Berechnung der Kreiszahl π mithilfe der Leibniz' Formel

Rafael Schreiber

Januar 2021

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Laut der Angabe ist das Ziel dieses Programms die Kreiszahl π zu berechnen. Dabei soll die Leibniz' Formel [1] zur Annäherung an die Zahl verwendet werden.

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Wie in der Formel zu erkennen ist, gibt es positive und negative Terme. In der geforderten Implementierung wird erwartet, dass die negativen und positiven Terme auf je zwei Prozesse aufgeteilt werden, damit die jeweiligen Ergebnisse parallel und damit auch hoffentlich schneller berechnet werden können.
Die Angabe schreibt vor, dass der Elternprozess diese zwei Kindprozesse erstellt und dann darauf wartet, bis die Kindprozesse ihre Terme fertig berechnet haben. Die jeweils positiven und negativen Ergebnisse werden in eine Textdatei geschrieben, welche dann vom Elternprozess ausgelesen wird, damit dieser die entgültige Berechnung durchführen kann.

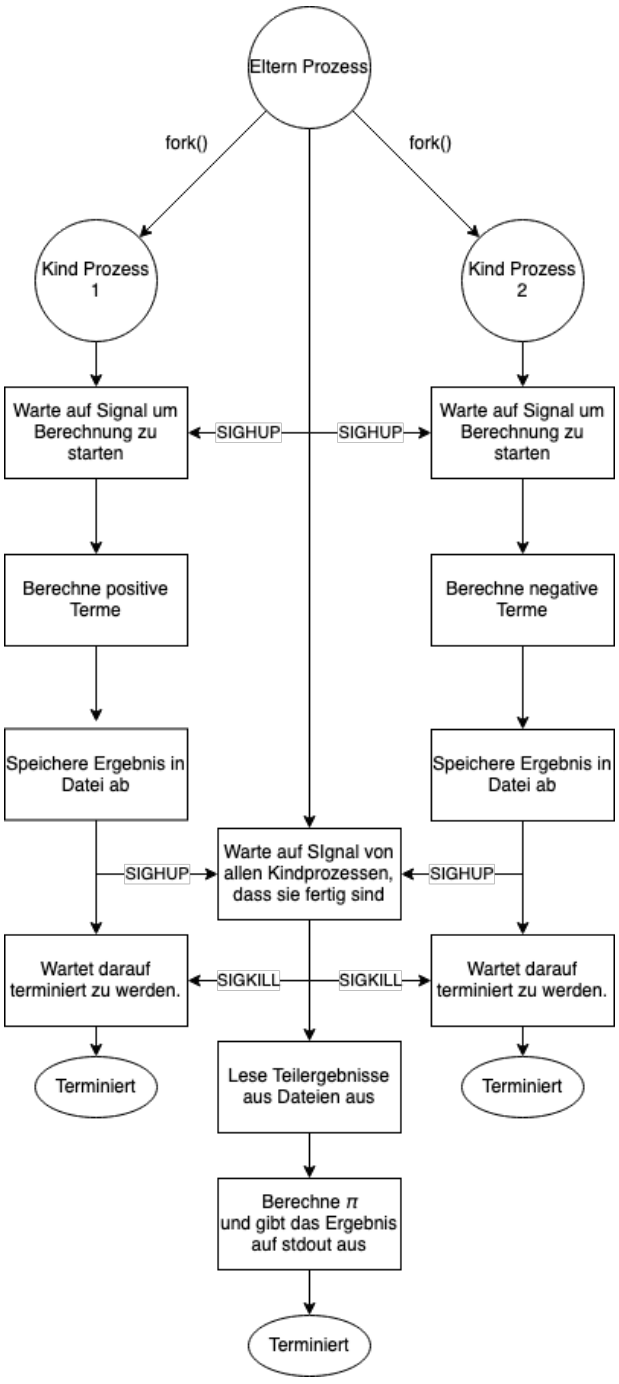

Abbildung 1: Ablaufdiagramm des picalc Programms

2 Implementierung

Zu aller erst wurden sämtliche Codierungsrichtlinien von Professor Dr. Günter Kolousek eingehalten. Für die Benennung von Variablen und Funktionen wurde "`sneak_case`"verwendet. Für eine einfachere Kodierung wurden außerdem mehrere externe Bibliotheken verwendet, auf welche im Abschnitt 2.2 näher eingegangen werden.

2.1 Erklärung bestimmter Codeteile

2.1.1 Nutzung von `__float128` für mehr Präzision

Bei der Berechnung von irrationalen Zahlen wie π oder e ist das Ziel, auf soviele Nachkommastellen wie nur möglich zu kommen. Die Anzahl der Nachkommastellen wird als Präzision bezeichnet. Diese muss auch hoch sein, um das Ergebnis auf soviele Nachkommastellen wie nur möglich berechnen und darstellen zu können.
`__float128` ist der zugehörige Datentyp für das „Quadruple-precision floating-point format“ welches in der aktuellen Fassung des IEEE 754-2008 [2] Standards beschrieben ist.
Das Problem dabei ist, dass nicht jeder Compiler bzw. jedes System diesen Datentyp unterstützt, deswegen schaut der Präprozessor nach, ob dieser Datentyp unterstützt wird. Wenn dies nicht der Fall ist, dann wird `long double` zur Speicherung von Kommazahlen verwendet.

```
#ifdef USE_FLOAT128
    typedef __float128  long_double_t;
#else
    typedef long double long_double_t;
#endif
```

2.1.2 Umwandlung von `long_double_t` zu `std::string`

Ein weiteres Problem ist, wie `long_double_t` zu einem String konvertiert werden soll. String wird benötigt, weil jeweils die Funktion zum Speichern der Teilergebnisse sowie die Ausgabe, ein Objekt mit dem Typen `std::string` benötigt.
An und für sich existiert die Funktion `std::to_string()` aber hier ist wieder

das Problem mit der Präzision. Deswegen wurde die Funktion `to_string_w_precision()` geschrieben, um dieses Problem zu beheben.

```
template <typename T>
string to_string_w_precision(const T a_value, const int n = 256){
    ostringstream out;
    out.precision(n);
    out << fixed << a_value;
    string flt = out.str();
    reverse(flt.begin(), flt.end());
    uint8_t trailing_zeros = 0;
    for (char chr : flt){
        if (chr != '0'){
            break;
        }
        trailing_zeros++;
    }
    flt = flt.substr(trailing_zeros, flt.size());
    reverse(flt.begin(), flt.end());
    return flt;
}
```

Diese Funktion erstellt Anfangs einen Puffer, wo die Kommazahl mit einer beliebigen Präzision `n` hineingeschrieben werden kann. Es werden standardmäßig 256 Nachkommastellen verwendet, weil in der Praxis diese Genauigkeit sowieso nicht überschritten wird. Da bei Zahlen mit einer geringeren Präzision die restlichen Nachkommastellen nicht verwendet werden, sondern mit 0en aufgefüllt werden, werden diese danach aus rein kosmetischen Gründen entfernt.

2.1.3 Algorithmus zur Berechnung der Teilergebnisse

Das sogenannte „'Herzstück“ dieses Programms ist der Algorithmus zur Berechnung der Teilergebnisse.

```
long_double_t calc_part_leibniz(uint64_t n, bool is_positive){
    uint8_t startpoint{1};
    if (!is_positive){
        startpoint = 3;
    }
    long_double_t part_result = 0.0;
    for (uint64_t i{0}; i < n; i++){
        part_result += (long_double_t) 1 / (startpoint + 4 * i);
    }

    if (!is_positive){
        part_result = -part_result;
    }

    return part_result;
}
```

Wie man sehen kann, verlangt diese Funktion zwei Argumente:

- `int n` Anzahl der Schleifendurchläufe, also wie oft der Algorithmus wiederholt wird und somit, wie genau die Annäherung an π ist.
- `bool is_positive` bestimmt ob von der Funktion entweder die positiven oder die negativen Terme berechnet werden sollen.

Die einzigen Unterschiede zwischen der Berechnung der positiven und der negativen Teilergebnisse sind, dass der Startwert im Nenner anders ist und dass das Ergebnis beim negativen Teilergebnis negiert wird.

2.1.4 Finale Berechnung der Kreiszahl π

Der Teil wofür dieses Programm überhaupt geschrieben wurde befindet sich in der folgenden Funktion:

```
void calc_pi(){
    long_double_t pos, neg, pi;
    string pos_str, neg_str;
    string storepath_pos{result_file_path + "part_pos_" +
                         to_string(getpid()) + ".txt"};
    string storepath_neg{result_file_path + "part_neg_" +
                         to_string(getpid()) + ".txt"};
    pos_str = tfile::read(storepath_pos.c_str());
    neg_str = tfile::read(storepath_neg.c_str());
    pos = stold(pos_str);
    neg = stold(neg_str);
    pi = (pos + neg) * 4;
    console->info("Final calculation finished. pi = \e[1m{}\e[0m",
                  to_string_w_precision(pi));
}
```

In dieser Funktion werden zuerst die Dateipfade beider Textdateien „zusammengebaut“ um diese dann mithilfe der Funktion `tfile::read()`, bereitgestellt von der Bibliothek tfile [2.2.3], auszulesen. Diese zwei Textdateien enthalten jeweils die positiven und negativen Teilergebnisse. Diese Teilergebnisse werden dann mit der Funktion `std::stold()` in den `long_double_t` Datentyp konvertiert, um damit rechnen zu können. Das finale Ergebnis, nämlich π, wird berechnet, indem das positive und das negative Teilergebnis addiert wird und dieses dann mit 4 multipliziert wird. Zuletzt wird das Endergebnis schön formatiert auf stdout ausgegeben.

2.2 Externe Bibliotheken im Einsatz

Um nicht jede Kleinigkeit selbst programmieren zu müssen und auch weil es von der Angabe so gefordert wurde, kamen in diesem Programm folgende Bibliotheken zum Einsatz.

2.2.1 CLI11

CLI11 [3] ermöglicht die Erstellung einer schönen Komandozeilenschnittstelle mit wenig Programmieraufwand. Diese Bibliothek ist eine header-only Bibliothek und bietet auch andere Vorteile wie z.B. eingebaute Methoden zum überprüfen von angegebenen Kommandozeilenparametern.

2.2.2 spdlog

spdlog [4] ist ebenfalls eine header-only Bibliothek um ein einfaches Loggen zu ermöglichen. Das Objekt zum loggen wird in diesem Programm global deklariert, damit jede Funktion darüber ihren Output geben kann.

2.2.3 tfile

Die ebenfalls header-only Bibliothek tfile [5] ist mit ihren 598 LOC die kleinste verwendete Bibliothek im ganzen Projekt. Obwohl tfile nicht groß ist, bietet sie viele nützliche Funktionen, um den Umgang mit Dateien zu vereinfachen.

3 Verwendung

3.1 Kompilierung

Die Abhängigkeiten vom Projekt sind:

- The Meson Build System [6]
- Die Bibliotheken aufgeführt im Punkt [2.2]
- Einen C++ Compiler

Getestet wurde die Kompilierung auf folgenden Plattformen:

- macOS Big Sur (Apple clang version 12.0.0)
- Ubuntu 20.04 LTS (gcc version 9.3.0)

Beide Plattformen basieren auf einen Prozessor mit x86 Befehlssatz.

Zum kompilieren wechselt man als erstes in das `build/` Verzeichnis, welches anfangs leer sein sollte. Zweitens werden mit `meson ..` die build-files generiert und zuletzt wird mit dem Befehl `ninja` das Programm kompiliert. Im selben Verzeichnis sollte dann die Binary `picalc` zu finden sein.

3.2 Kommandozeilenparamter

3.2.1 -i, –iterations UINT:POSITIVE REQUIRED

Nach diesem Parameter, wird als Argument die Anzahl der Durchläufe angegeben. Dabei ist es wichtig darauf zu achten, dass die Anzahl größer gleich 2 sein muss. Ist dies nicht der Fall oder wird dieser Parameter garnicht mitgegeben, dann weigert das Programm sich zu starten.

3.2.2 -l, –location TEXT:DIR OPTIONAL

Mit diesem Parameter kann spezifiziert werden, wo die Textdateien mit den Teilergebnissen gespeichert werden sollen. Standardmäßig werden diese unter `/tmp` abgespeichert. Wenn dies nicht erwünscht ist, dann kann als Argument ein anderer Speicherort angegeben werden.

3.2.3 -d, –delete

Dieser Flag gibt an ob die Dateien mit den Teilergebnissen nach der Berechnung gelöscht werden sollen. Standardmäßig ist das nicht der Fall, aber wenn man nicht möchte, dass das System mit Dateien zugemüllt wird, dann macht dieser Flag durchaus Sinn.

Literatur

[1] Wikipedia: Leibniz-Reihe
https://de.wikipedia.org/wiki/Leibniz-Reihe

[2] 754-2019 - IEEE Standard for Floating-Point Arithmetic
https://ieeexplore.ieee.org/document/8766229

[3] GitHub: CLIUtils/CLI11 command line parser
`https://github.com/CLIUtils/CLI11`

[4] GitHub: gabime/spdlog Fast C++ logging library.
`https://github.com/gabime/spdlog`

[5] GitHub: rec/tfile tiny C++11 file utilities by Tom Ritchford
`https://github.com/rec/tfile`

[6] The Meson Build System: Project Website
`https://mesonbuild.com/`