FREIE UNIVERSITÄT BERLIN

Untersuchung von Methoden zur Verbesserung der Laufzeit des Lucas-Lehmer-Test

Bachelorarbeit
zur Erlangung des Grades Bachelor of Science
im Studiengang Informatik

Vorgelegt am 17. November 2022 von

Markus Giese

Erstgutachter	**Zweitgutachter**
Dipl.-Inform. Barry Linnert	Prof. Dr.-Ing. Jochen Schiller

Fachbereich Mathematik und Informatik

Eidesstattliche Erklärung

Ich versichere, dass ich die Bachelorarbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit hat keiner anderen Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass bei Verwendung von Inhalten aus dem Internet ich diese zu kennzeichnen und einen Ausdruck mit Angabe des Datums sowie der Internet-Adresse als Anhang der Bachelorarbeit anzugeben habe.

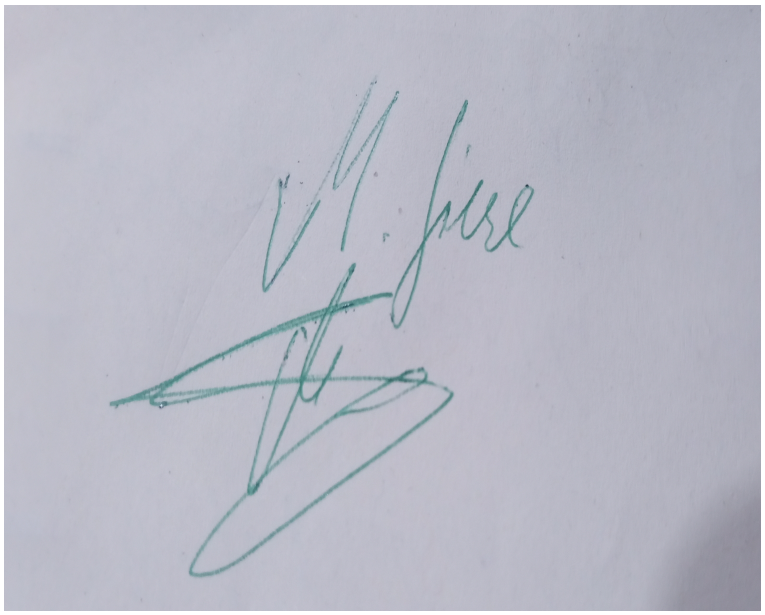

Abbildung 1: Unterschrift

Markus Giese, Berlin, den 17. November 2022

Danksagung

Da diese Arbeit den Abschluss meinen Studiums der Computerwissenschaften darstellt, möchte ich den Leuten danken, die auf dem Weg zu diesem Abschluss eine maßgebliche Rolle eingenommen haben. Ich danke, ganz besonders, meinem Onkel Robert Giese, der mich voll umfänglich bei diesem Studium unterstützt hat, ohne seinen eisernen Support, hätte ich eventuell nicht die Kraft gehabt dieses für mich riesige Projekt abzuschließen. Außerdem gilt mein besonderer Dank Matthias Sekul und Sven Wegner, beide waren maßgeblich am Ende meines Studiums beteiligt und haben meine manchmal doch komplizierte Art und Weise irgendwie ausgehalten und jeder auf seine Weise doch viel zum Abschluss dieser Arbeit beigetragen. Außerdem möchte ich hier zwei besondere Dozenten erwähnen: Frank Hoffmann und Klaus Kriegel. Beide haben mich seit dem Beginn des Studiums mit ihrem Fachwissen und Ihrer Leidenschaft und Expertise nachhaltig beeindruckt. Ohne die Qualität der Mathemodule von beiden oben genannten Dozenten, hätte das Studium wahrscheinlich seinen Reiz für mich verloren. Ich musste für die Mathemodule sehr hart Arbeiten und das heißt studieren ja auch am Ende - sich zu mühen. Meinen Betreuer Barry Linnert, darf ich hier aber auch nicht vergessen, nach beschwerlichen Jahren innerhalb des Studiums, war er es, der am Ende viel Zuversicht verbreitete und auch den Spaß am Fach und Programmieren zurück brachte. Last but not least, möchte ich Maikel Nadolski und Michael Mavroskoufis erwähnen, meine zwei Mafi Tutoren. Beide haben es geschafft ein schon fast volles Glas doch mit Wasser zu füllen. Vielen Dank an alle beteiligten Personen und jene die hier vergessen wurden!

Markus Giese

Inhaltsverzeichnis

Kapitel 1

Einführung

Eine Mersennezahl, ist eine Zahl der Form $2^n - 1$, wobei $n \in \mathbb{N}_+$. Eine Mersenneprimzahl ist eine Mersennezahl, die zusätzlich noch eine Primzahl ist. Bisher gibt es 51 bestätigte Mersenneprimzahlen und wir wissen, dass zwischen den ersten 48 Mersenneprimzahlen keine weiteren Mersenneprimzahlen existieren. Die Mersenneprimzahlen sind die derzeit größten bekannten Primzahlen.

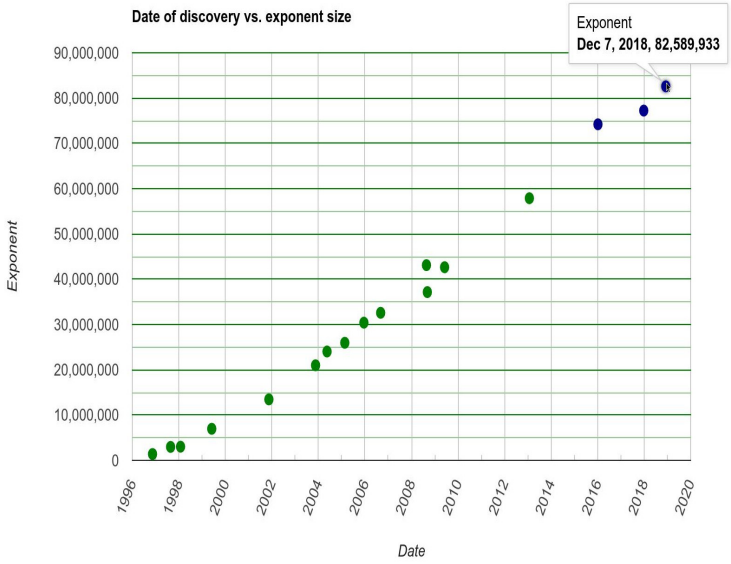

Abbildung 1.1: Zeitliche Entdeckung der bekannten Mersenneprimzahlen [9]

In Abbildung 1.1 ist die zeitliche Entdeckung einzelner Mersenneprimzahlen dargestellt. Wir erwarten nach Primzahlsatz [3], dass die 52. Mersenneprimzahl ungefähr im Jahr 2028 gefunden wird. Dies liegt an der Tatsache, dass die Primzahldichte für größere Zahlen immer geringer wird. Der aktuell schnellste

Algorithmus zur Überprüfung, ob es sich bei einer Mersennezahl um eine Mersenneprimzahl handelt, ist der Lucas-Lehmer-Test (LLT) mit einer Laufzeit von $\mathcal{O}(n^3)$.
In dieser Arbeit untersuchen wir Methoden zur Verbesserung der Laufzeit des LLT. Mittlerweile setzt prime95 [8], welches das derzeit größte Projekt zur Entdeckung von Mersenneprimzahlen ist, in erster Linie Primfaktortests (PRP) ein, um Mersennezahlen im Vorhinein schon auszuschließen. Diese Tests besitzen eine wesentlich bessere Laufzeit als der LLT und eignen sich daher besonders gut. Wir beschränken uns in dieser Arbeit auf den LLT und betrachten solche Primfaktortests nicht weiter. Konkret beschäftigen wir uns in dieser Arbeit aus zeitlichen Gründen mit dem LLT auf amd64/x86_64 Architekturen (Arm,PPC,Sparc und andere Risc Architekturen werden nicht betrachtet). Außerdem betrachten wir keine Algorithmen, die für GPUs optimiert sind. Wir beschränken uns auf Algorithmen, die ausschließlich auf der CPU laufen. Des Weiteren untersuchen wir inwiefern unterschiedliche Datentypen, Programmiersprachen und Multipikationsalgorithmen einen Einfluss auf die Laufzeit des LLT haben. Zudem untersuchen wir, ob wir den Algorithmus parallelisieren können, um eine Laufzeitverbesserung zu erzielen.

Kapitel 2

Vergleichbare Arbeiten

Prime95 von mersenne.org [8] ist das derzeit größte Projekt, das sich mit Mersenneprimzahlentdeckung beschäftigt. Dieses Projekt hat über einen Zeitraum von 20 Jahren den LLT verwendet. Bereits vor 26 Jahren (1996) gegründet geht auf dieses Projekt, unter anderem, die Entdeckung der Mersenneprimzahlen 35-51 zurück [20]. Der Initiator, George Woltman, beschäftigt sich auch mit anderen mathematischen Problemen, beispielsweise wurde der 6. Faktor der 12. Fermatzahl F_{12} von ihm entdeckt.
Mersenne@home ist ein BOINC-Projekt [1], das vor ca. 12 Jahren startete, aber mittlerweile eingestellt ist. Das Projekt hat unter anderem das Ziel verfolgt, dem einzelnen Nutzer zu ermöglichen, die vergebenen Preisgelder für neu entdeckte Mersenneprimzahlen zu verdienen. Es lassen sich leider keine Informationen finden, warum dieses Projekt eingestellt wurde.
Manuel Eberl beweist in seiner Arbeit [6] eine grundsätzliche Eigenschaft der Mersennezahlen. Zudem stellt er eine effiziente Implementierung des LLT vor und beweist in seiner Arbeit mit dem Isabelle Framework [1] seine Korrektheit. Wieb Bosma und Marc-Paul van der Hulst zeigen in ihrer Arbeit [2], wie das Überprüfen auf Primalität durch den Jacobi Summen Algorithmus [23], bis zu 500 Ziffern langen Zahlen, zu einer deutlichen Laufzeitverbesserung geführt hat. Der Algorithmus der Autoren testet mittels Faktorisierung, ob ein gegebenes n zusammengesetzt ist. Hier wird allerdings nicht der LLT verbessert, sondern der Jacobi Summen Test durch den LLT. Andrew Thall zeigte bereits 2007, wie es möglich ist, den LLT effizient auf GPUs mittels CUDA [17] zu implementieren. Mit CUDA und dem entwickelten gpuLucas ist es möglich double precision floating point Unterstützung aktueller Grafikkarten für die Berechnung des LLT zu nutzen [19]. Die Arbeit von D. Cortild und A. Villegas Sanabria beschäftigt sich tiefgehend mit dem LLT. Konkret zeigen die Autoren, dass sich der LLT in $\mathcal{O}(n^3)$ befindet und somit in polynomieller Zeit für ein gegebenes n eine Aussage über die Primalität einer Mersennezahl getroffen werden kann [5].

[1] https://isabelle.in.tum.de

Kapitel 3

Hintergrund

In diesem Kapitel schauen wir uns den LLT an. Dazu klären wir zuerst den mathematischen Hintergrund. Hierfür beweisen wir zuerst allgemeine Sätze über Primzahlen. Dann beweisen wir eine Eigenschaft von Mersenneprimzahlen. Danach schauen wir uns die mathematische Definition des LLT an und zum Schluss die naheliegende algorithmische Implementierung, mit der wir uns im Verlauf der Arbeit weiter auseinandersetzen.

3.1 Mathematischer Hintergrund

Definition 3.1.1 (Primzahlen)**.** *Eine natürliche Zahl $p \in \mathbb{N}_+$ ist eine Primzahl genau dann, wenn $p > 1$ und $a \in \mathbb{N}_+ : a \mid p \Rightarrow a = 1 \vee a = p$.*

Theorem 3.1.1 (Primfaktorzerlegung)**.** *Für $n \in \mathbb{N}_+$ mit $n > 1$ existieren $p_1 < \ldots < p_s$ und $k_1, \ldots, k_s \in \mathbb{N}_0$, so dass*

$$n := p_1^{k_1} \cdot \ldots \cdot p_s^{k_s}.$$

Beweis. Sei T die Menge an natürlichen Zahlen, die keine Primfaktorzerlegung besitzen, d.h.

$$T := \{n > 1 |\ n \text{ ist kein Produkt von Primzahlen}\}.$$

Nun müssen wir zeigen, dass die Menge T leer ist. Angenommen T ist nicht leer. Da T nur natürliche Zahlen enthält und die Menge der natürlichen Zahlen eine totale Ordnung besitzen, existiert das kleinste Element $n_0 \in T$. Für n_0 gilt, dass es keine Primzahl ist, denn ansonsten besitzt n_0 eine Primfaktorzerlegung. Daher existieren die Zahlen $a > 1$ und $b > 1$, so dass $n_0 = a \cdot b$. Zudem folgt $a < n_0$ und $b < n_0$

und somit $a \notin T$ und $b \notin T$. Damit existieren für a und b jeweils Primfaktorzerlegungen. Seien

$$\begin{aligned} a &= \prod_{i=1}^{l} p_i^{k_i} \\ b &= \prod_{j=1}^{m} p_j^{k_j}. \end{aligned}$$

Dann folgt

$$\begin{aligned} n_0 &= a \cdot b \\ &= \prod_{i=1}^{l} p_i^{k_i} \cdot \prod_{j=1}^{m} p_j^{k_j}. \end{aligned}$$

Damit besitzt auch n_0 eine Primfaktorzerlegung. Somit folgt $T = \emptyset$. □

Theorem 3.1.2. *Es existieren unendlich viele Primzahlen.*

Beweis. Sei $P = \{p_1, p_2, \ldots, p_m\}$ die Menge an endlichen Primzahlen und sei $p_1 < p_2 < \ldots < p_m$. Sei nun

$$n := 1 + p_1 \cdot p_2 \cdot \ldots \cdot p_m$$

Da $n \in \mathbb{N}_+$ enthält n eine Primfaktorzerlegung. Daher folgt

$$n = p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_m^{k_m}.$$

Sei $j \in \{1, \ldots, m\}$ mit $k_j > 0$, dann folgt $p_j \mid n$. Nun gilt $p_j \mid n \Rightarrow p_j \mid (p_1 \cdot p_2 \cdot \ldots \cdot p_m)$. Da $p_j \mid n$ und $p_j \mid (p_1 \cdot p_2 \cdot \ldots \cdot p_m) \Rightarrow p_j \mid (n - p_1 \cdot p_2 \cdot \ldots \cdot p_m)$. Daher folgt

$$\begin{aligned} n - p_1 \cdot p_2 \cdot \ldots \cdot p_m &= 1 + p_1 \cdot p_2 \cdot \ldots \cdot p_m - p_1 \cdot p_2 \cdot \ldots \cdot p_m \\ &= 1. \end{aligned}$$

Also $p_j \mid 1$. Widerspruch. □

Definition 3.1.2 (Gruppe)**.** *Eine Gruppe ist ein Paar $(G, \star)$, das aus einer Menge G und einer Abbildung*

$$\begin{aligned} \star : G \times G &\to G \\ (g, h) &\mapsto g \star h \end{aligned}$$

besteht und folgende Eigenschaften erfüllt:

1. *Assoziativität:* $\forall a, b, c \in G : (a \star b) \star c = a \star (b \star c)$.

2. *Neutrales Element:* $\exists e \in G \quad \forall g \in G : e \star g = g = g \star e$.

3. *Inverses Element:* $\forall g \in G \quad \exists g^{-1} \in G : g \star g^{-1} = e = g^{-1} \star g$.

Definition 3.1.3 (Abelsche Gruppe)**.** *Das Paar* $(G, \star)$ *ist eine abelsche Gruppe, wenn* $(G, \star)$ *eine Gruppe ist und* $\star$ *kommutativ ist, d.h.* $\forall g, h \in G$ *gilt*

$$g \star h = h \star g.$$

Definition 3.1.4 (kommutativer Ring mit 1)**.** *Ein Tupel* $(R, 0, 1, +, \cdot)$*, das aus einer Menge* R *mit* $0, 1 \in R$ *und den Abbildungen*

$$\begin{aligned} + : R \times R &\to R \\ (r, s) &\mapsto r + s \end{aligned}$$

und

$$\begin{aligned} \cdot : R \times R &\to R \\ (r, s) &\mapsto r \cdot s \end{aligned}$$

besteht, ist ein kommutativer Ring mit 1, wenn gilt:

1. *Das Paar* $(R, +)$ *ist eine abelsche Gruppe mit neutralem Element* 0.

2. *Das neutrale Element für die Multiplikation ist* 1*, d.h.*

$$\forall r \in R^* : \quad 1 \cdot r = r = r \cdot 1.$$

3. *Die Multiplikation ist assoziativ, d.h.*

$$\forall r, s, t \in R^* : \quad (r \cdot s) \cdot t = r \cdot (s \cdot t).$$

4. *Die Multiplikation ist kommutativ, d.h.*

$$\forall r, s \in R^* : \quad r \cdot s = s \cdot r.$$

5. *Das Distributivgesetz gilt, d.h.*

$$\forall r, s, t \in R^* : \quad r \cdot (s + t) = r \cdot s + r \cdot t$$

Definition 3.1.5 (Körper)**.** *Ein Körper ist ein Ring* $(K, 0, 1, +, \cdot)$ *in dem gilt:*

1. $0 \neq 1$.
2. $\forall r \in K \setminus \{0\} \quad \exists r^{-1} \in K : \quad r \cdot r^{-1} = 1 = r^{-1} \cdot r$.

Definition 3.1.6 (Division mit Rest)**.** *Sei $z \in \mathbb{Z}$ und $m \in \mathbb{N}_+$, dann existieren $s \in \mathbb{Z}$ und $t \in \mathbb{N}$ mit $t < m$, so dass*

$$z = s \cdot m + t.$$

Definition 3.1.7 (Kongruenz)**.** *Seien $a, b \in \mathbb{Z}$ und sei $n \in \mathbb{N}$. Wir definieren*

$$a \equiv b \mod n \Leftrightarrow n \mid (a - b).$$

Definition 3.1.8 (Restklassen)**.** *Sei $n \in \mathbb{N}_+$, dann ist $(\mathbb{Z}/n\mathbb{Z}) =: \mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ ein Ring $(\mathbb{Z}_n, 0, 1, +, \cdot)$ mit n Elementen. Der Ring umfasst die Restklassen, die durch die Division mit Rest durch n entstehen*

$$\begin{aligned}
0 &\equiv \{\ldots, -2n, -n, 0, n, 2n, \ldots\} \\
1 &\equiv \{\ldots, -2n+1, -n+1, 1, 1+n, 1+2n, \ldots\} \\
&\vdots \\
n-1 &\equiv \{\ldots, -2n+(n-1), -n+(n-1), n-1, n+(n-1), 2n+(n-1), \ldots\}
\end{aligned}$$

Für $a \in \mathbb{Z}$ mit $a \equiv b \mod n$ folgt $b \in \mathbb{Z}_n$.

Definition 3.1.9 (Restklassenkörper)**.** *Sei p eine Primzahl, dann ist $\mathbb{Z}_p$ ein Körper. Wir bezeichnen mit $\mathbb{Z}_p^*$ den multiplikativen Körper, d.h. $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$.*

Theorem 3.1.3 (Satz von Euler)**.** *Sei $n \in \mathbb{N}_+$ und $a \in \mathbb{N}$ mit ggT(a, n) = 1, dann folgt*

$$a^{\varphi(n)} \equiv 1 \mod n.$$

Theorem 3.1.4 (Kleiner Satz von Fermat)**.** *Sei p eine Primzahl und $k \in \mathbb{N}_+$, dann folgt*

$$k^p \equiv k \mod p.$$

Beweis. Es gilt für p Primzahl, dass $\varphi(p) = p - 1$. Nach Satz von Euler folgt:

$$\begin{aligned}
k^{\varphi(p)} \equiv k^{p-1} &\equiv 1 \mod p \quad | \cdot k \\
k \cdot k^{p-1} &\equiv k \mod p \\
k^p &\equiv k \mod p
\end{aligned}$$

$\square$

Definition 3.1.10 (Mersennezahl)**.** *Sei $n \in \mathbb{N}_+$, dann ist $2^n - 1$ die dazugehörige Mersennezahl M_n.*

Definition 3.1.11 (Mersenneprimzahl)**.** *Sei $n \in \mathbb{N}_+$ und sei $2^n - 1$ eine Primzahl, dann ist M_n eine Mersenneprimzahl.*

Theorem 3.1.5. *Ist $2^n - 1$ eine Primzahl, dann ist auch $n \in \mathbb{N}_+$ eine Primzahl.*

Beweis. Sei p eine Primzahl und $m \in \mathbb{N}_+$. Sei weiterhin $n = p \cdot m$ und $a := 2^p$, dann folgt

$$\begin{aligned} 2^n - 1 = 2^{pm} - 1 = (2^p)^m - 1 = a^m - 1 \\ &= a^m - 1^m \\ &= (a-1) \cdot \left(a^{m-1} + a^{m-2} + \ldots + a + 1\right) \end{aligned}$$

Wegen $2^n - 1$ ist Primzahl und $a - 1 > 1$ folgt

$$\left(a^{m-1} + a^{m-2} + \ldots + a + 1\right) = 1.$$

Daher $m = 1$. $\square$

Sei p eine Primzahl $\not\Rightarrow 2^p - 1$ ist eine Primzahl. 11 ist eine Primzahl, aber $2^{11} - 1 = 2047 = 23 \cdot 89$ ist keine Primzahl.

Theorem 3.1.6 (Lucas-Lehmer-Test)**.** *Sei p eine Primzahl und $p > 2$. Die Folge $S(k)$ ist wie folgt definiert:*

$$\begin{aligned} S(1) &= 4 \\ S(k+1) &= S(k)^2 - 2 \end{aligned}$$

Es gilt $M_p = 2^p - 1$ ist eine Primzahl $\Leftrightarrow S(p-1) \equiv 0 \mod M_p$.

3.2 Algorithmus Lucas-Lehmer-Test

Wie wir im vorherigen Abschnitt bewiesen haben, muss die Eingabe für den LLT eine Primzahl sein, denn anderenfalls kann die Mersennezahl keine Primzahl sein. Mit dem LLT testen wir daher, ob der eingegebene Exponent eine Mersenneprimzahl bildet.

Algorithm 1 Lucas-Lehmer-Test

Require:
p > 2, a prime number for which M_p , $2^p - 1$ is the Mersenne number to be tested

```
procedure LUCAS-LEHMER-TEST(p)
    s ← 4
    for i ∈ {1, 2, ..., p − 2} do
        s ← s^2 − 2 mod M_p
    end for
    return s == 0
end procedure
```

Algorithmus 1 funktioniert für alle $p > 2$. Die Eingabe p ist der Exponent der Mersennezahl M_p. $S(0)$ wird mit 4 initialisiert, dies ist der Startwert. Dann wird S über eine Schleife von 1 bis $p - 2$ iteriert und immer $S(i)$ berechnet. Wobei $S(i)$ immer $S(i-1)^2 - 2 \mod M_p$ ist.

Wenn nach der letzten Modulo Operation $S(p-2) \equiv 0 \mod M_p$, dann gilt $M_p | S(p-2)$ und somit handelt es sich bei M_p um eine Mersenneprimzahl.

Der Algorithmus determiniert in $\mathcal{O}(n^3)$, was für große Exponenten Jahre dauern kann.

Kapitel 4

Beschleunigung des LLT

In diesem Kapitel schauen wir uns die Möglichkeiten der Beschleunigung des LLT an. Unter Beschleunigung verstehen wir eine Laufzeitverbesserung des LLT gegenüber der naiven Implementierung des Algorithmus 1. Dazu gehen wir zuerst auf die Messplattform ein. Danach betrachten wir die benötigten Hilfsmittel. Dann gehen wir auf die Verbesserungen der Multiplikation und Modulo Operation ein. Zum Schluss untersuchen wir Möglichkeiten der Parallelisierung des LLT.

4.1 Messung der Beschleunigung

Bevor wir uns die verwendeten Hilfsmittel anschauen, betrachten wir zuerst die verwendeten Referenzsysteme und Messmethode.

4.1.1 Referenzsysteme

Wir führen unsere Messungen mit dem Referenzsystem 1 durch.

Referenzsystem 1, Messungen:
AMD Ryzen 5600X, 2*8GB DDR4 2600 Speicher, Samsung PRO 980 NVMe M2 SSD

Referenzsystem 2, sPrime:
Lenovo ThinkCentre M920s mit Intel Core5 8500 Prozessor, 16GB 2400T Speicher, no HDD/SDSystem

Beide Referenzsysteme sind mit einem 1 Gbit/s Netzwerk angebunden. Wenn es nicht genauer spezifiziert ist, dann verwenden wir für die Messung das Referenzsystem 1. Referenzsystem 2 dient nur als Ersatzsystem und für spätere Messungen mit sPrime.

4.1.2 Messmethode

Wir messen die Performance mit Hilfe von Zeitstempeln, da dies in Go im Vergleich zur Messung von CPU-Zyklen einfacher zu realisieren ist.

```
Startzeit := time.Now()

// die zu messende Berechnung (for Schleife)

Berrechnungszeit := time.Since(Startzeit)
fmt.Printf("Berrechnungszeit: %s\n", Berrechnungszeit)
```

Abbildung 4.1: Beispielshafte Zeitstempelmessung in Go

Bei den Messungen haben wir darauf geachtet, dass sich die Systeme im Idle Zustand befinden. Darunter verstehen wir, dass das Testsystem während der Messung nicht von anderen Programmen ausgelastet ist.

4.2 Verwendete Hilfsmittel

4.2.1 Unsere Projektsoftware

Die hier vorgestellten Programme haben wir im Verlauf der Arbeit entwickelt.

prime22 Wir haben prime22 als Messwerkzeug entwickelt. Damit messen wir unter anderem die Geschwindigkeit der gmp Bibliotheken und der verschiedenen Implementierungen der LLT Varianten. Die Messungen haben wir mit den Menüpunkten in Sektion 2 in der Rubrik Algorithmen von prime22 durchgeführt. Zudem haben wir immer wieder verschiedene selbst implementierte Datenbank Kommandos verwendet, um Eigenschaften bestimmter Primzahlen aus der Datenbank auszulesen (prime22,Sektion 4,Menüpunkte A-Z).

sPrime Außerdem entwickelten wir mit sPrime noch ein zweites Messwerkzeug. Die Software kann innerhalb einer Instanz mehrere Datenbanken für mehrere Exponenten verwalten und speichert LLT Iterationen in diesen Datenbanken zwischen. Darüber hinaus können mehrere Instanzen der Software gleichzeitig gestartet werden, um die vorhandenen CPU Kerne durch die Berechnung mehrerer Exponenten gleichzeitig voll auszulasten. Diese Idee wird nicht innerhalb von prime22 umgesetzt, um die einzelnen Programme nicht zu überladen und somit übersichtlicher zu halten.

4.2.2 gmp Bibliotheken

Wir haben Version 6.1.2 und 6.2.1 der „The GNU Multiple Precision Arithmetic Library“ verwendet [22]. Mit diesen ersetzen wir später die Go eigene Multiplikationsfunktion, welche im LLT verwendet wird.

Zudem haben wir neben den Sourcecode Varianten der 6.1.2 und 6.2.1 Version zusätzlich verschiedene weitere Versionen mit Patches benutzt. Diese sind in Tabelle 4.9 aufgelistet.

4.2.3 Rocks Datenbank

Wir verwenden mehrere Instanzen der nosql Datenbank. Rocks DB wird für drei Zwecke benutzt. Der erste Zweck ist eine Primzahldatenbank innerhalb des prime22 Projektes. Sie besteht aus dem Key (Big Integer) und Value, zwei property flags (bool,bool). Die Keys sind alle Primzahlen von 1-340.000.000. Die property flags heißen isChecked und isMersenneprime. Auf diese Weise wird die Datenbank einerseits als Primzahldatenbank benutzt, um beispielsweise einen Primfaktorzerlegungsalgorithmus zu implementieren, anderseits dient sie in zweiter Instanz als Datenbank für Mersenneprimzahlen.

Die dritte Variante wird im Projekt sPrime verwendet. In der hier verwendeten Rocks Datenbank besteht die Key,Value Struktur aus zwei Big Integern. Der erste referenziert einen Index für die Variable S im LLT. Der zweite ist der eigentliche Wert von S zur Zeit der Iteration am entsprechenden Index.

4.3 LLT in Python verbessert mit dem Umstieg auf Go

Python vs. Go Der naive LLT ist die Referenzimplementierung von Wikipedia [18]. Für die Berechnung des Exponenten $p = 756839$ hat der Pythoncode, nach Tabelle 4.1 mehr als 80 Stunden gebraucht. Da es sich bei Python um eine interpretierte Programmiersprache handelt, wollen wir die Berechnungsdauer mit einer compilierten Programmiersprache wie zum Beispiel Go vergleichen, um herauszufinden, ob dies einen signifikanten Einfluss auf die Berechnungsdauer hat.

Die einzige Anforderung an beide Implementierungen ist, dass der LLT bereits Big Integer benutzt. Wir haben keine Optimierungen vorgenommen. Für die Messung haben wir die ersten 14 Mersenneprimzahlen mit den entsprechenden Implementierungen getestet. Dabei haben wir die benötigte Zeit in Sekunden protokolliert.

Go vs. C Warum haben wir uns für die Programmiersprache Go anstelle der hochperformanten Programmiersprache C entschieden? Go verwaltet den Speicher selbst, bietet einen integrierten Debugger und liefert bereits eine Big Integer Bibliothek mit. Go verwendet eine zu C sehr ähnliche Syntax, enthält aber eine Reihe von Vereinfachungen für den Programmierer. Beispielsweise muss nicht jede Codezeile mit einem Semikolon abgeschlossen werden, viele Klammern werden in Go als überflüssig betrachtet. Der Compiler weist einen fortwährend auf überflüssige Klammern und Datentyp Deklarationen hin und zwingt den Programmierer diese zu entfernen. Um den Geschwindigkeitsvorteil von C im Vergleich zu Go zu messen und in ein Verhältnis zu setzen, haben wir C und Go mit der gleichen hoch performantenten Bibliothek gegeneinander getestet.

Weitere Informationen zum Vergleich zwischen C, C++ und Go finden sich hier [12].

Messergebnis und Interpretation

Exponent p	Python Zeit in s	Go Zeit in s
11198	1,83s	0,305s
19937	9,77s	1,19s
21701	12,72s	1,45s
23209	15,5s	1,79s
44497	103s	8,6s
86243	723s	39,82s
110503	1508s	1m18s
132049	2572s	1m56s
216091	-	7m
756839	>81h	1h9m56s

Tabelle 4.1: LLT Python vs. Go

In Tabelle 4.1 testen wir mit dem LLT bis zum Exponenten 756839. Anhand der Messdaten erkennen wir, dass die Go Implementierung beim Exponenten 11213 um einen Faktor von ca. 6 schneller ist. Beim Exponent 756839 ist die Go Implementierung sogar um einen Faktor > 80 schneller.

Der von hier an verwendete Go Code (prime22,Sektion2,Punkt 3) ist bereits deutlich schneller als der ursprüngliche Python Code und nun der Ausgangspunkt für weitere Untersuchungen.

Des Weiteren haben wir Geschwindigkeitsmessungen für Multiplikation und Exponentation mit sehr großen Zahlen durchgeführt. Wir haben die Multiplikation von Zahlen, die $35 \cdot 10^6$ Ziffern enthalten, gemessen.

Ziffernanzahl	Python	Go default	Go gmp	C gmp
$35 * 10^6$	87,4 s	16,849 s	0,321 s	0,321 s

Tabelle 4.2: LLT Python vs. GO

In Tabelle 4.2 finden sich nur die Ergebnisse für Multiplikation wieder. Abgesehen von Rundungsfehlern oder Schedulerungenauigkeiten lieferten die Exponentation / Power Funktionen in allen 3 Sprachen die gleichen Ergebnisse wie die jeweiligen Multiplikationsfunktionen, weswegen diese Ergebnisse hier nicht auftauchen.

Außerdem haben wir zusätzlich zur math / big Bibliothek von Go auch in Go die gmp Bibliothek verwendet. Die Messungen der Multiplikation großer Zahlen haben wir in C und Go mit gmp-6.2.1, der aktuellen Version, durchgeführt. Python ist hier um den Faktor 5 langsamer als die Go Multiplikation und um den Faktor 200 langsamer als Go mit gmp Bibliothek.

Die Go Multiplikation ist um den Faktor 50 langsamer als die schnellste Version der gmp Bibliothek. Wir erkennen, dass sowohl C als auch Go die Multiplikation von $35 \cdot 10^6$ in ungefähr 0,33 Sekunden

durchführen, wenn beide die gmp Bibliothek zum Multiplizieren benutzen. Somit bietet C bei der Multiplikation mit gmp keinen Geschwindigkeitsvorteil gegenüber Go.

4.4 LLT Go verbessert mit Int64 Laufvariable in der Schleife anstatt Datentyp Big Integer

Nachdem wir herausgearbeitet haben, dass sich Go für unser Vorhaben eignet und C in Bezug auf Performance für unseren Anwendungsfall keinen Geschwindigkeitsvorteil bietet, schauen wir uns nun Verbesserungen der Go Implementierung an. Die Standard Go Version des LLT (Prime22, Sektion 2, Punkt 3) benutzt in der for Schleife des LLT Big Integer als Datentyp, was die Größe der Zahlen nur durch den Speicher begrenzt. Es ist beispielsweise ein Preisgeld von 150.000$ ausgegeben [24] für die erste Mersennezahl, welche die Länge von 10^8 überschreitet. Im Verlaufe der Arbeit zeigen wir, dass wir anhand des Exponenten, die maximale Ziffernanzahl von S bestimmen können (Seite 33). Der Exponent $3,5 \cdot 10^7$ besteht aus 8 Ziffern. Mit dem Datentyp Int64 lassen sich Zahlen bis zu 19 stelligen Ziffern darstellen. Int64 als Datentyp für die Schleife ist also ausreichend groß. Deswegen haben wir untersucht welchen Effekt bezüglich der Laufzeit die Verwendung von Int64 anstelle von Big Integer als Datentyp hat.

Messergebnis und Interpretation

Exponent	Go Big Integer	Go Int64
11213	0,30s	0,30s
19937	1,19s	1,19s
21701	1,44s	1,45s
23209	1,78s	1,78s
44497	8,59s	8,59s
86243	39,9s	39,83s
110503	1m18s	1m18s
132049	1m56,7s	1m56,4s

Tabelle 4.3: Big Integer vs. Int64

Anhand von Tabelle 4.3 erkennen wir keine Laufzeitverbesserungen bei dem Umstieg von Big Integer auf Int64. Wir vermuten, dass die Implementierung der Big Integer Bibliothek im Wertebereich von Int64 im Hintergrund weiterhin diesen Datentyp verwendet. Nach kurzer Betrachtung des Quelltextes können wir dies bestätigen. Go verwendet für den Big Integer Datentyp ein Array von unsigned Integern. Auf einer 64-Bit Architektur ist ein unsigned Integer in Go 64-Bit groß. Die Go Implementierung mit Int64 Datentyp findet sich in prime22 im Unterpunkt Algorithmen,Punkt 4.

4.5 LLT Go verbessert durch Modulo Verbesserung

Bei weiteren Untersuchungen der Implementierung stießen wir auf die langsame Modulo Realisierung. Unsere Implementierung des LLT benutzt eine algebraische Struktur, welche Ring genannt wird. Unsere Implementierung des LLT quadriert den Startwert S in einer for Schleife. Die for Schleife läuft $p - 2$ mal durch, wobei p der eingegebene Exponent ist. Dies hat zur Folge, dass S sehr groß wird. Da wir uns aber in einem Ring befinden, kann das quadrierte S immer auf die kleineren Werte der jeweiligen Äquivalenzklasse zurückgeführt werden. Dadurch ist es möglich, die Zahlen maximal klein zu halten, was wiederum zu einer deutlich besseren Laufzeit der Quadrierung führt. In Abbildung 4.2 ist die Berechnung innerhalb der for Schleife der Standard Implementierung dargestellt.

```
s.Mod(vorher.Sub(s.Mul(s, s), big.NewInt(2)), m)
```

Abbildung 4.2: Standard Implementierung modulo Teil

Die Standard Go Implementierung rechnet $S(i)$ in jeder Iteration modulo $2^p - 1$.

```
for s.Cmp(m) == 1 {
// And is big's logical and, Rsh is right shift
s.Add(dummy1.And(s, m), dummy2.Rsh(s, p))
}
if s.Cmp(m) == 0 {
s = zero
}
```

Abbildung 4.3: Die modifizierte Modulo Operation von CJ Enright [4].

In Abbildung 4.3 ist die Variante der Modulo Operation von CJ Enright [4] dargestellt.

Sei $m = 2^p - 1$. Die Implementierung wertet zuerst aus, ob S größer m ist. Für den Fall, dass S größer als m ist, wird S innerhalb der algebraische Struktur, im Ring, so lange verkleinert, bis S kleiner oder gleich m ist. Wenn S gleich m ist, setzt die Implementierung S auf 0.

Dies ist die Umsetzung des hier [21] beschriebenen Verfahrens, den Modulo mit BitShift Operation durchzuführen. Entscheidend ist dabei, dass die Implementierung auf laufzeitintensive Divisionen verzichtet. Diese Modulo Operation funktioniert aber nur, weil $2^p - 1$ ein Spezialfall ist. $2^p - 1$ besteht in der Binärdarstellung nur aus Einsen, daher können wir die Modulo Operation mit Hilfe der logischen UND-Operation und RSH-Operation implementieren. Generell funktioniert diese Modulo Implementierung für beliebige Modulo Operanden nicht.

Messergebnis und Interpretation

Exponent	Go default MOD	Go FastMod
11213	0,30s	0,075s
19937	1,19s	0,347s
21701	1,44s	0,431s
23209	1,78s	0,518s
44497	8,59s	2,7s
86243	39,9s	14,8s
110503	1m18s	29,4s
132049	1m56,7s	47,8s
216091	7m	2m43s
756839	2h36m23s	1h9m56s
859433	3h55m26s	1h39m17s
1257787	9h56m52s	4h12m57s
1398269	13h56m54s	5h36m45s
2976221	–	39h58m31s

Tabelle 4.4: Modulo Verbesserung

Wir erkennen anhand von Tabelle 4.4 eine signifikante Laufzeitverbesserung durch die verbesserte Implementierung der Modulo Operation, die wir von nun an mit FastMod bezeichnen.

4.6 Parallelisierung des LLT

Nachdem wir uns die Verbesserung der Modulo Operation angeschaut haben, schauen wir uns nun an, wie wir den LLT mit Hilfe von Parallelisierung optimieren können. Durch die Parallelisierung erwarten wir eine weitere Verbesserung der Laufzeit der Multiplikation, die sich direkt auf die Laufzeit des LLT auswirkt.

4.6.1 Parallelisierung der Multiplikation

lokale Parallelisierung - der naive Ansatz

Die beiden vorgestellten Implementierungen (original Python und Standard Go Implementierung), verwenden nur einen CPU-Kern. Nach erfolgter Analyse der Implementierung stellten wir fest, dass innerhalb der $p - 2$ Iterationen im LLT, in fast jeder Iteration große Ganzzahlen multipliziert werden. Daher versuchen wir die großen Ganzzahl-Multiplikationen aufzuteilen. Hierfür verwenden wir die Formel:

$$
\begin{aligned}
a &:= \left\lfloor \frac{m}{n-1} \right\rfloor \\
b &:= m - \left\lfloor \frac{m}{n-1} \right\rfloor \cdot (n-1) = m - a \cdot (n-1) \\
m \cdot m &= m \cdot \left(\left\lfloor \frac{m}{n-1} \right\rfloor \cdot (n-1) + \left(m - \left\lfloor \frac{m}{n-1} \right\rfloor \cdot (n-1) \right) \right) \\
&= m \cdot (a \cdot (n-1) + b) \\
&= m \cdot a \cdot (n-1) + b \cdot m
\end{aligned}
$$

Wir teilen die schnell wachsende große Variable S aus der for Schleife im LLT durch die Anzahl an Clienten $n-1$. Wir teilen nur durch $n-1$, um einen Host für die Berechnung des Restes freizuhalten. Auf diese Art und Weise kann beispielsweise jeder Host/Client einen kleinen Teil der eigentlichen Quadrierung durchführen.

Messergebnis und Interpretation

Das Problem, welches nun entsteht, ist, dass alle Hosts die gleiche Aufgabe berechnen und somit sinnlose Arbeit verrichten. Um eine gute Vergleichbarkeit gewährleisten zu können, sind sowohl die Standard Go Implementierung als auch die Go Implementierung mit FastMod abgebildet.

Exponent	Go Mult single threaded	Go FastMod single threaded	Go naiv parallel
11213	0,30s	0,075s	1,09s
19937	1,19s	0,341s	3,85s
21701	1,45s	0,426s	4,73s
23209	1,79s	0,516s	5,81s
44497	8,6s	2,71s	27,45s
86243	39,82s	14,86s	2m17s
110503	1m18s	29,59s	4m53s
132049	1m56s	47,97s	6m36s

Tabelle 4.5: Naive Parallelisierung

Wir erkennen anhand der Messdaten in Tabelle 4.5, dass die naive Parallelisierung drei Mal langsamer ist, als die Go Standard-Multitiplikation. Die Implementierung ist korrekt und liefert das Ergebnis, das wir beim Quadrieren der Eingabe erwarten. Jedoch ist dieser Ansatz, obwohl alle Cores gleichmäßig ausgelastet werden, durch den erzeugten Overhead beim Splitten und wieder zusammenfügen der S Werte in den einzelnen Iterationen viel zu langsam.

Messungen zu dieser Implementierung finden sich in prime22 in Sektion 3, Punkt 4.

lokale Parallelisierung - die Schulmultiplikation

Da der vorangegangene naive Parallelisierungsansatz viel zu ineffizient ist, haben wir nach einem besseren Algorithmus gesucht. Der erste Algorithmus an den wir dachten war jener, den man als Kind in der Schule lernt. Dieser lässt sich gut parallelisieren. Wir implementieren die Quadrierung mit dem aus der Schule bekannten Multiplikationsalgorithmus. Dieser zerlegt den 2. Faktor der Multiplikation in einzelne Ziffern und multipliziert dann jede einzelne Ziffer mit dem 1.Faktor. Dabei speichert der Algorithmus den Offset für jede einzelne Ziffer. Am Ende werden alle Ergebnisse unter Berücksichtigung des Offsets aufaddiert. Dies erzeugt aber einen sehr großen Overhead. Beispielsweise wird eine 10 stellige Zahl dann in 10 Ziffern zerlegt, aus einer Rechnung werden so 20 Rechnungen. Es müssen am Ende 10 Teilergebnisse addiert werden. Um den Overhead etwas geringer zu halten haben wir uns für die folgende Variante entschieden:

Wir teilen den 2. Faktor durch die Anzahl der lokalen Prozessorkerne. Damit verringern wir den Overhead, welcher entsteht, wenn man den 2. Faktor in einzelne Ziffern zerlegt. Zusätzlich lassen wir dann die Multiplikation der Zahlen in den einzelnen Blöcken mit dem 1.Faktor von Goroutinen parallel abarbeiten.

Messergebnis und Interpretation

In Tabelle 4.6 ist der Vergleich der Go Multiplikation mit der aus der Schule bekannten Multiplikationsvariante dargestellt.

Exponent	Go Mult	Go Smul
11213	0,30s	2,2s
19937	1,19s	8,2s
21701	1,45s	10,1s
23209	1,79s	12,1s
44497	8,6s	1m1s
86243	39,82s	4m51s
110503	1m18s	9m16s
132049	1m56s	49m39s

Tabelle 4.6: Vergleich der Standard Go Multiplikation mit unserer implementierten Schulmultiplikation.

Die Schulmultiplikation ist in allen gemessenen Bereichen deutlich langsamer als die Go interne Big Integer Multiplikationsfunktion. Somit hat dieser Parallelisierungsansatz zu keiner Laufzeitverbesserung geführt.

lokale Parallelisierung - durch externe Bibliotheken

Da die frühen Versuche der Parallelisierung, die naive Parallelisierung und die Schulmultiplikation zu keinen veritablen Ergebnissen führten, haben wir uns dazu entschieden andere Multiplikationsalgorithmen zu verwenden. Der erste Algorithmus den wir untersucht haben, ist der Karatsuba-Algorithmus. Da dieser aber wie sich kurz darauf zeigte in der Standard Go Multiplikationsbibliothek [15] verwendet wird und

wir bereits deren Geschwindigkeitsnachteil im Vergleich mit gmp Bibliotheken in Tabelle 4.3 gesehen haben, haben wir uns dazu entschieden, den Karatsuba-Algorithmus nicht weiter zu betrachten.

Weitere Recherchen führten uns dann zur Fast Fourier Transformationen (FFT). FFT Algorithmen sind die heutzutage schnellsten bekannten Varianten große Ganzzahlen zu multiplizieren.

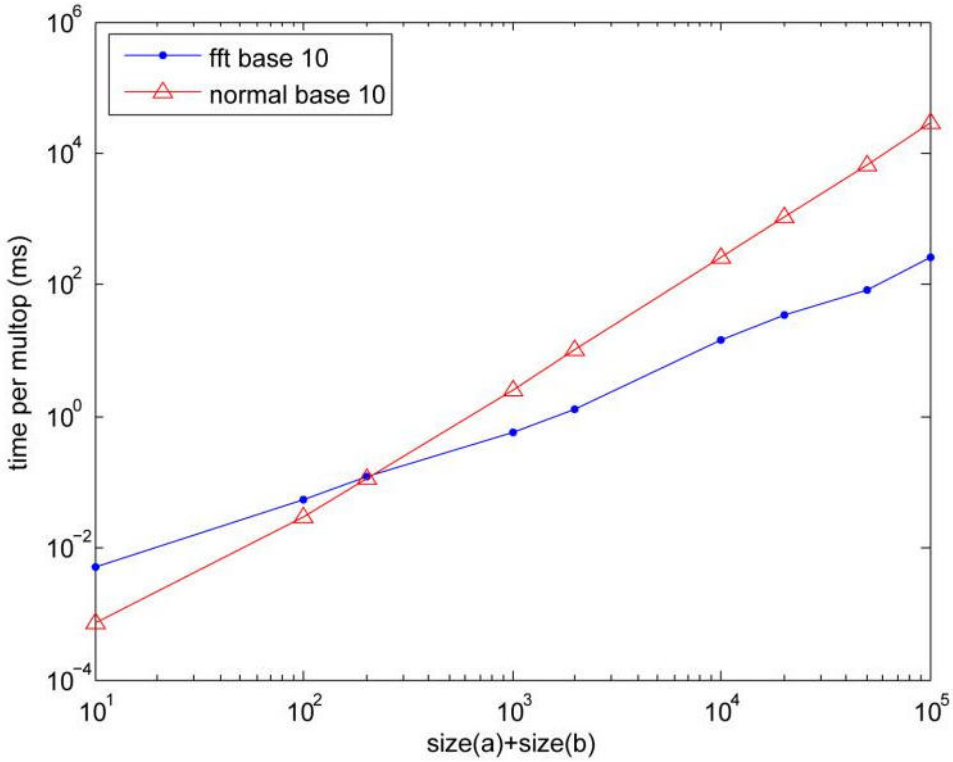

Abbildung 4.4: FFT zur Basis 10 vs. default Basis 10 Multiplikation [7]

In Abbildung 4.4 ist zu erkennen, wie sich durch Verwendung der FFT die Laufzeit der Standard Multiplikation verbessern lässt. Je größer die Zahlen werden, um so vorteilhafter ist die Verwendung von einer Multiplikationsvariante auf Basis von einer FFT.

Es existieren viele spezielle FFT Versionen. Einen groben Überblick, wie sich mit Hilfe der FFT eine Multiplikation realisieren lässt, liefert Abbildung 4.5. Dadurch reduziert sich die Laufzeit der Multiplikation von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n \cdot \log(n))$. Ando Emerencia beschreibt in seiner Arbeit im Detail, wie sich eine solche Multiplikation implementieren lässt [7]. Die hier dargestellte FFT verwendet noch keine Parallelisierung.

$$
\begin{tikzcd}
\begin{array}{l}\mathbf{a} = (a_0, a_1, \ldots, a_n) \\ \mathbf{b} = (b_0, b_1, \ldots, b_n)\end{array} \arrow[r, "\text{Klassische Polynommultiplikation}", "\mathcal{O}(n^2)"'] \arrow[d, "\text{FFT}", "\mathcal{O}(n \log n)"'] & \begin{array}{l}\mathbf{c} = \mathbf{a} * \mathbf{b} \\ \quad = \mathrm{IFFT}(\hat{\mathbf{c}})\end{array} \\
\begin{array}{l}\hat{\mathbf{a}} = \mathrm{FFT}(\mathbf{a}) \\ \hat{\mathbf{b}} = \mathrm{FFT}(\mathbf{b})\end{array} \arrow[r, "\text{Komponentenweise Multiplikation}", "\mathcal{O}(n)"'] & \begin{array}{l}\hat{\mathbf{c}} = (\hat{a}_0 \cdot \hat{b}_0, \hat{a}_1 \cdot \hat{b}_1, \\ \qquad \ldots, \hat{a}_n \cdot \hat{b}_n)\end{array} \arrow[u, "\text{IFFT}", "\mathcal{O}(n \log n)"']
\end{tikzcd}
$$

Abbildung 4.5: Darstellung einer Multiplikation mit Hilfe einer FFT [13]

Eine Übersicht über verschiedene FFT, Radix 2-5, Varianten kombiniert mit besonders schnellen FMA Assembler Instruktionen findet sich hier [10]. Eine dieser Varianten findet sich auch im prime95 Projekt [8] von George Woltman wieder. Es ist zwar eine Sourcecode Variante von prime95 verfügbar, aber die entsprechenden Bibliotheken sind nur als binär Code verfügbar und somit nicht einsehbar. Dennoch konnten wir herausfinden, dass prime95 eine FFT mit FMA3 Instruktionen für die Multiplikation einsetzt.

FFT Algorithmen sind sehr umfangreiche Algorithmen von denen es etliche Varianten und Kombinationen gibt. Diese ließen sich nicht in akzeptabler Zeit implementieren, deshalb haben wir nach bereits existierenden Varianten für Go gesucht.

Die beste verfügbare FFT Version für Go ist die von Rémy Oudompheng [16]. Die Implementierung ist beim Multiplizieren von großen Big Integern im Vergleich zur Go Standard Multiplikation schneller. Da sich das Zusammenfassen der Ergebnisse am Ende der Implementierung parallelisieren lässt, haben wir dies durch Goroutinen realisiert. (prime22,Sektion3, Punkt 3a).

In Abbildung 4.6 ist der ursprüngliche Code von Rémy Oudompheng dargestellt.

```
1  for i := range r.values {
2    r.values[i] = bits[i*(n+1) : (i+1)*(n+1)]
3    z := buf.Mul(p.values[i], q.values[i])
4    copy(r.values[i], z)
5  }
```

bigfft–Mul.go

Abbildung 4.6: Ursprünglicher Code von Remy Oudompheng [16]

```go
type result struct {
  id    int
  zahl fermat
}

var wg sync.WaitGroup
results := make(chan result, len(r.values))


for i := range r.values {

  r.values[i] = bits[i*(n+1) : (i+1)*(n+1)]

  wg.Add(1)
  ii := i

  go func(iii int) {
    defer wg.Done()
    buf := make(fermat, 8*n)
    buf.Mul(p.values[iii], q.values[iii])

    results <- result{iii, buf}
  }(ii)

}
wg.Wait()

a := len(results)
for i := 0; i < a; i++ {
  temp := <-results

  copy(r.values[temp.id], temp.zahl)

}
```

mfft–Mul–goroutines.go

Abbildung 4.7: mfft.go - Mul Funktion, welche die FFT Ergebnisse mittels Goroutinen zusammenfässt.

Diesen Ansatz haben wir mit Hilfe von Goroutinen erweitert. In Abbildung 4.7 ist unser Ansatz dargestellt.

Messergebnis und Interpretation

Exponent	Go Mult	mFFT FastMod	bigfft	mFFT (bigfft + Goroutinen)	Anzahl Ziffern der Mersennezahl/Eingabelänge
11213	0,30s	0,075s	0,30s	0,30s	3376
19937	0,34s	0,34s	1,2 s	1,2s	6002
21701	1,19s	0,43s	1,45 s	1,45s	6533
23209	1,79s	0,51s	1,79 s	1,79s	6987
44497	8,6s	2,7s	8,6s	8,6s	13395
86243	39,82s	14,8s	39,73s	40,1s	25962
110503	1m18s	30,6s	1m18s	1m18s	33265
132049	1m56s	49,9s	2m30s	2m58s	39751
216091	7m	2m43s	7m39s	8m10s	65050
756839	2h36m23s	1h9m56s	2h16m11s	2h21m46s	227832
859433	3h55m26s	1h39m17s	3h2m40s	3h6m46s	258716
1257787	9h56m52s	4h12m57s	7h32m	7h47m55s	378632
1398269	13h56m54s	5h36m45s	10h30m38	10h8m1s	420921
2976221	–	39h58m31s	–	–	895932

Tabelle 4.7: LLT Go vs. bigfft vs. mfft

Eingabelänge	Go Mult	bigfft	bigfft mit Goroutinen (mFFT)
$0,33 \cdot 10^6$	0,009s	0,004s	0,003s
$0,5 \cdot 10^6$	0,019s	0,006s	0,005s
$1 \cdot 10^6$	0,058s	0,014s	0,010s
$2,4 \cdot 10^6$	0,239s	0,039s	0,026s
$5 \cdot 10^6$	0,775s	0,114s	0,071s
$10 \cdot 10^6$	2,3s	0,199s	0,110s
$20 \cdot 10^6$	6,9s	0,413s	0,233s
$35 \cdot 10^6$	16,7s	0,9333s	0,449s
$50 \cdot 10^6$	27,5s	1,49s	0,788s
$100 \cdot 10^6$	1m22s	4,4s	2,2s
$150 \cdot 10^6$	2m57s	4,5s	2,4s

Tabelle 4.8: Multiplikation Go vs. bigfft vs. mfft

Wir erkennen anhand der Tabelle 4.7, dass der FFT Code erst ab einer gewissen Größe effizienter als die Standard Go Implementierung ist. Beispielsweise erreicht die FFT Implementierung bei kleinen Exponenten keinen Geschwindigkeitsvorteil. Die FFT wandelt die zu multiplizierende Zahl zuerst in eine Vektordarstellung um. Dann muss dieser Vektor in ein Polynom transformiert werden, welches dann komponentenweise mit sich selbst multipliziert wird. Dieses Ergebnis wird dann wieder in die Vektordarstellung umgewandelt, um am Ende wieder eine Big Integer Zahl innerhalb unseres LLT zu repräsentieren. Dieser Aufwand lohnt sich erst sobald unser Big Integer im LLT eine Größe von 1800 Maschinenwörtern

(64Bit), also 115.000 Bits, überschreitet. Das entspricht ungefähr einer 14 Kilobyte großen Zahl.

Wir haben es aus zeitlichen Gründen nicht geschafft, den LLT mit genügend großen Exponenten rechnen zu lassen. Deswegen haben wir, um die Ergebnisse einordnen zu können, extra Messungen erstellt, die nur einfache Multiplikationen mit Zahlen mit sehr vielen Ziffern durchführen. Diese Messungen sind in Tabelle 4.8 dargestellt.

Wir erkennen anhand von Tabelle 4.9, dass auch die mFFT genannte Variante, die um Goroutinen ergänzte FFT von Rémy Oudompheng chancenlos gegen die Bibliotheken von gmp ist. Die Tabelle 4.9 enthält Messdaten von verschiedenen gmp Bibliotheken. Wir haben angefangen mit der gmp-6.2.1 Bibliothek aus dem Portage Repository von Gentoo Linux. Bei weiteren Nachforschungen stießen wir auf das Projekt von Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann [14]. Die Autoren stellen einen Patch für gmp-6.1.2 bereit, fortan fT abgekürzt, der gmp-6.1.2 um bis zu 26% beschleunigen kann. Nach umfangreicher Prüfung haben wir herausgefunden, dass dieser Patch offenbar bereits Einzug in den Hauptbranch von gmp gefunden hat und in der aktuellen Version gmp-6.2.1 enthalten ist. Dennoch haben wir über den Umweg dieses Ansatzes herausgefunden, dass wir die schnellste Variante der gmp Bibliothek mit dem manuellen compilieren des Sourcecodes der Version 6.2.1 erzeugen. Diese Version ist, wie wir in Tabelle 4.10 erkennen, beim Multiplizieren von $3,5 \cdot 10^7$ langen Ziffern nochmals 50ms schneller, als die Quellcodevariante von Gentoo´s Portage Version und auch schneller als die gepatchte und optimierte gmp-6.1.2 Version von [14]. Grund dafür sind die verwendeten configure flags. Die selbst compilierte Version benutzt CPU spezifische flags, während beispielsweise Portage amd64-x86 generische flags benutzt.

Exponent	dft Mult	gmp-6.2.1 src	gmp-6.2.1 portage	gmp-6.1.2 src	gmp-6.1.2 fT	mFFT+FM
11213	0,30s	0,062s	0,067s	0,066s	0,066s	0,077s
19937	1,19s	0,232s	0,241s	0,241s	0,240s	0,35s
21701	1,45s	0,273s	0,291s	0,299s	0,299s	0,44s
23209	1,79s	0,320s	0,336s	0,354s	0,354s	0,54s
44497	8,6s	1,39s	1,51s	1,51s	1,5 s	2,7s
86243	39,82s	6,79s	7,32s	7,29s	7,3 s	15,1s
110503	1m18s	11,91s	12,81s	12,77s	12,76s	29,8s
132049	1m56s	18,47s	19,80s	19,84s	19,83s	1m32s
216091	7m	57,06s	1m5s	1m4 s	1m4s	3m12s
756839	2h36m23s	14m33s	15m41 s	15m37s	15m4s	37m12s
859433	3h55m26s	16m44s	18m20 s	18m12s	17m20s	46m45s
1257787	9h56m52s	38m33s	42m15 s	42m1s	39m37s	1h28m20s
1398269	13h56m54s	45m28s	54m54 s	53m38s	46m9s	1h45m2s
2976221	–	3h47m	4h9m	4h8 m	3h33m7s	8h6m6s

Tabelle 4.9: Vergleich der verschiedenen Multiplikationsalgorithmen

Big Integer	gmp-6.2.1 src	gmp-6.2.1 portage	gmp-6.1.2 fT
$3,5 \cdot 10^7$	0.322s	0.377s	0,343s

Tabelle 4.10: Vergleich der verschiedenen gmp Bibliotheken in Bezug auf die Geschwindigkeit der Multiplikation.

Parallelisierung im Netzwerk

Nachdem wir uns gmp angesehen haben, verfolgen wir nun einen anderen Ansatz. Wir versuchen die Rechenlast, die beim Multiplizieren von S innerhalb des LLT entsteht, ab einer gewissen Grenze in kleinere Multiplikationen aufzuteilen und diese auf mehrere Maschinen zu verteilen. Die Idee dahinter ist, dass sich viele kleine Multiplikationen schneller rechnen lassen als eine Große, wie wir in Tabelle 4.8 gesehen haben.

Daher teilen wir ab einem genügend großen Wert von S diesen durch die Anzahl der zur Verfügung stehenden Clienten. Jeder Client berechnet dadurch nur einen kleinen Teil von S. Die zum Server zurückgesendeten Werte werden dann addiert. Eine Multiplikation ist in der Regel viel langsamer als eine Addition und daher erwarten wir, dass wir durch dieses Verfahren einen Geschwindigkeitsvorteil erzielen. Die Implementierung findet sich in prime22 in Sektion 1 unter Punkt 1.

Messergebnis

Exponent	Go Mult	Go FastMod	Go Client-Server 2C	Go Client-Server 4C
11213	0,305s	0,075s	9,6s	10,3s
19937	1,19s	0,341s	37,1s	38,3s
21701	1,45s	0,426s	44,9s	46,5s
23209	1,79s	0,516s	52,2s	54s
44497	8,6s	2,71s	3m57s	4m6,9s
86243	39,82s	14,86s	19m25s	21m2s
110503	1m18s	29,59s	36m39s	40m16s
132049	1m56s	47,97s	1h0m0s	1h3m0s
216091	1m56s	47,97s	3h34m3s	3h49m28s
756839	1m56s	47,97s	–	weniger als 20% nach 15h

Tabelle 4.11: Client Server Ansatz

Interpretation

Wie wir in Tabelle 4.11 erkennen, ist unser Ansatz nicht konkurrenzfähig gegen die bereits erzielten Ergebnisse. Dies liegt vermutlich am zu großen Overhead, der durch das Splitten und Versenden der Daten über das Netzwerk erzeugt wird. Im weiteren Verlauf der Arbeit werden wir auf genauere Untersuchungen insbesondere das Splitten von Big Integer und das Marshalling / Unmarshalling der Zahlen eingehen.

Zusammenfassung

In diesem Teilkapitel haben wir als erstes die ineffektive naive Parallelisierung und die langsame parallele Schulmultiplikation betrachtet. Wir sind dann über die Karatsuba Multiplikation zur FFT gekommen. Diese wurde in Teilen von uns parallelisiert, hat aber im direkten Vergleich mit der gmp Bibliothek bei Geschwindigkeitsmessungen das Nachsehen. Der letzte Ansatz in diesem Kapitel, eine Parallelisierung über das Netzwerk zu benutzen um große Ganzzahlen zu multiplizieren, erzeugt zu viel Overhead und ist am Ende im direkten Vergleich mit gmp nicht schnell genug.

4.6.2 Parallelisierung der Klasse des Problems

Nachdem wir uns die Parallelisierung des LLT angeschaut haben, betrachten wir nun die Parallelisierung der Klasse des Problems. Darunter verstehen wir, dass wir mehrere LLT mit verschiedenen Exponenten gleichzeitig auf einer Maschine mit mehreren Kernen rechnen lassen. Dies führt dazu, dass alle Kerne der Maschine ausgelastet sind.

Hierfür haben wir die Software sPrime entwickelt. Dort können mehrere Instanzen des LLT gleichzeitig gestartet werden.

Damit wir im Falle eines Stromausfalls die Zwischenergebnisse des LLT nicht verlieren, speichern wir die Zwischenergebnisse persistent in einer Datenbank. Der LLT iteriert für jeden Exponenten p genau $p - 2$ mal. Bei einem Beispielexponent 116229631 sind dann mehr als 10^8 Iterationen notwendig. Die Idee hinter sPrime ist nun alle x Iterationen den Wert von S in der for Schleife in einer Datenbank zu speichern, um so den Test jederzeit abbrechen und wieder fortsetzen zu können. Um dies zu testen, haben wir sPrime bei einem Testuser an einem anderen Standort mit dem LLT und dem Exponenten 756839 gestartet.

Wir haben nach 200000 Berechnungen, die Software beendet (STRG+C). Danach haben wir die erzeugte Datenbank von sPrime gepackt (12MB) und per Netzwerk versendet und auf anderer Hardware in das sPrime Projekt eingefügt. Dann haben wir eine neue Berechnung mittels LLT zu starten (sPrime, Punkt l) „continue exponent calculation“ (sPrime, Punkt c) ausgewählt, um die Berechnung fortzusetzen. Nachdem sPrime die Berechnung erfolgreich abgeschlossen hat, haben wir mit „value info“(sPrime, Punkt v) den Wert von S am Index 750000 ausgegeben.

Auf einer anderen Maschine wurde mit einem präparierten LLT bei der 750000. Iteration auch S ausgegeben. Beide Werte stimmten überein. Somit gehen wir davon aus, dass sPrime korrekt funktioniert.

Zudem haben wir einen Stromausfall, mittels Netzkabel ziehen, simuliert, um eine laufende LLT Berechnung in sPrime (sPrime, Punkt l) zu unterbrechen. Auch hier haben wir mit „continue exponent calculation“ (sPrime, Punkt c) die Berechnung fortgesetzt und später das Ergebnis mit einem Standard LLT abgeglichen. Die Ergebnisse stimmten am Ende überein.

Die letzte Simulation von sPrime testet sPrime auf paralleles korrektes Verhalten.

sPrime speichert in statischen Intervallen den Wert S aus der Iteration des LLT zwischen. Bei welchem Index der Wert S in der Datenbank gespeichert wird kann im Quellcode vom Benutzer festgelegt werden. Derzeit steht die Variable $a = 500000$ im LLT von sPrime auf $0,5 \cdot 10^6$. Dieser Wert ist bewusst so groß gewählt um zu gewährleisten, dass 6 Exponenten genügend Werte auf einem 64GB USB-Stick zwischenspeichern können. Der verwendete use-case in dessen Rahmen wir die Software hier getestet haben, ist wie folgt:

Zuerst haben wir den USB-Stick im Referenzsystem 2 verwendet. In diesem PC arbeitet ein 6 Kern Intel Core i5 8500 Prozessor. Wir haben den USB-Stick so konfiguriert, dass 6 Instanzen von sPrime parallel laufen.
Dafür haben wir die Exponenten 11601191,116011997, 116012189,116012201,1160122251,116012293 gewählt.

Für diese Exponenten müssen jeweils über $116 \cdot 10^6$ Iterationen durchgeführt werden. Das entspricht theoretisch $116 \cdot 10^6$ Datenbankeinträgen der Variablen S pro Exponent. Das ist natürlich viel zu viel und würde deutlich über Terabytes hinausgehen. Das Marhalling in und aus der Datenbank darf nicht vergessen werden. Schreib und Leseoperationen in der Datenbank sind bei Zahlen mit mehreren Millionen Ziffern sehr rechenlastig/zeitintensiv.

Daher mussten wir einen guten Kompromiss finden, wie oft bei einem beliebigen Speichermedium zwischengespeichert werden kann.

Wir gehen bei den gewählten Exponenten und der verwendeten Hardware von deutlich über 500 Tagen Berechnungszeit aus. Wählen wir nun die Variable $a = 500000$, so speichert sPrime bei 200000 Iterationen pro Tag und Exponent, ungefähr alle $2,5$ Tage einen Wert von S ab. Dieser Wert ist ca. 15MB, 35MB komprimiert durch Rocks DB, groß. Da wir 6 Exponenten gleichzeitig berechnen und jeder Exponent ca. $116 \cdot 10^6$ Iterationen benötigt, belegen unsere Datenbanken am Ende 232 Speicherungen ·15MB ·6 Exponenten in etwa 21GB. Somit kann ein 32GB USB-Stick für dieses Projekt verwendet werden und hält noch etwas Reserve für das Betriebssystem bereit.

Wir wollen simulieren, wie sich sPrime verhält, wenn der USB-Stick versehentlich oder absichtlich entfernt wird. Nun spielt es für die Geschwindigkeit der Berechnung eine Rolle, wann der USB-Stick konkret entfernt wird. Wird der USB-Stick genau eine Iteration nach der Speicherung eines S, sagen wir zur Demonstration $S(2500001)$ entfernt, kann auch dort weiter gerechnet werden, weil S bei der $2,5 \cdot 10^6$. Iteration gespeichert wurde. Der Zeitverlust ist hier marginal.

Sollte jedoch der Stick beispielsweise bei $S(4999999)$ entfernt werden, so würden wir ca. 2,5 Tage Rechenzeit verlieren, da die letzte Speicherung bereits knapp 2,5 Tage hinter uns liegt und S bei der $2,5 \cdot 10^6$. Iteration letztmalig gespeichert wurde.

In beiden Fällen hat aber die Endberechnung von sPrime mit der „continue“ Variante des LLT zum richtigen Ergebnis geführt. Wir haben die Berechnungen mit angepassten Variablen durchgeführt, da im Rahmen dieser Arbeit keine Zeit war Exponenten der Größe $116 \cdot 10^6$ zu berechnen. Um die Vergleichbarkeit zu gewährleisten haben wir den Exponenten 756839 verwendet und so in allen Experimenten den gleichen Exponent verwendet.

sPrime stellt nicht nur Sicherungen der Ergebnisse der Berechnung verschiedener LLT bereit. Ein entscheidender Vorteil des selbst entwickelten sPrime ist die Tatsache, dass bereits gestartete Rechnungen auf schnellerer Hardware fortgesetzt werden können. Damit ist es dann möglich die Exponenten so schnell wie möglich zu berechnen und die konkrete Laufzeit durch den Umstieg auf andere Hardware jederzeit verbessern zu können.

4.7 Weitere Messungen

In diesem Kapitel finden sich weitere Messdaten, die während des Projektes erhoben wurden und wichtig und interessant sind, sich aber thematisch nicht direkt in ein anderes Kapitel einordnen lassen.

Wir untersuchen, welche Teilschritte im Algorithmus zum Aufteilen der Rechenergebnisse mit der Client Server Architektur wie viel Zeit benötigen. Einer dieser Schritte ist das Aufteilen / Splitten großer Big Integer Werte. Ein weiterer Schritt ist das Marshalling.

4.7.1 Benchmarks Big Integer Splitting

Wir wollen sehr lange Zahlen aufsplitten, um die einzelnen kleineren Teile auf verschiedenen Clienten schneller zu multiplizieren. Dazu möchten wir wissen ob es günstiger ist wenige große oder viele kleine Teile zu erzeugen.

Anzahl Ziffern	Zeit 2 Clienten	Zeit 16 Clienten	Zeit 32 Clienten
80	840ns	2,1μs	3,5μs
800	7,7μs	9μs	10,5μs
8000	158μs	150μs	135μs
80000	6,3ms	3,7ms	3,6ms
400000	116ms	50ms	43ms
800000	418ms	156ms	134ms

Tabelle 4.12: splitting langer Zahlen für die Berechnung auf 2,16 oder 32 Clienten

Wir erkennen anhand von Tabelle 4.12, dass es vorteilhafter für die Laufzeit ist, in kleine Stücke zu splitten. Da mehr CPU Kerne / Clienten auch mehr Daten benötigen, skaliert hier die Laufzeit sehr gut.

4.7.2 Marshalling / Speicherung in der rocks Datenbank

Wir benutzen in sPrime eine rocks DB, um Zwischenwerte S des LLT zu speichern. Dieses Kapitel stellt Daten bereit, wie viel Zeit das Marshalling / Unmarshalling inklusive Speichern / Laden in der Rocks DB für die von uns untersuchten Exponenten $p \geq 116000000$ benötigt. Wir testen, wie lange es dauert eine $3,5 \cdot 10^7$ Ziffern lange Zahl in der Datenbank abzulegen und wieder auszulesen. Wir haben uns für eine $3,5 \cdot 10^7$ Ziffern lange Zahl entschieden, denn wie wir nachfolgend in Tabelle 4.14 sehen werden, haben die S Werte in der for Schleife nicht mehr als $3,5 \cdot 10^7$ Ziffern für den Exponenten 116011991.

Exponent p	Marshalling / Speichern	Unmarshalling / Laden
116011991	4 s	1,78 s

Tabelle 4.13: Exponent 116011991 - Speichern in / Laden aus der rocks DB

Das Speichern dauert mit 4 Sekunden fast 2 mal so lange wie das Laden mit etwa 2 Sekunden. In unserer Erklärung zu sPrime haben wir die Variable a auf $0,5 \cdot 10^6$ gesetzt und damit ergeben sich für unseren Testexponenten p = 116011991 in etwa 232 Speicherungen und somit benötigen wir innerhalb des LLT $232 \cdot 4s$, also ca 15 Minuten. Bei einer zu erwartenden Laufzeit des LLT von 432 Tagen können wir 15 Minuten extra Zeit für den Mehrwert, der durch das Abspeichern entsteht, vernachlässigen.

4.7.3 Bestimmung der maximalen Länge von S im LLT

Wir interessieren uns für die maximale Länge von S im LLT, da wir durch diese möglicherweise die Laufzeit der Berechnung für einen konkreten Exponenten abschätzen können.

Exponent p	Ziffern von $S(max)$	Index von S	Exponent p / $S(max)$
11198	3376	14	3.317
21701	6533	15	3.321
23209	6987	15	3.321
44497	13395	16	3.321
65254697	$\approx$ 19600000	26	3.329
110014039	$\approx$ 33100000	26	3.323
332199001	$\approx$ 100000000	28	3.321

Tabelle 4.14: Exponent - $S(max)$ ratio

Der Wert S im LLT hat ca 1/3 der Länge des jeweiligen Exponenten. Beispielsweise beträgt für p = 65254697 die maximale Länge von $S(max)$ ungefähr $2 \cdot 10^7$ Ziffern, wie an Tabelle 4.14 ablesbar ist. Mit diesem Wissen können wir nun die Berechnungszeit des LLT sehr gut abschätzen.

Die Spalte Index von S, bezeichnet den Index, ab dem S das erste Mal $S(max)$ lang ist. Es gibt jeweils $p-2$ Iterationen im LLT. Somit ist für große Exponenten p, die Anzahl der S, welche kleiner als $S(max)$ ist, vernachlässigbar. Wir können also die Laufzeit des LLT eines Exponenten durch die Laufzeit der

Quadrierung von $S(max)$ multipliziert mit der Anzahl der Schleifendurchgänge abschätzen. Damit ergibt sich die Formel:

$$(O)(LLT(p)) \leq (p-2) \cdot (O)\left(Square\left(\left\lfloor \frac{p}{3.321} \right\rfloor\right)\right)$$

4.7.4 Beispielberechnung der Laufzeit eines Exponenten mit dem LLT

Wir wollen die Laufzeit des LLT für einen beliebigen Exponenten abschätzen.

Unsere Grundlage bilden die Messdaten in Unterabschnitt 4.7.3. Wir vermuten anhand der Messdaten, dass die maximale Anzahl an Ziffern von S in den Iterationen des LLT durch einen konstanten Faktor der der im Verhältnis zum Exponenten p steht berechnet werden kann.

Wir wählen als Beispielexponent 116235187. Teilen wir nun diesen Exponenten durch den Faktor $3,321$, den wir in Unterabschnitt 4.7.3 ermittelt haben, bekommen wir eine erwartete $S(max)$ Länge von $3,5 \cdot 10^7$ Ziffern. Wir wissen, dass der LLT insgesamt $p-2$ Iterationen für den Exponenten p in seiner for Schleife durchführt. Vernachlässigen wir nun die Laufzeit der Subtraktion von 2 und die Modulo Operation in jedem Schleifendurchgang, können wir durch die Laufzeit der Quadrierung von $S(max)$ mit der Länge von $3,5 \cdot 10^7$ Ziffern multipliziert mit der Anzahl der Iterationen von $p-2$ die Laufzeit approximieren. In unserem konkreten Fall ergibt sich nun:

Exponent	Anzahl Iterationen	Laufzeit $S(max)$ Quadrierung	ungefähre Laufzeit des LLT
116235187	116235185	$0,322$ms	$37,4 \cdot 10^7$s

Tabelle 4.15: Beispiel Exponent 116235187

Das entspricht ca. 10400 Stunden und ungefähr 432 Tagen. Diese Rechnung können wir für jeden Exponenten für den LLT anstellen.

4.8 Zusammenfassung

In diesem Kapitel haben wir uns die Methoden zur Verbesserung der Laufzeit des LLT angeschaut. Dabei haben wir das gesetzte Ziel, auf amd64/x86_64 Hardware den LLT zu optimieren erreicht. Wir haben die hier vorgestellten Optimierungen in prime22 implementiert. Dies umfasst eine Kombination der selbst kompilierten Quellcode Bibliothek gmp-6.2.1 und der vorgestellten FastMod Implementierung in der Sprache Go. Zudem kann sPrime benutzt werden, um eine Maschine und dessen Prozessorkerne mit der Berechnung mehrerer Exponenten auszulasten. Auch hier benutzen wir die selbst kompilierte Bibliothek gmp-6.2.1 und die vorgestellte FastMod Implementierung in der Sprache Go. Unsere Versuche durch Parallelisierung eine Exponentenberechnung auf mehrere Maschinen zu verteilen und die Laufzeit der Berechnung des Exponenten zu verbessern, haben nicht zum Ziel geführt. Hier zeigt sich, dass der in diesem Projekt implementierte Code zu viel Overhead erzeugt. Dies liegt vorrangig an unserer Aufteilung

großer Ganzzahlen und der Versendung der codierten Teile über das Netzwerk. Dennoch haben wir in Tabelle 4.8 gesehen, dass sich viele kleine Multiplikationen schneller berechnen lassen als eine Große.

Kapitel 5

Schlusswort und Ausblick

Im Rahmen dieser Arbeit haben wir verschiedene Möglichkeiten zur Laufzeitverbesserung des LLT untersucht. Im ersten Schritt haben wir eine geeignete Programmiersprache bestimmt. Dann haben wir durch die Verbesserung der Modulo Implementierung im LLT eine signifikante Laufzeitverbesserung erreicht. Abschließend haben wir durch die Verwendung der gmp Bibliotheken, die für die Multiplikation FFT einsetzt, eine weitere maßgebliche Laufzeitverbesserung erreicht.
Mersenne.org hat für das Finden der Mersenneprimzahlen die Software prime95 entwickelt. Im Vergleich zu prime95 sind unsere erreichten Laufzeitverbesserungen des LLT vernachlässigbar. Beispielsweise dauert die Berechnung einer Iteration eines beliebigen Exponenten, welcher im Bereich von ca. $65 \cdot 10^6$ liegt, mit unseren Optimierungen ca. 170ms. Prime95 schafft dies jedoch um den Faktor 45 schneller. Dabei setzt prime95 einen optimierten FFT Code, mit unbekanntem Radix, in Kombination mit besonders schnellen FMA3 CPU Instruktionen ein. Aus zeitlichen Gründen haben wir es, innerhalb dieses Projektes, nicht mehr geschafft, die prime95 Software zu disassemblieren. Daher ist es uns nicht gelungen die Laufzeit des LLT bis zu dem Niveau von George Woltman zu verbessern.

Zum Abschluss ein Zitat eines Users aus dem Forum von Mersenne.org, prime95, welcher auf den Zeitumfang eingeht, den George Woltman, Initiator und Hauptverantwortlicher von prime95, verwendet hat, um seinen Speedup zu erreichen :

„George has put a lot of time and talent into improving Prime95's performance, for over a quarter century, including a lot of CPU-model-specific optimizations.“ [11]

Wir sind zuversichtlich, dass wir den LLT mit Hilfe von George Woltmans Algorithmen um ein vielfaches beschleunigen können. Somit gibt es noch viel Potenzial die Laufzeit des LLT zu verbessern.

Literaturverzeichnis

[1] Berkeley Open Infrastructure for Network Computing. Mersenne@home, 2012. https://www.rechenkraft.net/wiki/Mersenne@home_(beendet), besucht am: 2022-10-05.

[2] Wieb Bosma and Marc-Paul van der Hulst. Faster primality testing. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 652–656. Springer, 1989. besucht am: 2022-10-5.

[3] K Chandrasekharan. Der Primzahlsatz. *Einführung in die Analytische Zahlentheorie*, pages 188–199, 1966. besucht am: 2022-09-24.

[4] CJ Enright. Lucas-lehmer-test optimierung using fastmod function, 2019. https://github.com/CJEnright, besucht am: 2022-05-13.

[5] D Cortild and A Villegas Sanabria. Lucas-lehmer primality test, 2022. besucht am: 2022-10-02.

[6] Manuel Eberl. Mersenne primes and the lucas–lehmer test, 2021. besucht am: 2022-10-25.

[7] Ando Emerencia. Multiplying huge integers using fourier transforms, 2007. besucht am: 2022-09-20.

[8] George Woltman et al. Great internet mersenne prime search, 1996-2022. https://www.mersenne.org/, besucht am: 2022-09-30.

[9] George Woltmann et al. Great internet mersenne prime search, 1996-2022. https://www.mersenne.org/primes/, besucht am: 2022-06-27.

[10] Herbert Karner, Martin Auer, and Christoph W Ueberhuber. Top speed ffts for fma architectures. *AURORA Tech. Report TR1998-16, Institute for Applied and Numerical Mathematics, Technical University of Vienna*, 49:50, 1998. besucht am: 2022-08-13.

[11] kriesel. Prime95 sorcery, 2022. https://mersenneforum.org/showpost.php?p=606707&postcount=4, besucht am: 2022-09-28.

[12] Marcin Pasinski. Comparing go vs. c in embedded applications, 2022. https://stackoverflow.blog/2022/04/04/comparing-go-vs-c-in-embedded-applications/, besucht am: 2022-06-02.

[13] Peter Pall et all. Schnelle fourier-transformation, -2022. https://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation, besucht am: 2022-07-05.

[14] Paul Zimmermann Pierrick Gaudry, Alexander Kruppa. A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm, 2007. besucht am: 2022-05-06.

[15] Robert Griesemer, Rob Pike, Ken Thompson. Source code natural numbers - the go programming language, 2012-2022. https://go.dev/src/math/big/nat.go, besucht am: 2022-07-26.

[16] Rémy Oudompheng. Big integer multiplication library for go using fast fourier transform, 2015-2019. https://github.com/remyoudompheng/bigfft, besucht am: 2022-08-12.

[17] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. besucht am: 2022-6-10.

[18] Schwitters et al. Lucas-lehmer-test, 2020. https://de.wikipedia.org/wiki/Lucas-Lehmer-Test, besucht am: 2022-06-13.

[19] Andrew Thall. Implementing a fast lucas-lehmer test on programmable graphics hardware. *Tech. Rep. CIM-007-02*, pages 1–4, 2007. besucht am: 2022-5-06.

[20] various. Liste aller bekannten Mersenne Primzahlen. https://de.wikipedia.org/wiki/Mersenne-Zahl#Liste_aller_bekannten_Mersenne-Primzahlen, besucht am: 2022-09-30.

[21] various. Lucas-lehmer-test time complexity, ?-2022. https://en.wikipedia.org/wiki/Lucas%E2%80%93Lehmer_primality_test#Time_complexity, besucht am: 2022-06-1.

[22] various. The gnu multiple precision arithmetic library, 2000-2022. https://gmplib.org/, besucht am: 2022-09-30.

[23] various. Jacobi sum, 2020. https://en.wikipedia.org/wiki/Jacobi_sumbesucht am: 2022-09-16.

[24] George Woltman and Scott Kurowski. On the discovery of the 45th and 46th known mersenne primes. *Fibonacci Quart*, 46(47):09, 2008. besucht am: 2022-08-16.

Abbildungsverzeichnis

Quellcode Referenzen

Menü template fuer prime22, prime22-client, sPrime:
https://github.com/turret-io/go-menu

Karatsuba Algoritmus (Int64):
https://github.com/NanXiao/golang-algorithm/blob/master/code/karatsuba.go

Fast Fourier Transformation Algorithmus (Big Integer):
https://github.com/remyoudompheng/bigfft

Sieb des Eratosthenes:
https://de.wikipedia.org/wiki/Sieb_des_Eratosthenes

LLT (besonders mit besonders schneller Modulo Rechnung):
https://github.com/CJEnright/Lucas-Lehmer/blob/master/main.go

prime22 - Hauptsoftware dieser BA:
https://github.com/landjalan/prime22

prime22-client - Client Software für Netzwerkcluster, welcher in prim22 enthalten ist:
https://github.com/landjalan/prime22-client

sPrime - USB Stick Linuxdistribution kann mehrere Exponenten gleichzeitg berechnen:
https://github.com/landjalan/sPrime