LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 108 – Abgabe zu Aufgabe A329
Wintersemester 2022/23

Cara Dickmann Thua Duc Nguyen Eslam Nasrallah

1 Einleitung

Pi (π) ist die mathematische Konstante, deren Wert das Verhältnis zwischen Umfang und Durchmesser eines Kreises wiedergibt.

Während es bei einfachen geometrischen Berechnungen oft ausreicht, mit dem Näherungswert von π, wie 3.14, zu rechnen, gibt es auch Gebiete, in denen ein hohes Maß an Präzision erforderlich ist. So ist es beispielsweise im Bereich der Computergrafik von Vorteil, mit einem sehr genauen Wert von π zu rechnen, um Bilder und Animationen von runden Formen möglichst realistisch zu gestalten. Auch in der Kryptographie nutzt man π mit möglichst vielen Nachkommastellen, um damit Zufallszahlen zu erzeugen.

Es gibt viele andere Anwendungsbereiche, in denen ein möglichst exaktes π verlangt ist. Da π jedoch eine irrationale Zahl ist, besitzt es eine unendliche Anzahl von nichtperiodischen Dezimalstellen und lässt sich nicht ohne weiteres berechnen. Im Laufe der Zeit wurden allerdings diverse Methoden entwickelt, um sich π zu approximieren.

Dazu gehört die in diesem Projekt angegebene Formel 1, welche für $n = \inf$ den genauen Wert von π ergäbe.

$$2 + \sum_{k=1}^{n} \frac{k!^2 \cdot 2^{k+1}}{(2k+1)!} \tag{1}$$

Da es natürlich nicht wirklich mit $n = \inf$ gerechnet werden kann, wird mit einer reellen Zahl für n auf die gewünschte Anzahl von Nachkommastellen gerundet. Um die Berechnung so effizient wie möglich zu gestalten, wird für die Berechnung des Terms (1) Binary Splitting angewendet. Zur effektiveren Durchführung der hierbei erforderlichen Multiplikationen wird, entsprechend der Aufgabenstellung, die Karazuba-Multiplikation verwendet.

2 Lösungsansatz

2.1 Darstellung großer Zahlen

2.1.1 Bignum Konstruktion

Um Pi mit einer beliebig hohen Präzision zu berechnen, wird eine Datenstruktur mit unlimitierter Anzahl an Nachkommastellen benötigt. Da dies bei den standardmäßigen Datentypen nicht vorgesehen ist, wurde eine dafür geeignete Struktur, namens „Bignum“ erstellt.

Ein Bignum Objekt besitzt ein Array mit Datenblöcken, welche die Ziffern eines unsigned Integer zur Basis 32 Bit repräsentieren. Zudem enthält ein Objekt vom Typ bignum ein Attribut, dass die Gesamtlänge der Zahl enthält, sowie eines, das die Anzahl an Nachkommastellen angibt. Obwohl bei einer größeren Basis, wie 64 Bit weniger Rechenoperationen nötig wären, wird als Basis 32 Bit gewählt, damit sich bei der Multiplikation zweier Zahlen keine Probleme ergeben.

Die Zahlen werden in Little-Endian gespeichert, da die Mehrheit der arithmetischen Operationen aufgrund des Übertrags mit der niederwertigsten Ziffer starten. Daher befindet sich der Block mit der kleinsten Stellenwertigkeit am Anfang des Datenarrays.

Da für die Berechnung von Pi keine negativen Zahlen benötigt werden, sind ist Bignum Struktur unsigned.

2.1.2 Darstellung von Nachkommastellen

Zur Darstellung der Nachkommastellen wird eine Bignum als Festkommazahl interpretiert.

Die Verwendung von Fließkommazahlen nach IEEE-754 ist nicht geeignet, da diese Darstellung nur eine begrenzte Anzahl von Bits zur Darstellung des Exponenten und der Signifikanten bietet. Dadurch verliert diese Art der Zahlendarstellung zu sehr an Genauigkeit, wenn mit großen Zahlen gerechnet wird. Rundungsfehler können auch auftreten, wenn mit einer solchen Darstellung Rechenoperationen auf Zahlen mit unterschiedlicher Genauigkeit durchgeführt werden.

Um den Rechenaufwand in der Implementierung zu minimieren und weniger Speicher zu verbrauchen, wird auf die Darstellung von Nachkommastellen, die nicht signifikant sind, also Nullen die am Ende der Zahl stehen, verzichtet.

2.2 Arithmetik

In diesem Projekt ist sowohl Ganzzahlarithmetik als auch das Rechnen mit Nachkommastellen notwendig. Durch die Konstruktion des Structs „Bignum“ ist es nicht kompliziert Addition oder Subtraktion durchzuführen. Multiplikation und Division erfordern jedoch fortgeschrittenere Verfahren, die Zahlen mit beliebiger Genauigkeit effizient miteinander verrechnen können. Nach den Anforderungen der Aufgabenstellung wird für das Bestimmen von Produkten die Karazuba-Multiplikation verwendet. Um Quotienten zu berechnen, wird das Newton-Raphson-Division Verfahren angewendet.

2.2.1 Addition/Subtraktion

Bei der Addition bzw. Subtraktion zweier Zahlen, es ist wichtig sicherzustellen, dass nur Blöcke mit der gleichen Signifikanz addiert/subtrahiert werden. Falls bei der Addition, eine Zahl mehr Nachkommastellen hat, kann deren Wert an den zugehörigen Blöcken einfach übertragen werden.

Die Subtraktion unterscheidet sich nicht sehr von der Addition. Allerdings muss beachtet werden, ob der Subtrahend größer als der Minuend ist. In diesem Fall werden die beiden Zahlen zuerst vertauscht. Dann kann eine vorzeichenlose Subtraktion ausgeführt werden. Dabei muss beachtet werden, ob der Minuend oder der Subtrahend einen längeren Nachkommateil hat. Hat der Minuend einen Überschuss an Nachkommastellen, können diese ebenfalls unverändert übertragen werden. Ist der Nachkommaanteil des Subtrahenden länger, muss der beim Minuend von höherwertigen Ziffern "geliehen"werden.

2.2.2 Multiplikation

Bei der Multiplikation ist die Anzahl der Nachkommastellen immer gleich der Summe der Nachkommastellen der Faktoren und die Position des Kommas kann Einfluss auf die eigentlichen Werte der Ziffern, kann also während der Rechnung vernachlässigt werden. Um die Laufzeit einer naiven Multiplikation zu reduzieren, wurde der Karazuba-Algorithmus verwendet. Dadurch kann die Multiplikation von der Laufzeitklasse $\Theta(n^2)$ auf $O(n^{1.59})$ reduziert werden.

Hierfür werden die Faktoren in 4 gleiche Teile zerlegt. Ein Faktor x wird als $x = x_0 + b^m \cdot x_1$ betrachtet, wobei $m = \lceil \frac{n}{2} \rceil$, mit n die maximale Länge der beiden Faktoren ist und $b = (2^{32})$. Also enthält x_1 die obere Hälfte und x_0 die untere Hälfte der Blöcke. Damit alle Teile die gleiche Länge haben, wird wenn nötig Null-Padding an die höherwertigen Bits gesetzt hinzugefügt.

Anschließend wird das Produkt mit der Karazuba-Formel (2) berechnet.

$$x \cdot y = x_0 y_0 + b^2 \cdot x_1 y_1 + b \cdot ((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_0 y_0) \tag{2}$$

Die Multiplikation mit b bzw. b^2 kann jeweils durch einen Shift um einen bzw. zwei Blöcke nach links ersetzt werden.
Auf die Multiplikationen zwischen den Faktorteilen wird rekursiv wieder der Karazuba-Algorithmus angewendet, bis die Blocklänge eins erreicht ist, bei der dann normal multipliziert wird.

Dieser Algorithmus ließe sich durch Parallelisierung der Berechnung der einzelnen Faktoren beschleunigen, jedoch ist hier kein SIMD anwendbar.

2.2.3 Newton-Raphson Division

Divisionsalgorithmen lassen sich je nach Iterationsoperator hauptsächlich in zwei Klassen einteilen. Die erste Klasse verwendet die Subtraktion als iterativen Operator. Diese Methode erzeugt nur eine Ziffer pro Iteration und ist daher ziemlich zeitaufwändig. Die zweite Klasse, auch bekannt als Fast Division, verwendet die Multiplikation als iterativen Operator. Diese Algorithmen sind so konzipiert, dass sie praktisch und effizient sind und sich daher für eine Vielzahl von Anwendungen in der Computerarithmetik, der numerischen Analyse und dem wissenschaftlichen Rechnen eignen. Aus diesem

Grund benutzen wir in diesem Projekt die Newton-Raphson Division, welche ein Fast Divisionsalgorithmus ist [4].

Die Newton-Raphson Division verwendet die Newton-Methode, um den Kehrwert des Zählers zu ermitteln und diesen Kehrwert mit dem Nenner zu multiplizieren. Das Ergebnis davon ist der gesuchten Quotient. Der Algorithmus ist in die vier folgenden Schritte aufzugliedern.

- Normalisierung: Zähler Z und Nenner N werden zwischen 0,5 und 1 normalisiert. Dieser Schritt wird so durchgeführt, dass N nach recht geschifft wird, bis $N < 1$ gilt. Dies ist wichtig für die numerische Stabilität der Konvergenzrate des Bereichs der darstellbaren Zahlen und der Genauigkeit der Newton-Raphson Division.
- Initialisierung: Eine Konstante X_0 wird mithilfe von linearer Approximation in dem Form
$$\frac{48}{17} - \frac{32}{17} \cdot N'$$
berechnet, wobei N' die skalierte Version von N ist [5]. Die Konstante X ist eine Annäherung an den Kehrwert von N'.
- Iteration: Der Algorithmus führt eine Schleife durch, die
$$\left\lceil \frac{\log_2 (P+1)}{\log_2 17} \right\rceil$$
Runden durchläuft. P ist hierbei die Genauigkeit, also die Anzahl an Binärstellen des Ergebnisses. Diese Anzahl von Iterationen kann auf der Grundlage des festen Werts von P im Voraus berechnet werden.
Innerhalb der Schleife wird X durch mit
$$X_{n+1} = X_n \cdot (2 - N' \cdot X_n) \text{ [5]}$$
aktualisiert, um die Annäherung an den Kehrwert zu verbessern.
- Multiplikation: Die Multiplikation von Z' und X ergibt das Ergebnis mit gewünschten P Binärstellen.

2.3 Binary Splitting

Eine Methode zur schnelleren Auswertung von Reihen rationaler Terme ist das Binary Splitting.
Ein Grund für die Beschleunigung der Berechnung ist, dass Binary Splitting die Größe der entstehenden Teilprodukte reduziert. Außerdem ist beim binary Splitting nicht für jeden Term einer Reihe eine Division erforderlich, sondern nur abschließend eine einzige Division mit der gewünschten Genauigkeit.
Obendrein hat Binary Splitting eine geringere Komplexität als eine naive Berechnung einer Reihe.

Angenommen $M(n)$ sei die Komplexität der Multiplikation von zwei N-Bit-Zahlen. Die Komplexität der vom Binary Splitting verwendeten Berechnung (3) ist dann $O((\log N)^2 M(N))$ [6], was im Vergleich zur Komplexität der ursprünglichen Berechnung (1) $O(NM(N))$ sehr viel schneller ist.

Es folgt die Kernformel des Binary Splitting. Hierfür sei S_{n_1,n_2} für $n_1, n_2 \in N$ mit $n_1 < n_2$ eine Folge der folgenden Form und seien $a(n), b(n), p(n)$ und $q(n)$ Polynome mit ganzzahligen Koeffizienten:

$$S_{n1,n2} = \sum_{n=n1}^{n2-1} \frac{a(n)}{b(n)} \cdot \prod_{k=n1}^{n} \frac{p(k)}{q(k)} = \frac{T_{n1,n2}}{B_{n1,n2} Q_{n1,n2}} \tag{3}$$

2.3.1 Beweis der Polynomdarstellung

Damit die mathematische Konstante π effizient mittels Binary Splitting berechnet werden kann, muss aber zuerst die in der Aufgabenstellung gegebene Formel (1) umformuliert werden, um die für das binary Splitting verwendeten Polynome zu bestimmen.

$$\begin{aligned}
\pi &= 2 + \sum_{n=1}^{\infty} \frac{n!^2 \cdot 2^{n+1}}{(2n+1)!} \\
&= 2 + \sum_{n=1}^{\infty} \frac{2}{1} \cdot \frac{n!^2 \cdot 2^n}{(2n+1)!} = 2 + \sum_{n=1}^{\infty} \frac{2}{1} \cdot \frac{(2 \cdot 1^2) \cdots (2n^2)}{1 \cdots (2n+1)} \\
&= 2 + \sum_{n=1}^{\infty} \frac{2}{1} \cdot \prod_{k=1}^{n} \frac{2k^2}{2k \cdot (2k+1)} = 2 + \sum_{n=1}^{\infty} \frac{2}{1} \cdot \prod_{k=1}^{n} \frac{k}{2k+1} \\
&= 2 + \sum_{n=1}^{\infty} \frac{a(n)}{b(n)} \cdot \prod_{k=1}^{n} \frac{p(k)}{q(k)}
\end{aligned} \tag{4}$$

Dies beweist, dass Pi durch die folgenden Polynome, in die Formel des Binary Splittings passt, wie es in der Aufgabenstellung angegeben ist.

$$\boxed{a(n) = 2,\ b(n) = 1,\ p(k) = n,\ q(k) = 2n+1}$$

2.3.2 Die Hauptformel

Aus (1) und (4) können wir π wie folgt formulieren:

$$\pi = 2 + \frac{T(1,n)}{B(1,n)Q(1,n)}$$

Da $b(n)$ konstant den Wert 1 hat, gilt auch $B(1,n) = 1 \ \ \forall n \in [1, \infty[$. Deswegen lässt sich π wie folgt berechnen:

$$\pi = 2 + \frac{T(1,n)}{Q(1,n)}$$

2.4 Vergleichsimplementierung

2.4.1 Hex Implementierung

Unsere zweite Version leitet sich vom "Bailey-Borwein-Plouffe BBPAlgorithmus ab. Die BBP-Formel berechnet die n-te Ziffer zur Basis 16 (hexadezimal) von π. Die BBP-Formel zur Berechnung von π lautet:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

Um der Hexadezimalziffern von π beginnend nach den ersten d Hex Ziffern zu berechnen, kann die Formel folgendermaßen umgeschrieben werden [1]:

$$\{16^d \pi\} = \{4\{16^d S_1\} - 2\{16^d S_4\} - \{16^d S_5\} - \{16^d S_6\}\}$$

mit

$$16^d S_j = 16^d \sum_{k=0}^{\infty} \frac{1}{16^d(8k+j)} = \sum_{k=0}^{d} \frac{16^{d-k} \mod 8k+j}{8k+j} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j}$$

Um alle Ziffern von Pi bis zur Zahl n zu berechnen, wird die Formel für jede Zahl bis n angewendet. Die Komplexität dieser Formel ist $O((N \log N) M(\log N))$. [2]

2.4.2 Dezimal Implementierung

Diese Implementierung kann direkt aus der Leibniz-Reihe abgeleitet werden.

$$\pi = 2 + \frac{1}{3}(2 + \frac{2}{5}(2 + \frac{3}{7}(2 + \frac{4}{9}(2 + \frac{5}{11}(2 + \frac{6}{13}(2 + \frac{7}{15}(.....)))))))$$

Diese Formel läuft in quadratischer Zeit. [3]

3 Genauigkeit

3.1 Beschränkung der Eingabegröße

Um n Hexadezimalen Nachkommastellen genau darzustellen, brauchen wir $4n$ binäre Stellen. Die Größe der Eingabe für die Anzahl der binären Nachkommastellen ist auf eine 64 Bit Zahl beschränkt, wodurch die Eingabegröße für die Genauigkeit die Länge des Ergebnisses limitiert.

3.2 Genauigkeit der Division

Die Genauigkeit der Newton-Raphson Division ist stark abhängig von der Anzahl an Iterationen die Ausgeführt werden. Jedoch ist es auch nicht sinnvoll mehr Rechenzeit als nötig in die Division zu investieren, wenn eine geringere Präzision erwartet wird.

Der Initialisierungswert der Division lasst sich mithilfe der folgenden linearen Approximation-Funktion annähern [5]:

$$X_0 = \frac{48}{17} - \frac{32}{17} \cdot N' \approx \frac{1}{N'}$$

Daher haben wir die folgende Absolutwert-Fehlers Funktion:

$$\begin{aligned} |\epsilon| &= |1 - N' \cdot \frac{1}{N'}| \\ &= |1 - N' \cdot (\frac{48}{17} - \frac{32}{17} \cdot N')| \end{aligned}$$

Weil N' nach der Normalisierung über das Intervall von 0,5 bis 1 liegt, beträgt die Funktion folgenden maximalen Absolut-Fehlerwert:

$$|\epsilon| \leq \frac{1}{17}$$

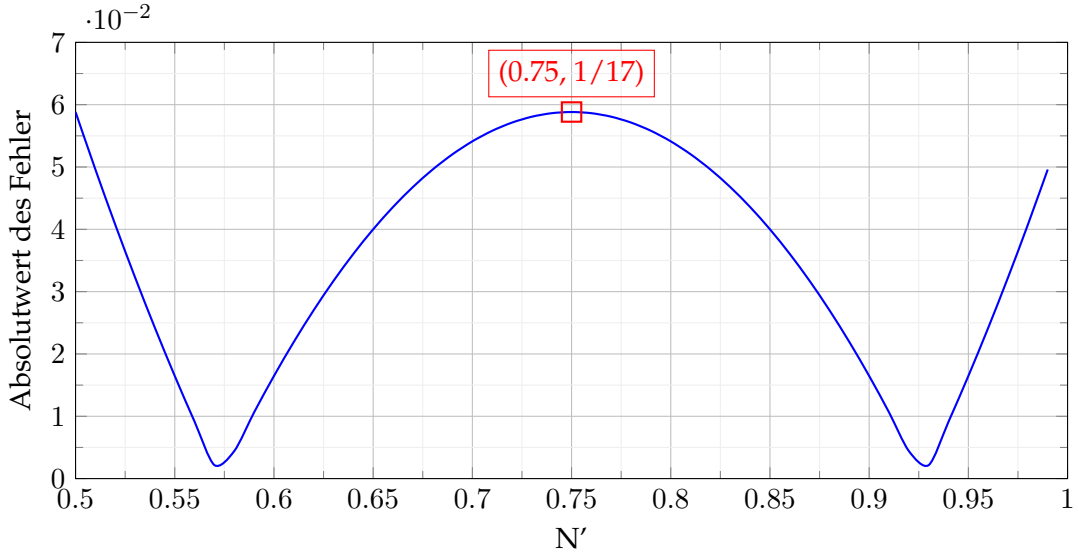

Abbildung 1: Absolutwert-Fehler Funktion

Sei die anfängliche Annäherung X_0, können wir die nächste Näherung der Funktion mit der Newton-Raphson-Methode berechnen, die sich wie folgt darstellt:

$$X_{n+1} = X_n \cdot (2 - N' \cdot X_n)$$

Da die Konvergenz der Newton-Raphson Division quadratisch ist, sich also der Fehler mit jeder Iteration quadriert reichen

$$\lceil \log_2 \frac{n+1}{\log_2 17} \rceil$$

Iterationsschritte um eine Genauigkeit von n Bit sicherzustellen [7].

4 Performanzanalyse

4.1 Test Umgebung

Getestet wurde auf einem System mit einem 11th Gen Intel® Core™ i7-1165G7 Prozessor, 2.80GHz x 8 Cores, 16Gb Arbeitsspeicher, Ubuntu 22.04.1 LTS, Linux Kernel 5.15.0-58-generic. Kompiliert wurde mit GCC 11.3.0 mit der Option -O3.

4.2 Die Performanzergebnisse

Hier werden die Hauptimplementation mit dem Bailey-Borwein-Plouffe Algorithmus verglichen. Getestet wurden mit verschiedener Anzahl von Hexadezimalen Nachkommastelle im Bereich von 1000 bis 500000. Für bessere Übersicht wurde dieser Eingabebereich aufgeteilt.
Mit den Eingabegrößen von 1000 bis 100000 wurde die Berechnung von Pi jeweils 10 mal durchgeführt. Bei diesen Eingabegrößen ist die Hauptimplementierung nicht schneller. Der Unterschied in der Berechnungszeit zwischen Haupt- und Vergleichsimplementierung bei kleinen Eingaben ist ziemlich groß, nimmt aber mit steigender Eingabegröße ab. Erst bei der Berechnung von 100000 Nachkommastellen sind die Berechnungszeiten fast gleich.
Bei den größeren Eingaben von 100000 bis 500000 wurden jeweils 3 mal die Zeit gemessen. Hier erweist sich die Hauptimplementierung als überlegen.

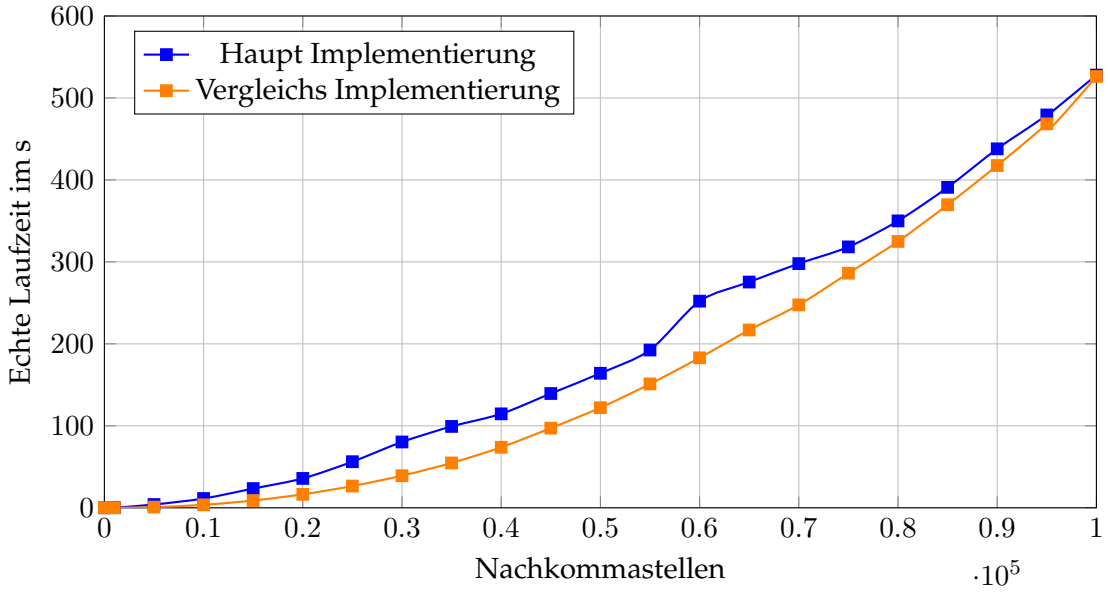

Abbildung 2: Durchschnittliche Laufzeit bei zehn Wiederholungen auf Intel i7-1165G7

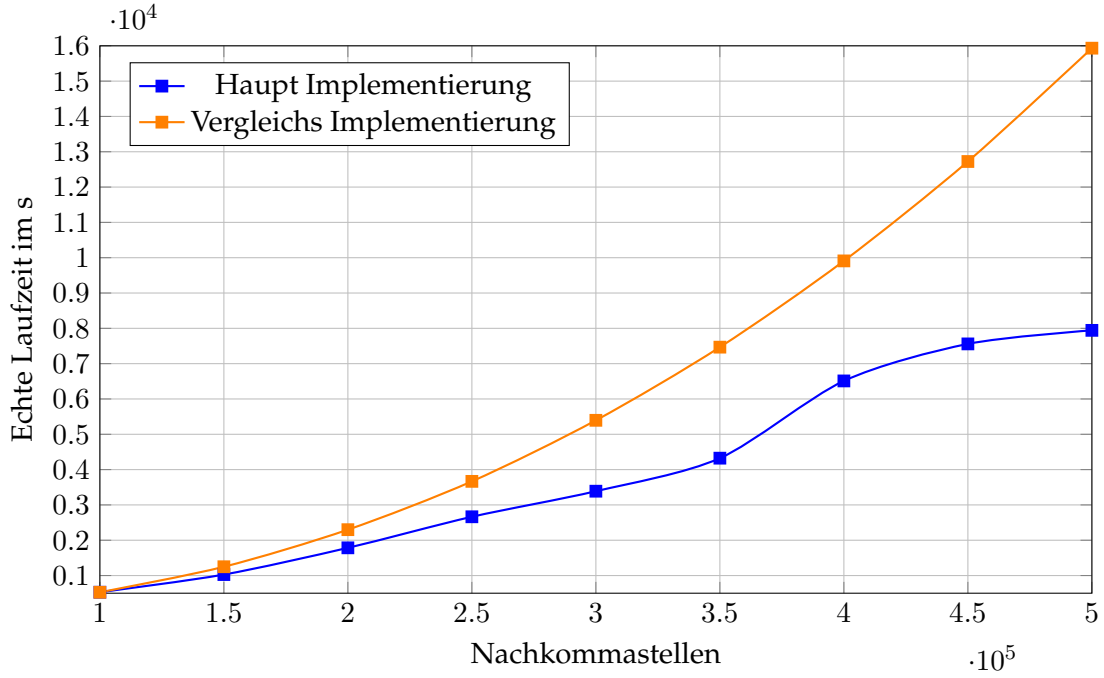

Abbildung 3: Durchschnittliche Laufzeit bei drei Wiederholungen auf Intel i7-1165G7

Die Hauptimplementation braucht 24461.969053 Sekunde, also 6.79492 Stunde, um die erste 1 Milionen Nachkommastellen zu berechnen.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

In dieser Projektarbeit war die Aufgabe Pi mit beliebiger Anzahl an Nachkommastellen zu berechnen. Dafür wurde eine Bignum-Struktur zur 32 Bit Basis mit entsprechender Arithmetik erstellt. Für die Multiplikation wurder der Karazuba-Algorithmus verwendet und für Division die Newton-Raphson Methode. Zur endgültigen Berechnung von Pi wurde Binary Splitting mit den Parametern gemäß der in der Aufgabe angegebenen Formel angewendet.

Im Performanztest ergab sich, dass diese Implemntation bei einer kleineren Anzahl an Nachkommastellen eher langsamer ist, jedoch ab 10^5 Hexadezimalstellen weniger Zeit benötigt.

5.2 Ausblick

5.2.1 Verwendung von 64 Bit Basis

Eine Möglichkeit die Effizienz unserer Implementation zu verbessern, wäre die Struktur mit der Basis 64 anstelle von Basis 32 zu implementieren. Dadurch könnten mehr

Daten auf einmal Verarbeitet werden, wodurch vor allem bei Addition, Subtraktion und Division Zeit gespart würde, da weniger Operationen erfordern wären. Hierfür müsste eine zusätzliche Behandlung der für die Multiplikation entwickelt werden, da die Multiplikation von zwei 64-Bit-Blöcken ein 128-Bit-Ergebnis ergibt, das ohne Änderung zu einem Überlauf führt und falsche Ergebnisse liefert.

5.2.2 Verwendung von SIMD

Die Bit-Shift-Methode wird recht häufig aufgerufen, nicht nur bei der Berechnung, sondern auch bei der Umwandlung des Bingums von hexadezimaler in dezimale Form. Hierfür können SEE-Instruktionen für SIMD verwendet werden, um die Leistung zu optimieren.

Die Addition/Subtraktion bietet auch eine Möglichkeit SIMD anzuwenden. Hierbei könnten 4 Bignum Blöcke in einem 128 Bit Register auf einmal verarbeitet werden. Sowohl die Addition, als auch die Subtraktion von zwei Blöcken kann zu Überträgen führen, welche sich auf die nächsten Blöcken auswirken würden. Da jedoch SIMD hat keine Carry-Flags hat, müsste darauf geachtet werden den Übertrag zusätzlich zu berechnen.

Literatur

[1] David H. Bailey. *The BBP Algorithm for Pi*, September 2006. https://www.experimentalmath.info/bbp-codes/bbp-alg.pdf, visited 2023-01-21.

[2] David H. Bailey, Peter Borwein, and Simon Plouffe. *ON THE RAPID COMPUTATION OF VARIOUS POLYLOGARITHMIC CONSTANTS*. NSERC of Canada. https://www.davidhbailey.com/dhbpapers/digits.pdf, visited 2023-01-21.

[3] Madeleine Jansson. *Approximation of pi*. Lund University/LTH. https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8983341&fileOId=8983342, visited 2023-01-21.

[4] M.J Flynn. *Chapter 5: Division*, September 2002. https://web.stanford.edu/class/ee486/doc/chap5.pdf, visited 2023-01-20.

[5] Pandey, Pawan Kumar and Singh, Dilip and Chandel, Rajeevan. *Proceeding of Fifth International Conference on Microelectronics, Computing and Communication Systems*, September 2020. https://link.springer.com/book/10.1007/978-981-16-0275-7#bibliographic-information, visited 2023-01-21.

[6] Tsz-Wo Sze. *A Fast Self-correcting pi Algorithm*, December 2019. https://arxiv.org/pdf/1912.05319.pdf, visited 2023-01-21.

[7] Vestias, Mario P. and Neto, Horacio C. *Decimal Division Using the Newton–Raphson Method and Radix-1000 Arithmetic*, 2013. https://doi.org/10.1007/978-1-4614-1362-2_2, visited 2023-01-24.