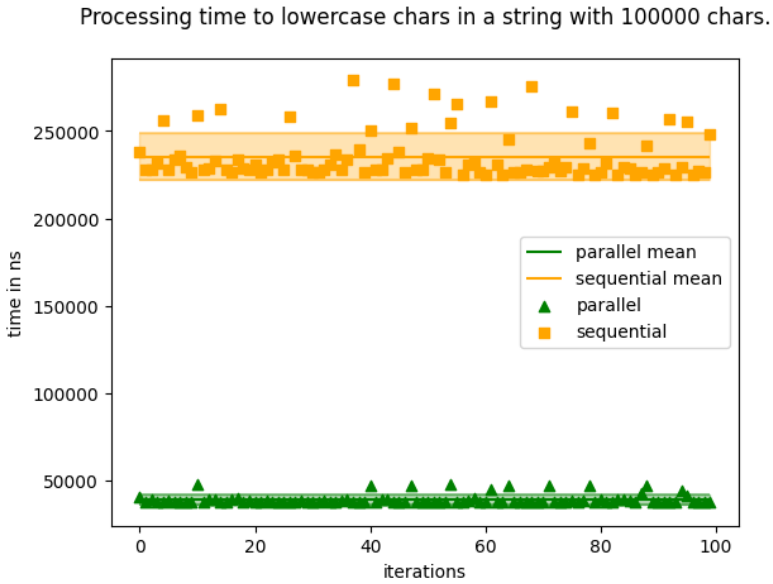

Abbildung 9: Durchführungszeiten von toLowercase für einen String mit 100.000 Zeichen.

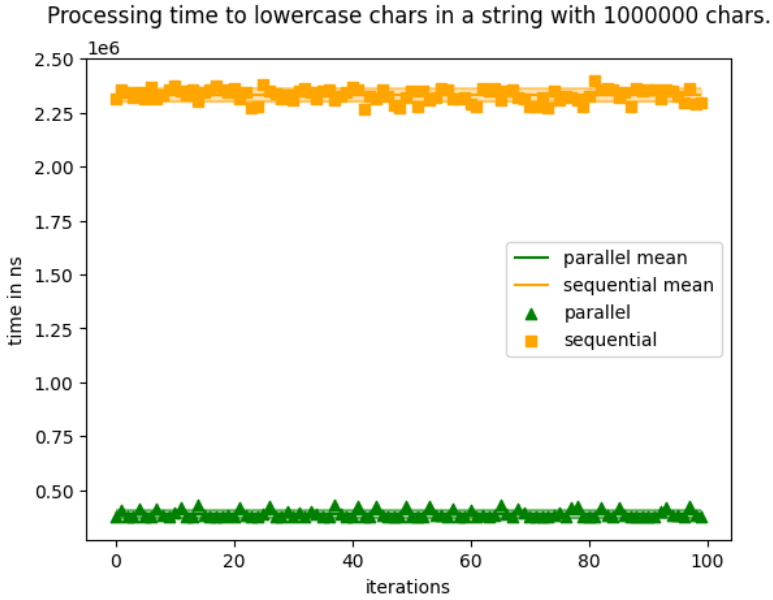

Abbildung 10: Durchführungszeiten von toLowercase für einen String mit 1.000.000 Zeichen.

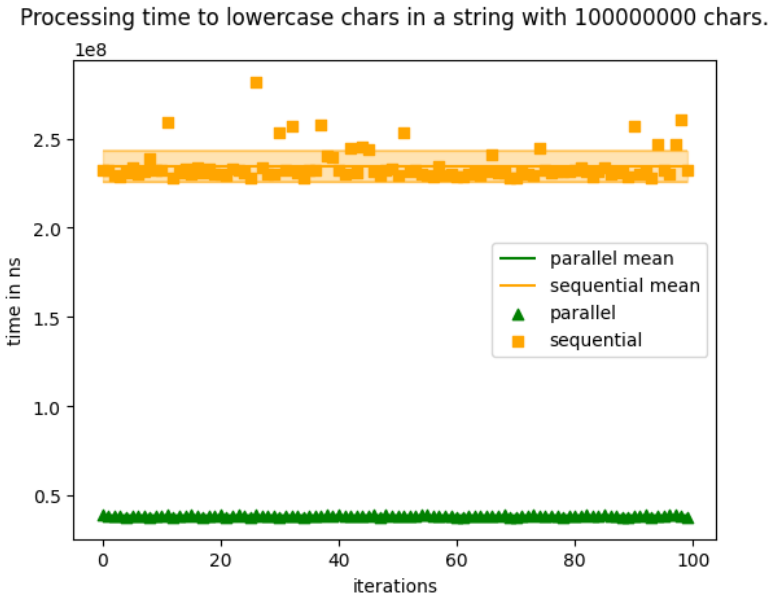

Abbildung 11: Durchführungszeiten von toLowercase für einen String mit 100.000.000 Zeichen.

4.3.3 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 deutlicher schneller ist.
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, allerdings nicht bei 'lowercase' und 'uppercase' also sollte das Problem auch nicht in bei Speichermangel etc. liegen. Ich konnte das gleiche Verhalten bei meinem Laptop nicht beobachten und kann es auch nicht sicher erklären.

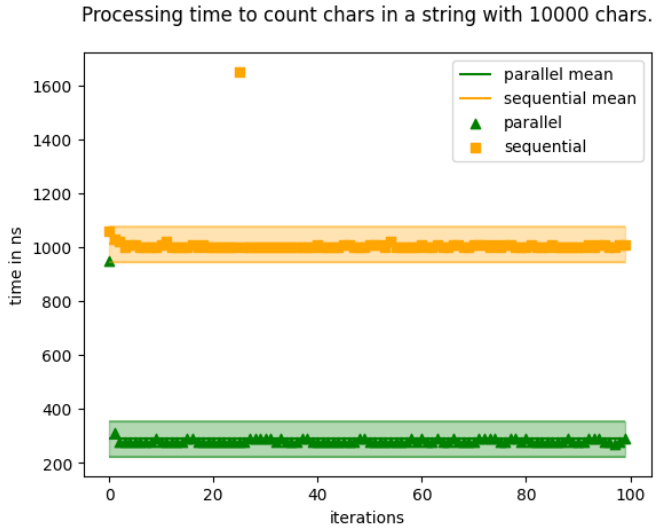

Abbildung 12: Durchführungszeiten von countChars für einen String mit 10.000 Zeichen.

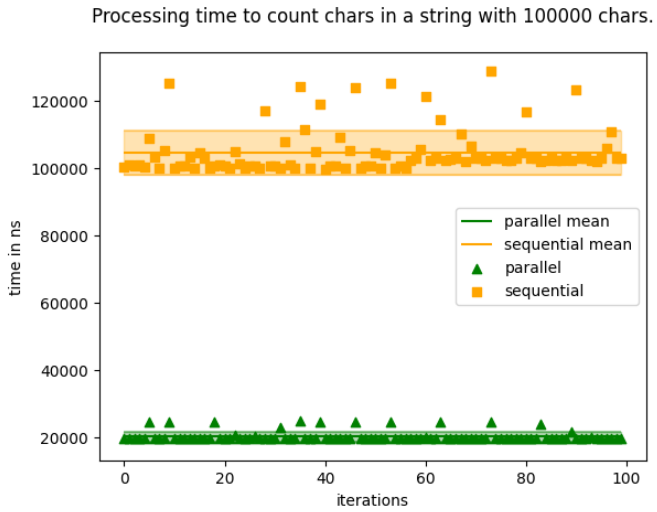

Abbildung 13: Durchführungszeiten von countChars für einen String mit 100.000 Zeichen.

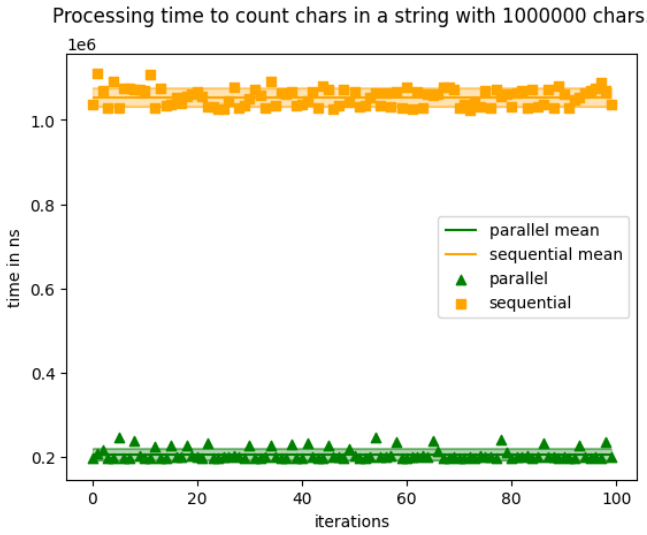

Abbildung 14: Durchführungszeiten von countChars für einen String mit 1.000.000 Zeichen.

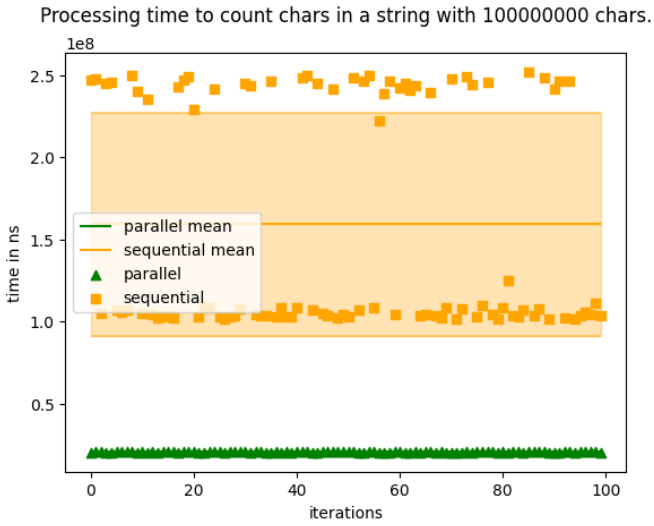

Abbildung 15: Durchführungszeiten von countChars für einen String mit 100.000.000 Zeichen.

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

[2] HPC Compendium, 'HPC Resources', 12.01.2024
https://doc.zih.tu-dresden.de/jobs_and_resources/hardware_overview/#romeo