Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Informatik

Master-Thesis
zur Erlangung des akademischen Grades
Master of Science - M.Sc.

Parallele Algorithmen und die abc-Vermutung

vorgelegt von Jan Gampe

am 08. Mai 2013

Referent: Prof. Dr. Steffen Reith, Hochschule RheinMain
Koreferent: Prof. Dr. Jörn Steuding, Universität Würzburg

Erklärung gem. ABPO, Ziff. 6.4.3

Ich versichere, dass ich die Master-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 08. Mai 2013 Jan Gampe

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Thesis:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der Hochschule RheinMain mit Datenträger	√	
Einstellung der Arbeit in die Bibliothek der Hochschule RheinMain ohne Datenträger	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet	√	

Wiesbaden, 08. Mai 2013 Jan Gampe

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die abc-Vermutung ist eine mathematische Vermutung über die Eigenschaft von abc-Tripeln, formuliert von Joseph Oesterlé und David Masser. Ein abc-Tripel besteht aus drei positiven, natürlichen und paarweise teilerfremden Zahlen a, b und c, für die $a < b$ und $a + b = c$ gilt. Die abc-Vermutung sagt aus, dass es nur endlich viele Ausnahmen für abc-Tripel gibt, für die die Ungleichung $q(a, b, c) \geq \eta$ für beliebige $\eta \geq 1$, $\eta \in \mathbb{R}$ gilt. Dabei stellt q die Qualität

$$q(a, b, c) = \frac{log(c)}{log(r(a, b, c))}$$

und r das Radikal

$$r(a, b, c) =_{def} \prod_{\substack{p|abc \\ \text{p prim}}} p$$

dar. Desweiteren werden *gute* abc-Tripel als solche mit $q(a, b, c) \geq 1.4$ bezeichnet.

Die abc-Vermutung wird wegen ihrer weitreichenden Bedeutung für die Mathematik als der neue „heilige Gral“ bezeichnet. Die Suche nach abc-Tripeln zur Untersuchung der abc-Vermutung durch abc@Home gelangt an natürliche Grenzen des maschinell Überprüfbaren. Bislang wurde der Wertebereich für $a < b < c < 10^{18}$ untersucht und dabei 14.482.065 abc-Tripel entdeckt. Die Anzahl bekannter, guter abc-Tripel hat sich durch Projekte mit anderen Herangehensweisen auf 234 erhöht und gezeigt, dass dies alle guten abc-Tripel bis 10^{30} sind. Ein neuer Ansatz soll die Suche nach guten abc-Tripeln ohne Anspruch auf Vollständigkeit innerhalb eines Wertebereichs angreifen und einen Hinweis darauf liefern, ob die Menge der guten abc-Tripel endlich ist. Neue, gute abc-Tripel könnten sich

sehr nahe an der definierten Grenze der Qualität von $1,4$ befinden, darum ist die Arbeit mit hoher Nachkommapräzision erforderlich.

Die abc-Vermutung ist eng verknüpft mit anderen Sätzen und Fragestellungen der Mathematik. Sollte die abc-Vermutung als zutreffend bewiesen werden, würde sich der Beweis zum großen Satz von Fermat auf eine Seite reduzieren lassen.

Ziel dieser Master-Thesis ist es, parallele Algorithmen zu evaluieren und anzuwenden, welche es ermöglichen, *gute* abc-Tripel zu finden. Im Speziellen sollen geeignete Gitterbasisreduktionsalgorithmen auf ihre Paralleliserbarkeit hin untersucht werden.

Darauf aufbauend wird eine geeignete Parallelisierungsarchitektur ausgewählt und eingesetzt, um die Tauglichkeit empirisch zu belegen, sowie neue, gute abc-Tripel zu finden, die sich außerhalb des Suchraums anderer Projekte befinden. Das gewählte System soll in der Lage sein, auch nach dem Arbeitszeitraum dieser Thesis durch vertretbaren Pflegeaufwand die Suche nach guten abc-Tripeln fortsetzen zu lassen.

Kapitel 2

Grundlagen

2.1 Definition der abc-Vermutung

Seien a, b und c drei positive, natürliche und teilerfremde Zahlen, welche die Gleichung

$$a + b = c$$

erfüllen. Außerdem gilt $a < b$.

Definition 2.1.1 (*Radikal*)

Das Radikal ist definiert als das Produkt jedes Primfaktors von a, b und c. Dabei werden häufiger vorkommende Primfaktoren nur einmal berücksichtigt (Quadratfreiheit).

$$r(a,b,c) =_{def} \prod_{\substack{p|abc \\ p\ prim}} p$$

Definition 2.1.2 (*Qualität*)

Die Qualität q von a, b und c ist definiert als

$$q(a,b,c) = \frac{log(c)}{log(r(a,b,c))}$$

Definition 2.1.3 (*abc-Tripel*)

a, b und c bilden ein abc-Tripel (oder auch abc-Treffer), wenn $r(a,b,c) < c$.

Das Verhältnis zwischen der Summe von a und b und dem quadratfreien Produkt von a, b und c stellt die Besonderheit von ABC-Tripeln heraus. Der Umstand, dass die Addition von Zahlen eine größeres Ergebnis hat als diese spezielle Multiplikation macht diese verhältnismäßig selten.

Definition 2.1.4 (*gute abc-Tripel*)
a, b und c bilden ein gutes abc-Tripel, wenn $q(a, b, c) > 1.4$.

Die Definition von *gut* an dieser Grenze wurde willkürlich gewählt, ist aber bei der Suche nach abc-Tripeln mit hoher Qualität gebräuchlich (siehe [Nit], [ABC]).

Vermutung 2.1.1 (*abc-Vermutung*)
Für jede Zahl $h > 1$ *gilt: Es existieren nur endlich viele abc-Tripel mit* $q(a, b, c) >= h$.

Die abc-Vermutung ist bislang weder bestätigt noch widerlegt. Ein Beweis des Mathematikers Shinichi Mochizuki wird seit August 2012 untersucht. Sein Beweis basiert auf der von ihm begründeten inter-universellen Teichmüllertheorie, einer arithmetischen Variante der sich mit der Klassifikation von kompakten Riemannschen Flächen beschäftigende Teichmüller-Theorie [Moc12].

2.2 Die Suche nach abc-Tripeln

2.2.1 ABC@home

Das Projekt ABC@home der Universität Leiden/NL befasst sich mit der Frage, welche und wieviele abc-Tripel es innerhalb eines bestimmten Wertebereichs für a, b und c gibt. Ziel des Projekts ist es, konkrete Aussagen über diesen Wertebereich machen zu können und nicht, möglichst effizient abc-Tripel bestimmter Güte zu finden. Die Rechenkapazität, um eine solche Aufgabe zu erfüllen, beziehen sie aus einem Pool spendebereiter Heim-PC Nutzer, die über das BOINC Netzwerk Arbeitspakete entgegennehmen und diese in Zeiten ungenutzter Rechenzeit abarbeiten und zurückschicken.

Ein brute-force Ansatz, eine solche Flächendeckung zu erreichen, ist im Algorithmus 1 beschrieben.

Der Algorithmus von ABC@home arbeitet mit einem Zahlensieb, welches früh Zahlenkombinationen ausschließen kann, die nicht die Mindestanforderung an

```
Eingabe : n: zu durchsuchender Wertebereich
Ausgabe : Liste l: alle abc-Tripel im Wertebereich
for a=0 to 10^n do
    for b=0 to 10^n do
        c = a + b;
        if isCoprime(a,b) AND q(a,b,c) > 1 then
            l.add(abc);
        end
    end
end
```

Algorithmus 1: Eine Brute-Force Lösung zur flächendeckenden Suche nach abc-Tripel

die Qualität eines Tripels erfüllt. Das Projekt begann im September 2005. Im November 2011 hat ABC@home den Wertebereich $a < b < c < 10^{18}$ abgesucht und dabei 14.482.065 abc-Tripel gefunden[1]. Als nächster Meilenstein soll der Bereich bis $c < 2^{63}$ ($\approx 10^{18.9}$) überprüft werden [ABC]. Mit $\approx$ 140.000 aktiven Rechnern gehört ABC@Home zu den größeren aktiven BOINC Projekten. Die durchschnittliche verfügbare Rechenleistung beträgt $1,012$ TeraFLOP/s (vgl. [BOI], Stand 03.05.2013).

2.2.2 Überführung in ein Gitterreduktionsproblem

Tim Dokchister hat 2004 einen Algorithmus zur Konstruktion von ABC-Tripeln eingeführt, welcher sich des LLL-Gitterreduktionsalgorithmus bedient [Dok04].

Einführung in die Gittertheorie

Im Folgenden werden einige Begriffe der Gittertheorie näher erläutert, die notwendig sind, um die Idee des Algorithmus zu verstehen. Viele Konzepte der Linearen Algebra finden sich auch in der Gittertheorie wieder und lassen sich analog anwenden.

Weiterführende Literatur zu diesem Thema findet sich im Literaturverzeichnis unter [Sch10].

Gitter sind diskrete Untergruppen des euklidischen Vektorraums $\mathbb{R}^m$. Gitter in $\mathbb{R}^m$ haben die *Dimension* (oder den *Rang*) n. Sie lassen sich über eine Basis von

[1] Das Team von ABC@home führt noch Prüfungen durch, um sicherzustellen, ob der gesamte Wertebereich überprüft worden ist (Stand: 21.12.2012).

Vektoren beschreiben, sodass alle Linearkombinationen dieser Basisvektoren das Gitter erzeugen (aufspannen). Desweiteren können unterschiedliche Basisvektoren dasselbe Gitter erzeugen.

Definition 1 (*Gitter (vgl. [Sch10], Kap. 1)*)

Sei $\mathbb{R}^m$ der reelle Vektorraum der Dimension m und

$$b_1, \ldots, b_n \in \mathbb{R}^m$$

linear unabhängige Vektoren.
Es sei außerdem $B = [b_1, \ldots, b_n] \in \mathbb{R}^{m \times n}$ die Matrix mit Spaltenvektoren $b_1, \ldots, b_n$.
Dann bezeichnet

$$\mathcal{L}(B) = \mathcal{L}(b_1, \ldots, b_n) =_{def} \{B_z \mid z \in \mathbb{Z}^n\} = \sum_{i=1}^{n} b_i \mathbb{Z}$$

ein Gitter mit Basis $B = [b_1, \ldots, b_n]$.

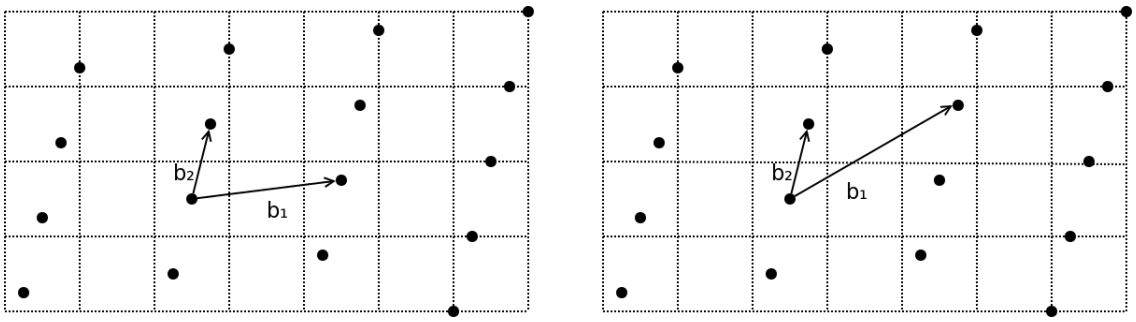

Abbildung 2.1: Dasselbe Gitter in $\mathbb{R}^2$ wird von zwei unterschiedlichen Basen gebildet.

Basisreduktion

Definition 2.2.1 (*L^3 reduzierte Basis (vgl. [Sch10], Kap. 4.1)*)

Eine Gitterbasis $b_1, \ldots, b_n$ heißt LLL (oder auch L^3, nach A. Lenstra, H. Lenstra, Lovász)-reduziert mit δ, wobei $\frac{1}{4} < \delta < 1$, wenn

1. *$\|b_1\| \leq \ldots \leq \|b_n\|$ (Ordnung)*
2. *$|u_{i,j}| \leq \frac{1}{2}$ für $1 \leq j < i < n$ (Längenreduziertheit)*
3. *$\delta \|\hat{b}_{k-1}\|^2 \leq \|\hat{b}_k\|^2 + \mu_{k,k-1}^2 \|\hat{b}_{k-1}\|^2$ (LLL-Eigenschaft)*

wobei $\mu_{i,j} =_{def} \frac{\langle b_i, \hat{b}_j \rangle}{\|\hat{b}_j\|^2}$ *(Gram-Schmidt Koeffizient).* $\|a\|$ *meint die Euklidische Norm.*

Der Grundgedanke der Basisreduktion ist es, zur Basis B eines Gitters $\mathcal{G}$ eine kürzere Basis (im Bezug auf die Längenreduziertheit) B' zu finden, die dasselbe Gitter bildet. Gitter vom Rang 3 oder höher lassen sich effizient mit Hilfe des LLL-Algorithmus reduzieren.

Der Parameter δ beeinflußt die Güte der Reduktion und war ursprünglich auf $\frac{3}{4}$ festgelegt.

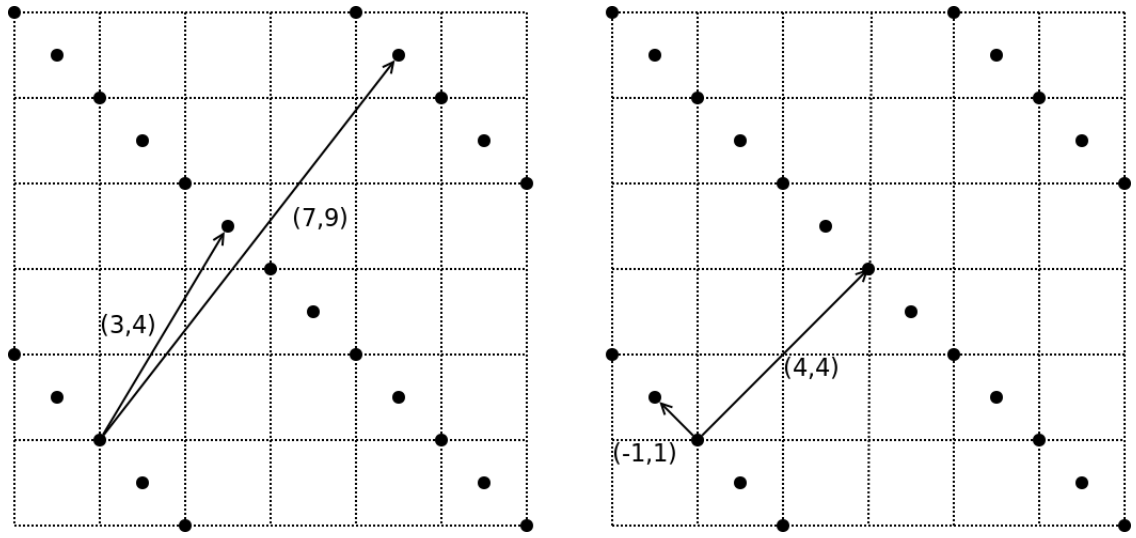

Abbildung 2.2: Dasselbe Gitter in $\mathbb{R}^2$. Die zweite Basis ist reduziert und orthogonal (Beispiel aus [Hel10], Abb. 5).

Anwendung

Der LLL-Algorithmus hat vielfältige Anwendung in der Kryptanalyse gefunden. Unter anderem lassen sich damit diophantische Gleichungen approximativ lösen, indem die Ausgangsmatrix entsprechend gesetzt wird. Dokchisters macht sich diesen Umstand zunutze und stellt die ABC-Gleichung um: $c - a + b = 0$ (s. [Dok04]). Mit beliebigen teilerfremden Startwerten für a, b und c kann der Algorithmus nach einer Lösung für die Gleichung $x_c c - x_a a + x_b b = 0$ suchen lassen. In einem rudimentären Skript ließ Dokchister den Algorithmus über ein Wochenende auf 20 Rechnern verteilt laufen. Dabei wurden unter anderem 41 neue, gute

abc-Tripel entdeckt. Diese Vorgehensweise wird in der Masterarbeit von Matthias Mahl näher erläutert [Mah10].

2.2.3 Einsatz Elliptischer Kurven

Die Idee, abc-Tripel unter Einsatz Elliptischer Kurven zu erzeugen, wurde von Johannes van der Horst am Mathematischen Institut der Universität Leiden untersucht.

Satz 1 (*vgl. [Hor10], Satz 4.1*)

Sei $d > 0 \in \mathbb{N}$, sodass die Elliptische Kurve $E_d : x^3 + y^3 = d * z^3$ linear unabhängige Punkte $P_1, \ldots, P_r$ mit $r \geq 1$ hat. Dann existiert eine Konstante $C > 0$, welche nur von d und der Wahl der Punktemenge $P_1, \ldots, P_r$ abhängt, sodass es unendlich viele paarweise teilerfremde Tripel $(p_n, q_n, r_n) \in \mathbb{Z}^3, n \geq 0$ gibt, sodass gilt:

$$(p_n : q_n : r_n) \in E_d$$

und die dazugehörigen abc-Tripel $(A_n, B_n, C_n) = (dr_n^3, -p_n^3, -q_n^3)$, $n \geq 0$ erfüllen

$$q((A_n, B_n, C_n)) > 1 + \frac{r \, log \, log \, C_n - C}{2 \, log \, C_n}$$

Seine Suchergebnisse beinhalten abc-Tripel in der Größenordnung von bis zu $c \approx 10^{340}$, welche jedoch alle von geringer Qualität ($q \leq 1,05$) sind.

2.2.4 Suchergebnisse

Durch den Einsatz verschiedener Algorithmen sind bislang 234 gute abc-Tripel entdeckt worden. Es wurde außerdem gezeigt, dass darin alle guten abc-Tripel mit $c < 10^{21}$ enthalten sind.

Abbildung 2.3 liefert einen Überblick über die von ABC@home in bestimmten Wertebereichen gefundenen abc-Tripel. Daraus lässt sich ablesen, dass bei jeder Verzehnfachung des Wertebereichs sich die Anzahl der ABC-Tripel ($q > 1$) ungefähr verzweifacht.

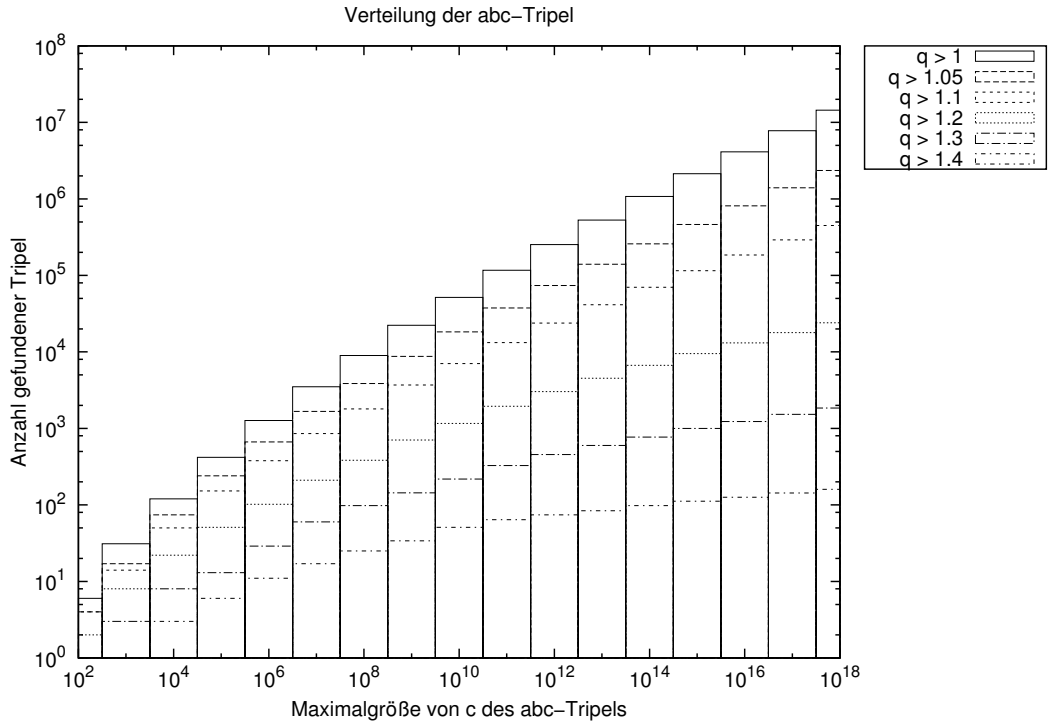

Abbildung 2.3: Verteilung der abc-Tripel für $c < 10^{18}$

2.3 BOINC

Die Berkeley Open Infrastructure for Network Computing ist eine Open-Source Middleware zum verteilten Rechnen, welche es wissenschaftlichen Projekten ermöglicht, gespendete Rechenzeit über das Internet in Anspruch zu nehmen [Ber]. BOINC ist ursprünglich als Middleware für SETI@Home, einem Projekt zur verteilten Analyse von Signalen aus dem Weltraum nach außerirdischer Intelligenz, entstanden und wurde dann als generische Plattform für beliebige Projekte weiterentwickelt.

2.3.1 Aufbau

Jedes BOINC Projekt ist ein klassisches Client-Server System, wobei serverseitig viele Teilkomponenten das Gesamtsystem bilden (siehe Abbildung 2.4). Die Komponenten eines BOINC Systems lauten wie folgt:

- Ein **Work Generator** erzeugt Arbeitspakete (Workunits), aus denen der Scheduler Arbeitsaufgaben (Tasks) für die Applikationen generiert.

- Der **Scheduler** vergibt Tasks an die Clients und nimmt Ergebnisse (Results) entgegen. Dabei verteilt er die vorhandenen Arbeitspakete bestmöglich an die Clients in Abhängigkeit von geschätztem Arbeitsaufwand des Arbeitspakets und Leistungsfähigkeit des Clients.

- **Applikationen** sind die Programme, die auf Seiten des Spenders laufen. Sie stehen serverseitig mit dem Scheduler in Kontakt und erhalten Aufgaben (Tasks) als Eingabe und liefern Ergebnisse (Results) zurück.

- Der **Upload Handler** ermöglicht das Hochladen von Arbeitsergebnissen. Ein zusätzlicher Mechanismus erlaubt die Fortsetzung des Uploads nach Verbindungsabbruch.

- Der **Feeder** bildet das Verbindungsstück zwischen BOINC-Datenbank und Scheduler, da der Scheduler größtenteils aus CGI Skripten besteht und keine direkte Verbindung implementiert worden ist.

- **Validatoren** überprüfen die Korrektheit von Arbeitsergebnissen. Die genaue Implementierung ist projektabhängig. Mitgelieferte Standard-Validatoren prüfen dadurch, dass eine Workunit redundant als zwei oder mehr Arbeitspakete an unterschiedliche Clients vergeben wurde, bestenfalls identische Ergebnisse miteinander.

- Der **Transitioner** überführt Arbeitspakete auf Seiten der Datenbank in unterschiedliche Zustände (initial, in Bearbeitung, ungültig, etc.).

- Der **Assimilator** ist für die Weiterverarbeitung von fertigen Arbeitsergebnissen zuständig (bspw. Übertragung in eine Datenbank)

- **File Deleter** löschen nicht mehr benötigte Arbeitspakete aus dem BOINC System.

- Der **Core Client** wird vom Spender installiert und ist in der Lage, sich mit beliebigen BOINC Servern zu verbinden und deren Applikationen auszuführen.

Die Abbildung 2.4 zeigt die Kompontenten in ihrer jeweiligen Beziehung. Projektspezifische Komponenten sind der Work Generator, der Validator, der Assimilator sowie die zu verteilende Applikation.

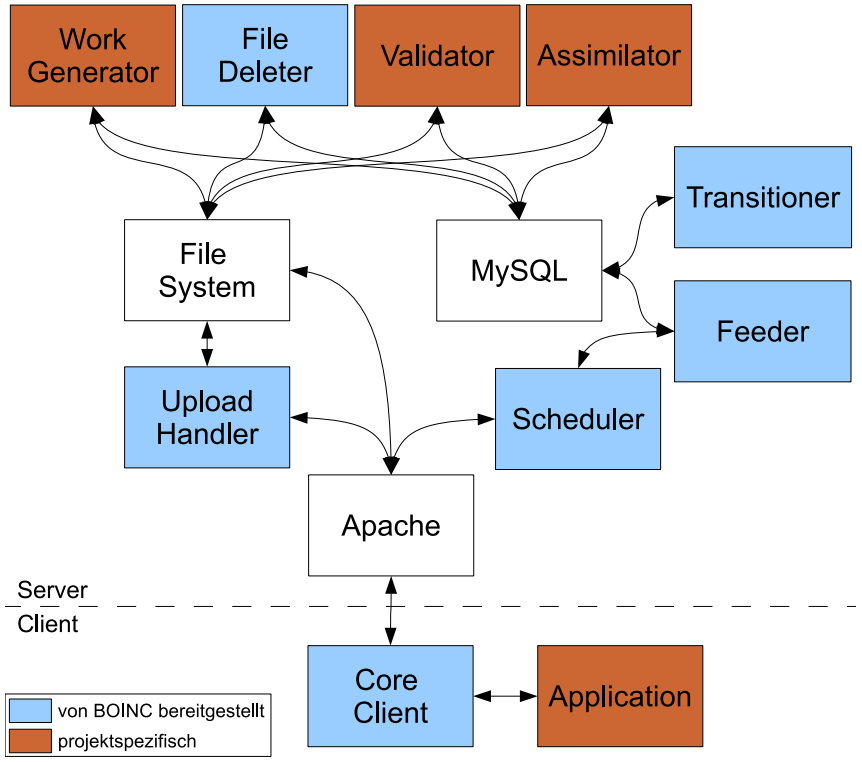

Abbildung 2.4: Die BOINC Architektur (aus [Flo])

2.3.2 Leistungsdaten

Die durchschnittliche Zahl der aktiven freiwilligen Spender beträgt ungefähr 250.000 Einzelpersonen mit 310.000 Rechnern, die Rechenkapazität aller aktiven Spender liegt im Tagesdurchschnitt bei rund 7400 TeraFLOP/s (Quelle: [Ber], Stand: Mai 2013). Ein direkter Vergleich in Bezug auf die Rechenkraft mit Großrechnern ist möglich, dabei ist aber zu beachten, dass einige Faktoren zu Ungunsten von BOINC berücksichtigt werden müssen. Je größer ein BOINC Projekt ist, desto größer müssen Redundanzen in der Arbeitsverteilung ausgelegt werden, um Fehlberechnungen, Fehler und Betrugsversuche von Benutzern zu minimieren. Wenn es nicht oder nur eingeschränkt möglich ist, die einzelne Arbeitsergebnisse zu validieren, müssen sie mehrfach an unterschiedliche Benutzer vergeben werden, um das Arbeitsergebnis zu bestätigen.

Platz	Name	Leistung	Standort
1	Titan	17590 TeraFLOP/s	USA
2	Sequoia	16324 TeraFLOP/s	USA
3	K computer	11280 TeraFLOP/s	USA
4	Mira	8162 TeraFLOP/s	USA
5	JUQUEEN	4141 TeraFLOP/s	Deutschland

Tabelle 2.1: Die Top 5 der schnellsten Supercomputer der Welt. Als Leistungsmaß wurde der LINPACK Benchmark herangezogen. (Quelle: www.top500.org, Stand: Mai 2013)

Kapitel 3

Entwurf

3.1 Ziel der Arbeit

In dieser Arbeit geht es darum, Algorithmen, mit denen sich nach guten abc-Tripeln suchen lässt, auf ihre Parallelisierbarkeit hin zu untersuchen, sowie passende Paralleliserungsframeworks dafür zu finden. Der aussichtsreichste Algorithmus soll mit Schwerpunkt auf Skalierbarkeit implementiert und dessen Leistungsfähigkeit gemessen werden.

Da der Wertebereich für $c < 10^{18}$ vollständig abgesucht worden ist und der für $c < 10^{30}$ durch andere Projekte bereits bearbeitet wird, soll mit dem gewählten Algorithmus außerhalb dieses Bereichs gearbeitet werden.

Die Leistungsfähigkeit des Algorithmus wird daran gemessen, wieviele Tripel in welcher Qualität und mit welchem Aufwand gefunden werden können.

3.2 Wahl des Algorithmus

Die Entscheidung, welcher Algorithmus für diese Arbeit geeignet ist, fiel auf den ABC-LLL Algorithmus nach Dokchister (siehe Abschnitt 2.2.2). Dieser überzeugt vor allem dadurch, dass er schon in der Vergangenheit gute Ergebnisse geliefert hat. Außerdem bietet er noch einige Variationsmöglichkeiten, die zu untersuchen sind.

3.2.1 Beschreibung

Der folgende Algorithmus wurde das erste Mal von Dokchsiter (s. [Dok04]) beschrieben und 2010 von Mahl (s. [Mah10]) vertiefend untersucht. Der in diesem Projekt eingesetzte Algorithmus ist hauptsächlich aus den Vorgaben von [Mah10], Kapitel 4 entnommen, angepasst und in Beschreibungslücken vervollständigt worden.

Der Übersicht halber wird der Algorithmus in seinen Teilkomponenten *findeKoeffizienten* und *TripelAusKettenbruch* beschrieben.

3.2.1.1 findeKoeffizienten

findeKoeffizienten ist die Komponente, die den LLL-Algorithmus zur Lösung diophantischer Gleichungen verwendet. Als Eingabe dienen beliebige teilerfremde Zahlen a_0, a_1, a_2, im weiteren Verlauf **Basis-Tripel** genannt. Die Ausgabe bildet eine Liste von Koeffiziententripeln (x_1, x_2, x_3), für die gilt:

$$x_1a_1 + x_2a_2 + x_3a_3 = 0$$

Algorithmus 2 (findeKoeffizienten) läuft im Detail wie folgt ab:

1. Setze $m = \lfloor\, 2^2 \, \|a\|_2^{\frac{1}{2}} \,\rfloor$ wobei $\|a\|_2$ die Euklidische Norm des Vektors

   $$\|a\|_2 = \sqrt{(a_1)^2 + (a_2)^2 + \ldots + (a_n)^2}$$

   meint.

2. Bilde ein Gitter $\mathcal{G} \subseteq \mathbb{R}^4$, welches von den Zeilen der Matrix

   $$G := \begin{pmatrix} 1 & 0 & 0 & ma_0 \\ 0 & 1 & 0 & ma_1 \\ 0 & 0 & 1 & ma_2 \end{pmatrix}$$

   erzeugt wird.

3. Wende die LLL-Reduktion auf das Gitter an. Die daraus resultierende Matrix

Eingabe : Basis-Tripel (a_0, a_1, a_2)
Ausgabe : Liste von Koeffiziententripeln : result
$m = \lfloor 2^2 \, \|a\|_2^{\frac{1}{2}} \rfloor$;
$G := \begin{pmatrix} 1 & 0 & 0 & ma_0 \\ 0 & 1 & 0 & ma_1 \\ 0 & 0 & 1 & ma_2 \end{pmatrix}$;
lllReduziere(G);
foreach *row : G* **do**
  if *row[0] != 0 AND row[1] != 0 AND row[2] != 0 AND row[3] == 0* **then**
    koeffiziententripel k;
    if *signum(row[0]) == signum(row[1])* **then**
      k = (abs(row[0]), abs(row[1]), abs(row[2]));
    end
    else if *signum(row[0]) == signum(row[2])* **then**
      k = (abs(row[0]), abs(row[2]), abs(row[1]));
    end
    else
      k = (abs(row[1]), abs(row[2]), abs(row[0]));
    end
    triple t = $(k_0 * a_0, k_1 * a_1, k_2 * a_2)$;
    if *erfuelltABCBedingungen(t)* **then**
      result.add(t);
    end
  end
  return result;
end

Algorithmus 2: findeKoeffizienten

wird nun wie folgt betrachtet:

$$G_{reduziert} := \begin{pmatrix} u_1 & u_2 & u_3 & u_4 \\ v_1 & v_2 & v_3 & v_4 \\ w_1 & w_2 & w_3 & w_4 \end{pmatrix}$$

Mindestens eine Zeile hat in der vierten Spalte keine Null.

4. Betrachte die Elemente (x_1, x_2, x_3) jeder Zeile x der Matrix, welche ausschließlich in der vierten Spalte eine Null hat.
   (a) Bereite ein Koeffiziententripel (r_1, r_2, r_3) vor.
   (b) Das Tripel (x_1, x_2, x_3) hat ein Element, welches ein anderes Vorzeichen als die anderen beiden besitzt. Dieses Element wird für r_3, die jeweiligen anderen Elemente aufsteigend nach ihrer Größe für r_1 und r_2 eingesetzt.
   (c) Prüfe, ob $r_1a_1 + r_2a_2 + r_3a_3 = 0$. Wenn dem so ist, wird das Koeffiziententripel in die Ergebnisliste eingetragen.

Mit dieser Teilkomponente lassen sich bis zu zwei ABC-Tripel pro Basis-Tripel finden. Liefert der Algorithmus zwei Koeffiziententripel, können durch Kettenbruchentwicklung weitere Koeffiziententripel erzeugt und damit weitere abc-Tripel gefunden werden. Dafür wird *TripelAusKettenbruch* ausgeführt.

3.2.1.2 TripelAusKettenbruch

Wenn

$$v_1a_1 + v_2a_2 + v_3a_3 = 0$$

und

$$w_1a_1 + w_2a_2 + w_3a_3 = 0$$

dann liefern auch Kombinationen $\lambda_1 v + \lambda_2 w = u$ neue, korrekte Koeffizienten. Eine Aufgabe des Algorithmus besteht nun darin, λ_1 und λ_2 so zu wählen, dass ein oder mehrere Elemente des Koeffiziententripels möglichst klein sind, um ein abc-Tripel mit möglichst kleinem Radikal und damit potenziell hoher Qualität zu erhalten.

Eingabe : Koeffiziententripel v und w, BasisTripel basis
Ausgabe : Liste neu gefundener abc-Tripel : result
for *i = 0 to 2* **do**
    list<bruch> appro = berechneApproximationsbrueche($\frac{v_i}{w_i}$);
    x = (signum(v_i) == signum(w_i)?1:-1);
    foreach $\frac{z}{n}$ *: appro* **do**
        koeffiziententripel k = z * v + x * n * w;
        tripel t = ($k_0 * basis_0, k_1 * basis_1, k_2 * basis_2$);
        if *erfuelltABCBedingungen(t)* **then**
            result.add(t);
        end
    end
end
return result;

Algorithmus 3: tripelAusKettenbruch

3.3 Suchmuster

Für den Suchalgorithmus müssen Basis-Tripel erzeugt werden, von denen ausgehend abc-Tripel gefunden werden können.

Der entwickelte Generator von Start-Tripeln arbeitet in zwei Phasen. In der ersten Phase wird bestimmt, welche Primfaktoren welcher Zahl (a, b oder c) zugeordnet werden sollen. Dazu werden drei Mengen A, B und C eingeführt. Diese Mengen haben die folgenden Eigenschaften:

- $A, B, C \subseteq \mathbb{P}_{20}$
  (A, B und C bestehen ausschließlich aus Elementen der Menge der ersten 20 Primzahlen)
- $A \cap B = B \cap C = A \cap C = \emptyset$
  (A, B und C haben paarweise leere Schnittmengen)
- $\#A, \#B, \#C \leq 4$
  (Die Mengen bestehen jeweils aus maximal 4 Elementen)

Bislang gefundene, hochqualitative abc-Tripel bestehen häufig aus wenigen, kleinen (bis zur 20. Primzahl) und ein bis zwei großen Primfaktoren. In einer Testreihe wurden Start-Tripel aus guten abc-Tripeln erzeugt, denen die letzten großen Primfaktoren (größer als die 20. Primzahl) fehlten. Der Suchalgorithmus ist häufig in der Lage, die original abc-Tripel wieder herauszufinden.

In der zweiten Phase werden alle gültigen Kombinationen möglicher Mengen weiter bearbeitet. Diese erzeugt für jede Eingabe (A, B, C) alle Zahlen a, b und c, für die gelten:

- Die Primfaktorzerlegung von a besteht einzig aus Elementen von A (b aus B und c aus C analog).
- a, b und c haben eine vorher definierte Minimal- und Maximalgröße.
- a, b und c müssen ungefähr gleich groß sein (Empfehlung von Dokchister, [Dok04], S.164). In diesem Projekt wird als Grenze

  $$max(log_{10}(a), log_{10}(b), log_{10}(c)) - min(log_{10}(a), log_{10}(b), log_{10}(c)) <= 10$$

  evaluiert.

Für die erste Phase wird ein Algorithmus eingesetzt, der alle Kombinationen von gültigen Mengen in eine numerische Reihenfolge bringt. Jede Verteilung von Primzahlen auf drei Mengen kann durch eine Zahl zur Basis 4 dargestellt werden. Die Codierung lautet dabei wie folgt:

- Die n-te Stelle des Codes bezieht sich auf die n-te Primzahl. Es wird bei der Stelle mit der geringsten Wertigkeit zu zählen begonnen (analog zum *Einer* im Dezimalsystem).
  - Eine Null bedeutet: Diese Primzahl gehört zu keiner der drei Mengen.
  - Die Zahlen 1-3 bedeuten: Diese Primzahl gehört in die Menge A (=1), B(=2) oder C(=3).
- Nicht codierte Primzahlen sind wie Code 0 zu interpretieren (so wie führende Nullen bei Rechnungen weggelassen werden).

So ist es möglich, über jede mögliche Kombination von Mengen zu iterieren, indem eine Zahl x zur Basis 4 hochgezählt, decodiert und auf die anderen Bedingungen an die Zahl überprüft wird. Die Schleife endet, wenn x so viele Stellen wie die Anzahl an zu verteilenden Primzahlen hat.

Beispiel 1

Die Zahl 3123002 zur Basis 4 codiert die Verteilung von Primfaktoren wie folgt:

- $A = \{13\}$
- $B = \{2, 11\}$
- $C = \{7, 17\}$

3.4 Parallelisierung

Der Suchalgorithmus benötigt als Eingabe ein Basis-Tripel, welches als Ausgangspunkt der Suche dient. Um ein größeres Suchfeld abzudecken, wird eine Liste solcher Tripel erzeut. Da jede Eingabe unabhängig bearbeitet werden kann, findet die Parallelisierung auf dieser Ebene statt.

An dieser Stelle ist kein Kommunikationsbedarf zwischen den Teilberechnungen notwendig, welcher die Skalierbarkeit des Algorithmus beeinträchtigen würde. Es

```
Eingabe : start: Liste aller Basis-Tripel
Ausgabe : res: gefundene abc-Tripel
for triple t : start-tripel do in parallel
    list<abcTriple> lllresult = LLLABC(t);
    res.addAll(lllresult);
end
```

Algorithmus 4: Paralellisierung des LLL-ABC-Suchalgorithmus

wird außerdem eine Verteilbarkeit erreicht, an der ein großes System partizipieren kann, wobei einzelne Rechnerinstanzen nicht ständig erreichbar sein müssen.

Durch diese Eigenschaften ist BOINC als Parallelisierungs-Framework für diesen Algorithmus geeignet. Die Vorteile eines lokalen Supercomputers (feste Bindung zwischen Recheninstanzen, Möglichkeiten zur Interprozesskommunikation, ständige Erreichbarkeit jeder Instanz) mit gleicher Rechenkapazität fallen nicht ins Gewicht.

Die Entscheidung, BOINC einzusetzen, hat außerdem Vorteile in der besseren Vereinbarkeit mit dem Software-Entwicklungsprozess. Es ist jederzeit möglich, die BOINC Applikation zu aktualisieren und zu verbessern. Desweiteren erhält man mit BOINC eine Managementschnittstelle, die bei der Fehleranalyse und Leistungsmessung hilft.

3.5 Übersicht

Das fertige Programm arbeitet mit den folgenden Komponenten:

- Langzahlbibliotheken für Gleitkommazahlen und Ganzzahlen. Diese sind notwendig, da die zu untersuchenden Zahlenbereiche außerhalb der aktuell üblichen 64-bit Maschinenwörtern liegen.
- Eine Bibliothek zur LLL-Reduktion von Gittern.
- Eine Bibliothek zum Faktorisieren großer Zahlen.
- Die BOINC API, um mit dem BOINC Client zu kommunizieren und die Betriebssystemschicht zu erreichen.

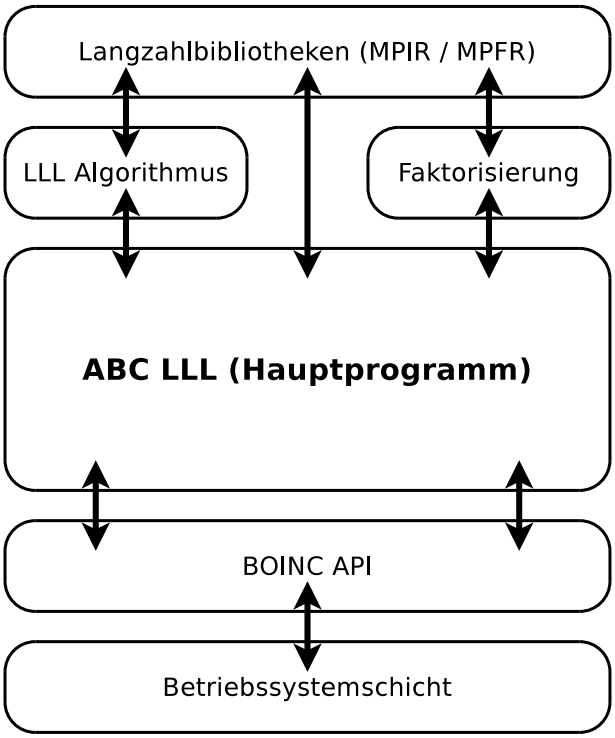

Abbildung 3.1: Architekturübersicht

Kapitel 4

Implementierung

In diesem Kapitel wird auf Details der Implementierung eingegangen. Die Abschnitte beziehen sich jeweils auf die einzelnen Komponenten, wie sie in Abbildung 2.4 dargestellt sind. Die Software ist vollständig in C/C++ entwickelt. Ein Hauptaugenmerk der Implementierung liegt auf der einfachen Portierung auf verschiedene Betriebssysteme und die Unabhängigkeit von proprietären Projekt- oder Dateiformaten.

4.1 Langzahlbibliotheken

Die Grundlage zur Arbeit mit Zahlen über der Maschinenwortgrenze bilden die Langzahlbibliotheken MPIR[1] für Ganzzahlen und MPFR[2] für Gleitkommazahlen. MPIR (Multiple Precision Integers and Rationals) ist eine Projektabspaltung von der GMP (GNU Multiprecision Library), welche sich darauf fokussiert, auf allen gängigen Betriebssystemen einfach gebaut und eingesetzt werden zu können. MPIR bietet Kompatibilitätsschnittstellen für Programme, die die GMP benötigt, somit macht es für Programmierer keinen Unterschied, welche Library eingesetzt wird. Die MPFR Bibliothek stellt keine C++ Header bereit. Zur Vereinfachung der Arbeit wird die freie Weiterentwicklung MPFR++[3] verwendet, welche die C++ Klasse `mpreal` einführt.

[1] http://www.mpir.org/
[2] http://www.mpfr.org/
[3] http://www.holoborodko.com/pavel/mpfr/

4.2 Faktorisierung großer Zahlen

Der ABC-LLL Algorithmus verlangt, dass den Elementen des Start-Tripel (welches in Primfaktorzerlegung vorliegt) jeweils ein Faktor hinzumultipliziert wird, damit aus dem Tripel ein ABC-Tripel wird. Um das Radikal eines Tripels (und damit die Qualität) zu bestimmen, müssen diese Faktoren ebenfalls in ihre Primfaktoren zerlegt werden. Erste Testreihen des Programms weisen darauf hin, dass in einem Suchbereich bis $max(a, b, c) = 2^{128}$ der ABC-LLL Algorithmus Faktoren in der Regel bis 2^{256} liefert, in Einzelfällen wurden aber auch Ausreißer bis zu 2^{81050} beobachtet. Dabei können diese Zahlen aus sehr wenigen, großen Primfaktoren bestehen und damit den Worst-Case für Faktorisierungsalgorithmen darstellen.

Die Größenordnung dieser Zahlen legt nahe, diese Arbeit an Systeme mit ausgereiften Algorithmen (General Number Field Sieve, Self-Initializing Quadratic Sieve, ECM, etc.) zu übergeben. Dabei hat man die Wahl zwischen dem Einsatz eines vollwertigen Computeralgebrasystems (Sage, Pari, Mathematica) oder einigen wenigen C-Libraries. Ersteres kommt für dieses Projekt nicht in Frage, da dadurch die Fähigkeit zur Modularität und Verteilbarkeit der Arbeit beeinträchtigt wird. Nach Betrachtung unter den Gesichtspunkten Benutzbarkeit, Leistungsfähigkeit und Produktreife fiel die Wahl auf das Open-Source Projekt *msieve*[1].

4.2.1 Ausbau von msieve

msieve ist eine Faktorisierungslibrary, welche die folgenden Algorithmen in einer *fire-and-forget* Funktion verwendet [Pap]:

- Trial Division
- Pollard-Rho (optional)
- Elliptic Curve Method (optional, beigesteuert durch gmp-ecm)
- General Number Field Sieve
- Multiple Polynomial Quadratic Sieve

Es bietet außerdem Möglichkeiten, mittels CUDA auf den Grafikkarten eines Rechners die Faktorisierung zu parallelisieren. Anhand der beigelieferten Demoapplikation lässt sich ablesen, wie die Funktion einzusetzen ist. Im Laufe des

[1] `http://sourceforge.net/projects/msieve/`

Einsatzes sind die nachfolgenden Probleme mit der Library aufgetreten. In Korrespondenz mit dem Entwickler konnten diese so weit möglich kompensiert werden. Einige hier entwickelten Erweiterungen flossen in den Entwicklungszweig von msieve wieder zurück.

1. Es existierte kein API-freundliches Errorhandling. Wenn Probleme während der Faktorisierung auftreten, bricht die Library mit *exit(-1)* ab. Für dieses Projekt ist das kein akzeptables Verhalten. Ein detailiertes Handling mit neu hinzugefügten Errorcodes wurde in die Algorithmen und Datenstrukturen eingepflegt, sodass auf die verschiedenen Fehler adäquat reagiert werden kann.

2. Der ausgiebige Einsatz dieser Library für Zahlen kleiner als 10^{20} hat einige Fehler in den Routinen des quadratischen Siebs erkennen lassen.

   - Das doppelte Freigeben von Speicher im Fehlerfall (*double-free*) verhinderte in Einzelfällen die Fehlerbehandlung.
   - Eine Primzahl, von der der Algorithmus fälschlich annimmt, dass sie die Eingabe teilt, führt zu einer nicht korrekt terminierenden Schleife.
   - Eine Division durch Null (*floating-point exception*) tritt unter bestimmten Bedingungen unter Windows und OS X auf.

Das Finden von Fehlern erwies sich als problematisch, da msieve mit probabilistischen Algorithmen arbeitet. Der Faktorisierungsfunktion werden zwei Zufallszahlen übergeben, die vor allem im Quadratischen Sieb verwendet werden. Um Bugs und andere Probleme zu finden oder überhaupt erst zu rekonstruieren, müssen die Zufallszahlen stets gleich sein. Es kommt in seltenen Fällen vor, dass aufgrund eines Fehlers im quadratischen Sieb die Faktorisierung fehlschlägt. Das erneute Würfeln von Zufallszahlen für den Algorithmus und ein erneuter Faktorisierungsversuch minimiert dieses Problem. Alle Änderungen wurden gegen msieve in der Version 1.51 vorgenommen. Ein Patchfile liegt dieser Arbeit bei und ist auch online erhältlich.

4.2.2 Einschränkung der Faktorisierungsaufgaben

Die Primfaktorzerlegung ist eine ressourcenintensive Aufgabe. Es lässt sich unmöglich vorhersagen, wie schwer die Faktorisierung bestimmter Eingaben ist,

da diese zum Zeitpunkt der Verteilung der Arbeitspakete nicht bekannt sind. Zahlen, die aus wenigen großen Primfaktoren bestehen, können den Faktorisierungsaufwand stark erhöhen. Unter dem Gesichtspunkt, dass die Applikation von möglichst vielen Spendern eingesetzt werden kann, ist eine Obergrenze für Faktorisierungsaufgaben eingesetzt worden. Eingaben, die die Obergrenze von 2^{128} überschreiten, werden abgebrochen. Die Eingabe wird in der Ergebnisdatei als fehlgeschlagen vermerkt. Diese Eingabe kann dann für stärkere Rechner vorgesehen werden.

4.2.3 C++ Wrapper

Für den Einsatz und Test der Faktorisierungsalgorithmen wurde ein vereinheitlichender C++11 Wrapper geschrieben. Dieser beinhaltet den Zufallszahlengenerator für die probabilistischen Algorithmen und wandelt die unterschiedlichen C-Listenimplementierungen zur Darstellung der Primfaktorzerlegung in den seit C++11 möglichen generischen Datentyp

vector<tuple<mpz_class, unsigned long int> >

um. *mpz_class* ist der C++ Wrapper für Ganzzahlen der GMP. std::tuple ist eine Standardklasse zur Erzeugung beliebiger n-Tupel, welche das mit C++11 neu eingeführte Konzept der variadic templates ausnutzt. Somit sind über das Projekt hinweg Primfaktorzerlegungen definiert als Liste von 2-Tupeln, bestehend aus einer Basis und einem Exponent.

4.3 ABC-LLL Algorithmus

Zur Vereinfachung der Arbeit mit Zahlentripeln wurden mehrere voneinander erbende C++ Klassen geschrieben.

- `bignumTriple`: Diese Klasse repräsentiert beliebige drei Langzahlen, welche in der in Abschnitt 4.2.3 vorgestellten Repräsentation in Primfaktordarstellung als auch in ausmultiplizierter Form speichert. Dabei ermöglicht diese Klasse durch verschiedene Konstruktoren die Eingabe eines Tripels entweder in Langzahlform oder in Primfaktoren zerlegt und führt bei Bedarf die jeweilige Methode zum Ausmultiplizieren oder Faktorisieren aus, um beide

Repräsentationen vorliegen zu haben. Weitere Hilfsmethoden lassen überprüfen, ob das gegebene Tripel die Anforderungen an ein abc-Tripel erfüllt.

- `abcTriple` (erbt von `bignumTriple`): In dieser Struktur werden abc-Tripel gespeichert. Ein `abcTriple` kann aus einem `bignumTriple` erzeugt werden, wenn es die Definition für abc-Tripel erfüllt. Zusätzlich bieten Methoden die Berechnung der Zahlengröße und der Qualität an.
- `coefficientTriple` (erbt von `bignumTriple`): Koeffizienten, die nach dem Algorithmus aus Abschnitt 3.2.1 aus einem Basis-Tripel ein abc-Tripel machen sollen, werden hier gespeichert. Einige Hilfsmethoden dienen dem Algorithmus *TripelAusKettenbruch* zur Entscheidung, welcher Koeffizient an welches Element des Basis-Tripels multipliziert werden muss.

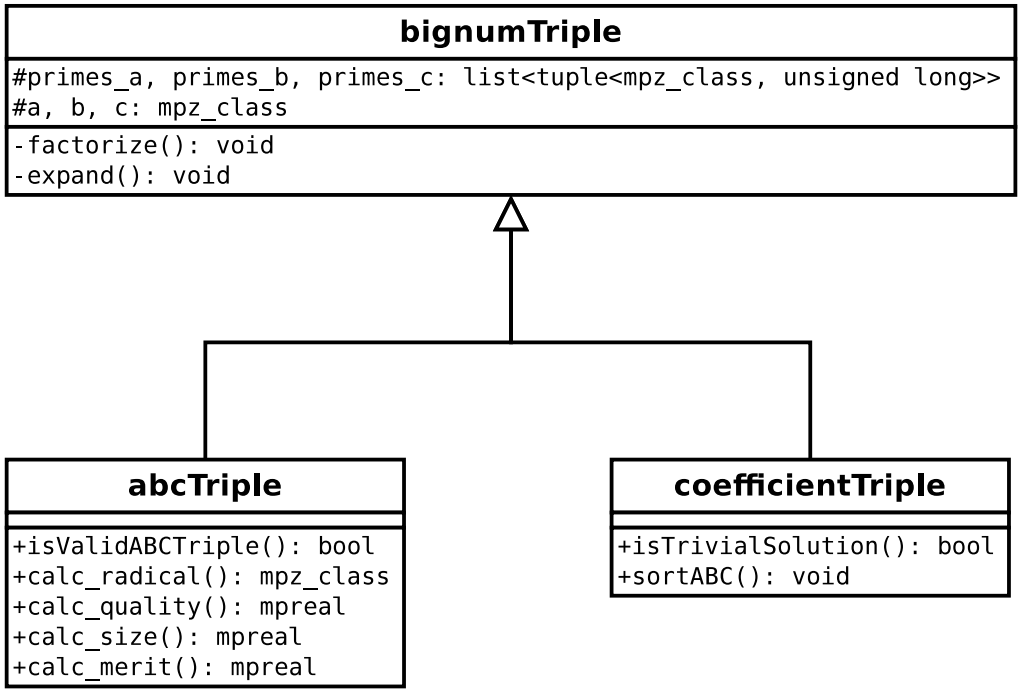

Abbildung 4.1: Struktur der Klassen zur Repräsentation von Tripeln

Der ABC LLL Algorithmus ist innerhalb der Klasse `tripleGenerator` implementiert. Die Methode `generateTriple` erwartet ein Basis-Tripel und eine Mindestqualität als Eingabe und gibt eine Liste gefundener abc-Tripel zurück. Verwendet wird sie von der Klasse `workunit`, welches eine deserialisierte Form von BOINC Arbeitspaketen und, falls vorhanden, einem Checkpoint darstellt.

Die Quelltexte umfassen ca. 3300 Zeilen C/C++ Code.

4.4 LLL-Reduktion von Gittern

Zur Gitterbasisreduktion sind die folgenden Bibliotheken betrachtet worden:

- *libfplll*: Dies ist die Referenzimplementation eines zuerst 2011 vorgestellten Algorithmus, der ein Gitter in quasilinearer Laufzeit LLL-reduziert. Diese Algorithmen sind seit 2011 auch im Computeralgebra SAGE enthalten.
- *NTL*: Die Number Theory Library bietet einen LLL-Algorithmus, welcher das letzte Mal 2009 aktualisiert worden ist.

Weitere Implementationen finden sich in den Computeralgebrasystemen *MAGMA* und *Pari*. Beide basieren auf *libfplll*, *MAGMA*s Implementierung wurde vom Entwickler der *libfplll* angepasst. *libfplll* erwies sich als sehr leicht in die Anwendung zu integrieren und bietet die Möglichkeit, die GMP Ganzzahlbibliothek für die interne Arithmetik anzuwenden.

4.5 Parallelisierung mit BOINC

Ein LAMP[1] System wird von der Hochschule RheinMain bereitgestellt, um ein BOINC Projekt darauf aufzusetzen. Zum Einsatz kommt der aktuelle BOINC *server-stable* build. Das Projekt ist unter `http://boinc2.cs.hs-rm.de` erreichbar und trägt den Namen **ABC Lattices @ Home** (kurz: ABCLLL).

4.5.1 Anwendungsentwicklung

Über BOINC werden Applikationen und sogenannte Tasks an die Spender verteilt. Die Applikationen kommunizieren via Shared Memory (bzw. Memory Mapped Files unter Windows) mit der Middleware, dafür muss sie gegen die BOINC Library gelinkt werden. Die Applikation wird vom BOINC Client gestartet, überwacht und gesteuert. Um das zu bewerkstelligen, ist die Einhaltung einiger Konventionen erforderlich.

[1] Abkürzung für **L**inux, **A**pache, **M**ySQL, **P**HP

- Dateizugriffe müssen über eine BOINC-Wrapper API geschehen. Dies ist vor allem eine Hilfestellung für Entwickler, da sie einige Besonderheiten unterschiedlicher Betriebssysteme intern abwickelt. Vor allem Windowsapplikationen profitieren davon, da Antivirenprogramme und andere Prozesse kurzfristig die Applikationsdateien für exklusiven Zugriff sperren.

- BOINC Applikationen laufen in der Regel mit geringer Priorität auf dem Zielsystem. Wenn der BOINC Client die Nutzung des Rechners durch den Benutzer bemerkt, werden zu dessen Komfort die BOINC Applikationsprozesse getötet. Um später das aktuelle Arbeitspaket nicht von vorne beginnen zu müssen, wurde das *Checkpointing* eingeführt. Ein Checkpoint ist eine Datei, in der eine Applikation für sich beschreiben kann, an welcher Stelle im Arbeitspaket sie sich befindet. Auf Existenz und Inhalt einer Checkpointdatei sollte die Applikation bei jedem Start prüfen, um gegebenenfalls von dort aus fortzusetzen. Über einen API-Aufruf lässt sich innerhalb der Applikation in Erfahrung bringen, ob es an der Zeit ist, einen Checkpoint anzulegen. Die gewünschten Checkpointintervalle werden vom Benutzer des BOINC Clients festgelegt. Außerdem gibt es API Zugriffe, mit denen dem Client ein kritischer Abschnitt angekündigt werden kann, beispielsweise für Dateizugriffe. Der BOINC Client vermeidet das Töten des Prozesses in kritischen Abschnitten.

- Das ordentliche Ende einer Applikation endet mit einem Aufruf von *boinc_finish(int exit_code)*. Alle anderen Wege, die zum Ende des Programms führen, werden als Berechnungsfehler interpretiert.

4.5.2 Serveranpassungen

Der BOINC Server ist fast ohne Modifikationen in seinen Standardvorgaben übernommen worden. Wie in Abbildung 2.4 auf Seite 11 zu sehen, müssen einige Komponenten vorgegeben oder angepasst werden.

- Der **Work Generator** ist in mehrere Komponenten unterteilt. Ein C++ Programm implementiert den Algorithmus wie in Kapitel 3.3 beschrieben. Die Ausgabe des Programms wird in eine Datei umgeleitet, in der jede Zeile ein zu untersuchendes Basis-Tripel darstellt. So bleibt es möglich, Arbeitspakete in verschiedenen Größenordnungen bereitzustellen, da die Anzahl der

Basis-Tripel beliebig gesplittet werden kann. Ein Bash-Skript teilt die Datei mittels *split* zu Blöcken à 3.000 Zeilen auf. Mittels eines BOINC Skripts wird aus diesen Teildateien und einer Beschreibung im XML-Format ein Arbeitspaket erzeugt und in das BOINC System eingepflegt.

- Der **Assimilator** ist ein Bash-Skript, welches alle neu eingegangenen Dateien konkateniert, ihre Ergebnisse sortiert und abspeichert.
  `$ cat wu_result_* | sort -u > sorted_results`

- Der **Validator** ist der Standard-Bitwise-Validator, den BOINC als Beispiel mitliefert. Dieser prüft, ob die Ergebnisse auf Byteebene vollständig übereinstimmen. Dies hat in frühen Versionen der Applikation zu Problemen geführt, da die von Windows-Applikationen erzeugten Ausgabedateien ein Byte pro Zeile länger gewesen sind. Eine neue Zeile wurde durch *std::endl* realisiert, welches unter Windows per Konvention zusätzlich ein Carriage-Return Zeichen vor dem Linefeed-Zeichen vorsieht.

- **Applikationen** müssen in eine Verzeichnisstruktur nach Applikationsname, Version, Architektur und ggf. Architekturbesonderheiten kopiert sowie signiert werden. Dazu wird bei Erstellung des Projekts ein Schlüsselpaar erzeugt.

Zusätzlich läuft auf dem Server ein Skript, welches regelmäßig das qualitativ hochwertigste Tripel aus den Ergebnissen sucht und auf der Projektwebseite veröffentlicht.

4.6 Architekturdaten

Die Minimalanforderung an ein Spendersystem besteht in einer CPU mit x86 Befehlssatz, SSE2 und MMX sowie 256 MB Arbeitsspeicher. Der geschätzte Rechenaufwand für einen Task beträgt $3*10^{13}$ FLOPs[1] und wird von einem durchschnittlichen Spender in 5 Stunden Rechenzeit erledigt. Um eine große Anzahl bereitwilliger Spender zu erreichen, wurde die für das Projekt geschriebene Applikation für Windows, OS X und Linux portiert. Speziell bei einigen 64-bit Windows-Varianten weniger Spender kommt es zu Speicherzugriffsfehlern, deren Ursache ungeklärt ist.

[1] Gleitkommaoperationen pro Sekunde

Das BOINC Projekt wird von ca. 120 Usern mit 200 Rechnern unterstützt. Im Durchschnitt werden momentan (Stand: 05. Mai 2013) 218 GigaFLOP/s Rechenzeit gespendet. Täglich melden sich zur Zeit 20 neue Benutzer an.

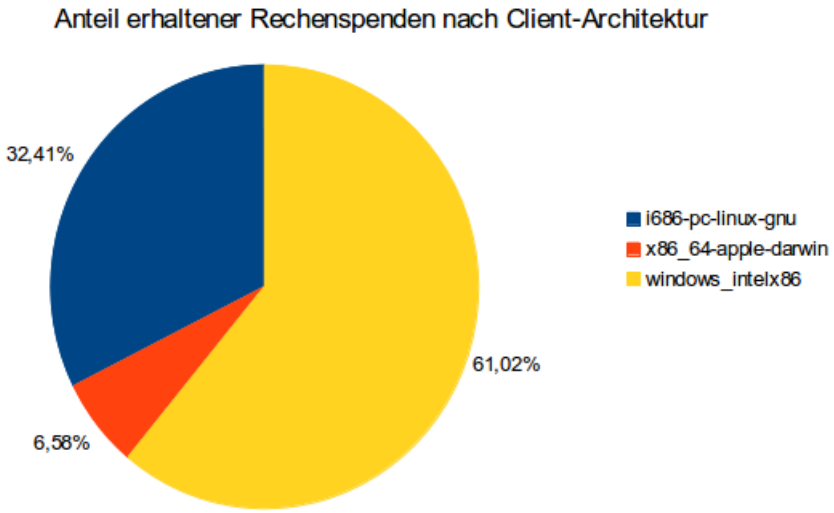

Abbildung 4.2: Nutzerstatistik, aufgeteilt nach benutzer Architektur

Zielarchitektur	Erfolgsrate	Berechnungsfehler	Abbruch durch Benutzer
i686-pc-linux-gnu	87.5%	1.4%	11.1%
x86_64-apple-darwin	75.8%	0%	24.2%
windows_intelx86	66.8%	7.7%	25.5%

Tabelle 4.1: Erfolgsstatistik der Applikation über die angebotenen Architekturen

Kapitel 5

Leistungsbewertung

In diesem Kapitel wird bewertet, wie leistungsfähig die Implementierung des ABC-LLL Algorithmus ist. Ein direkter Vergleich in „gute ABC-Tripel pro Zeit“mit vollständig anderen Algorithmen wurde nicht vorgenommen und wäre auch sehr ungenau, da die wenigen praktischen Algorithmen unterschiedliche Ziele (Abdeckung eines Suchraums gegenüber. gezielten Suchen von Tripeln bestimmter Qualität).

Außerdem wird eine Abschätzung vorgenommen, wo sich die praktischen Grenzen, nämlich Eingabegröße gegenüber Ressourcenbedarf und Laufzeit, befinden.

5.1 Implementationsvergleich

Als grobes Maß für die Qualität der implementierten Algorithmen gilt es, die Leistung von Dokchisters verteiltem Pari-Skript zu übertreffen. Dieser hat im Jahr 2003 mit 30 Rechnertagen 198 (davon 41 bislang unbekannte) gute ABC-Tripel finden können. Da der Begriff “Rechnertage“nicht weiter definiert wurde, dient es nur als Richtwert. In maximal 30 Tagen Arbeitszeit heutiger, handelsüblicher Rechner sollte mindestens dasselbe Ziel erreicht werden.

In einem ersten Versuch wurden ca. 16 Millionen Basis-Tripel erzeugt, welche die Primfaktoren bis zur Zahl 9 für Basis-Tripel bis zur Größe von 2^{64} durchprobiert. Die geschätzte Arbeit übersteigt dabei 720 Stunden. Außerdem wurde eine Schwäche in der Vergleichbarkeit der Implementierungen aufgedeckt. Der Workunit Generator des Projekts nimmt eine Tiefensuche vor, damit sind die Basis-Tripel innerhalb der generierten Eingabe nicht gleichverteilt. Wenn, wie in

diesem Versuch, die Suche nach einer bestimmten Zeit abgebrochen wird, ist die Suche nicht vergleichbar. Um etwas Breitensuche zu simulieren, wurden zusätzlich einige Tiefensuchen mit flachen Suchräumen vollständig durchgeführt. Insgesamt wurden 21.000 abc-Tripel gefunden, davon sind sechs mit einer Qualität $q > 1{,}4$. Der direkte Leistungsvergleich ist nur in Bezug auf die effiziente Wahl von Basis-Tripeln möglich. Der von Dokchister gewählte Tripelgenerator muss statt einer Tiefensuche eine Breitensuche implementiert haben, um schnell sehr unterschiedliche Basis-Tripel zu untersuchen.

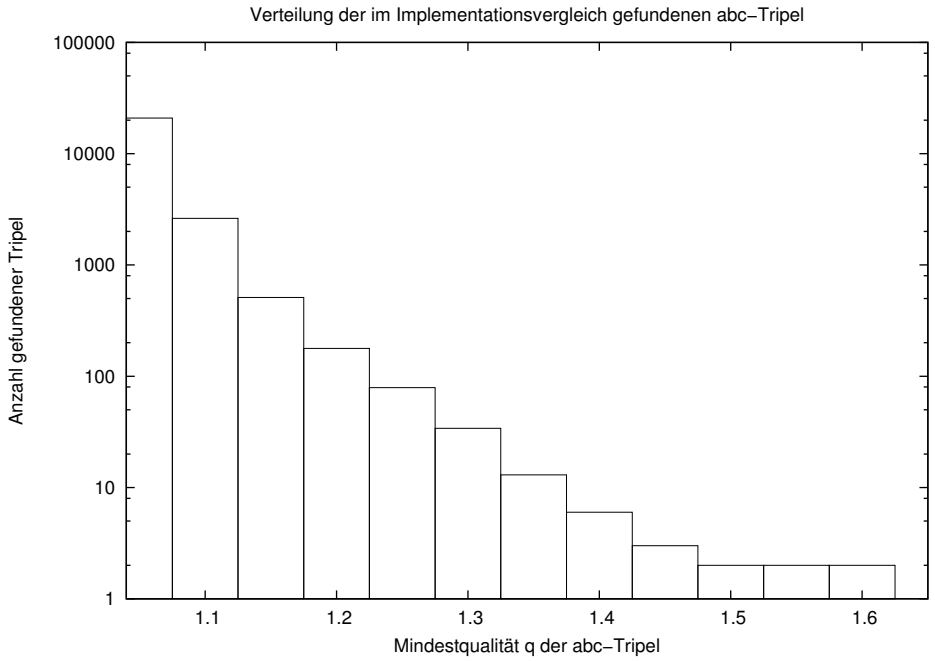

Abbildung 5.1: Anzahl der im Implementationsvergleich gefundenden abc-Tripel, aufgeschlüsselt nach Qualität

5.2 Laufzeitanalyse

Die Applikation wurde mittels *callgrind*[1] einer Laufzeitanalyse unterzogen. Dabei wurden zufällige Basis-Tripel als Eingabe gewählt. Der Aufrufgraph in Abbildung 5.2 zeigt die kostenintensivsten[2] Funktionen an. Die C++ Methoden, die mit

[1] Ein Werkzeug aus der valgrind tool-suite: `http://valgrind.org/info/tools.html`

[2] *callgrind* ermittelt aus verschiedenen Faktoren wie CPU Zyklen und Cachezugriffe einen Kostenindex für jeden Funktionsaufruf

abc-Tripeln arbeiten, rufen die Faktorisierungsfunktionen auf, um die Zahlen in Primfaktordarstellung zu speichern.

Auffallend ist, dass die LLL-Reduktion keine relevante Größe in Bezug auf die Rechenintensität darstellt. Die Faktorisierungsmethoden stellen 98 Prozent des Rechenzeitbedarfs.

Abbildung 5.2: Aufrufgraph der BOINC-Applikation mit den am meisten aufgerufenen Funktionen

5.3 Praktische Grenzen

Während der Implementierung der Applikation zeigten sich die Grenzen der gewählten Architektur. Wie in Abschnitt 4.2.2 beschrieben, können nicht alle vorkommenden Faktorisierungsaufgaben beim jeweiligen Spendersystem durchgeführt werden. Für den betrachteten Suchraum werden Basis-Tripel mit a, b und c in Größen zwischen 2^{105} und 2^{106} erzeugt. Je größer die Basis-Tripel sind, desto häufiger findet der Suchalgorithmus große Koeffizienten, die schwer zu faktorisieren sind. Ab einem Suchraum von ungefähr 2^{128} stellt dies nach Untersuchungen durch Stichproben ein Problem dar.

5.4 Neue ABC-Tripel

Das Rohergebnis der BOINC Eingaben beträgt aktuell 434.668 abc-Tripel. Abzüglich aller Mehrfacheinträge bleiben 28.000 bislang unentdeckte abc-Tripel mit $q >= 1.05$. Das beste, neu gefundene Tripel hat die Qualität 1,19. Abbildung 5.3 zeigt eine Aufstellung der neu gefundenen Tripel nach ihrer Qualität.

Qualität	A	B	C
1.15424	$5^{39} * 41^{1}$	$3^{63} * 683^{1} * 468175216162656106203231^{1}$	$2^{89} * 11^{4} * 83^{2} * 661^{1} * 15307^{2} * 3785210249^{1}$
1.1542	$5^{40} * 29^{3} * 95508072797^{1}$	$2^{99} * 19^{1} * 53^{2} * 173^{1} * 4643^{1} * 16453^{1}$	$3^{17} * 7^{28} * 11^{3} * 23^{1} * 257489839^{1}$
1.1542	$5^{46} * 53^{1} * 83^{1} * 1831^{1}$	$3^{39} * 11^{7} * 2617^{1} * 132409^{1} * 543113279^{1}$	$2^{27} * 7^{12} * 13^{11} * 79^{1} * 881^{1} * 8009^{2}$
1.15544	$3^{64} * 17^{3} * 79^{1} * 421^{2}$	$5^{42} * 13^{1} * 593^{1} * 579287^{1} * 25432339^{1}$	$2^{20} * 7^{30} * 1103^{1} * 990616734911^{1}$
1.15635	$3^{34} * 7^{16} * 1559^{1} * 2239^{2}$	$5^{40} * 11^{2} * 61^{2} * 5560364936593^{1}$	$2^{102} * 43^{1} * 54421^{1} * 1918907593^{1}$
1.15653	$2^{65} * 7^{7} * 37^{1} * 113^{1} * 42667^{1}$	$3^{49} * 11^{7} * 1549^{1} * 4783^{1} * 180868618933^{1}$	$5^{52} * 59^{1} * 47699415941^{1}$
1.15664	$5^{35} * 439^{2}$	$3^{25} * 7^{12} * 11^{4} * 109^{2} * 1582249851310317433^{1}$	$2^{109} * 31^{1} * 160426248195817^{1}$
1.15778	$5^{29} * 59^{2} * 4153^{1} * 23021^{1} * 139273^{1}$	$2^{106} * 29^{1} * 1237^{1} * 1823^{1} * 3929^{1}$	$3^{32} * 11^{15} * 83^{3} * 859^{1} * 5483^{1}$
1.15861	$3^{10} * 5^{35} * 11^{1} * 37^{1} * 12457^{1} * 284233^{1}$	$2^{30} * 13^{21} * 102931^{1} * 10192299841^{1}$	$7^{37} * 23^{2} * 67^{1} * 167^{2} * 181^{2} * 463^{1}$
1.15907	$2^{15} * 11^{7} * 19^{15} * 2207^{1}$	$3^{69} * 5^{10} * 887^{1} * 1381^{1} * 4201^{1}$	$7^{29} * 16111^{1} * 35521^{1} * 227554672330183337^{1}$
1.16133	$2^{51} * 11^{15} * 17^{6}$	$3^{42} * 5^{9} * 23^{1} * 2203^{1} * 27527^{1} * 2850401^{1} * 36487819^{1}$	$7^{39} * 41^{1} * 47^{2} * 499^{1} * 947^{1} * 796387^{1}$
1.16248	$7^{28} * 1074121^{1}$	$2^{105} * 17^{5} * 3181901^{1}$	$3^{34} * 5^{1} * 11^{4} * 13^{3} * 47^{1} * 1453749219019264199^{1}$
1.16289	$2^{31} * 11^{20} * 71^{1} * 431^{1} * 7013^{1}$	$3^{65} * 13^{1} * 124942189^{1}$	$5^{15} * 7^{19} * 743^{1} * 65933548999^{1}$
1.1657	$2^{39} * 7^{14} * 29^{2} * 1459^{1}$	$5^{43} * 1019^{1} * 43691^{1} * 18673167536629751^{1}$	$3^{26} * 11^{25} * 53^{1} * 6475133515469^{1}$
1.16775	$3^{52} * 11^{2} * 1280070949^{1}$	$7^{29} * 103307^{1} * 37006003231^{1}$	$2^{110} * 53^{1} * 178939^{1}$
1.16798	$5^{34} * 191^{1} * 1794913^{1} * 215545853^{1}$	$2^{46} * 7^{22} * 1249^{2} * 27479^{1}$	$3^{69} * 17^{1} * 4027^{1} * 207227^{1}$
1.16839	$11^{31} * 37^{2}$	$2^{49} * 5^{10} * 62311^{2} * 97930373255909^{1}$	$3^{51} * 7^{7} * 29^{1} * 406395054452251^{1}$
1.1698	$7^{32} * 73^{4} * 1127573^{1}$	$2^{3} * 11^{15} * 19^{12} * 89^{1} * 163^{1} * 431^{1} * 42967^{1} * 44851^{1}$	$3^{25} * 5^{28} * 61^{1} * 233^{1} * 12547^{1} * 12583^{2}$
1.17133	$5^{30} * 79^{1} * 509^{2} * 661^{1} * 3326507^{1}$	$3^{18} * 7^{27} * 29^{1} * 61^{1} * 401^{1} * 160474417^{1}$	$2^{110} * 11^{1} * 13^{3} * 17^{1} * 43^{2} * 2939^{1}$
1.17472	$7^{20} * 11^{4} * 259929645754279^{1}$	$2^{15} * 3^{54} * 13^{2} * 61^{1} * 66313487^{1}$	$5^{44} * 101^{3} * 222419^{1}$
1.17574	$5^{43} * 13^{1} * 317^{2} * 2123411^{1}$	$3^{45} * 7^{13} * 1246902877673^{1}$	$2^{29} * 11^{21} * 29^{2} * 53^{1} * 61^{1} * 33332363^{1}$
1.17672	$2^{72} * 13^{4} * 17^{1} * 23^{1} * 83^{2} * 69857^{1}$	$3^{41} * 7^{9} * 11^{12} * 41^{1} * 20536057^{1}$	$5^{43} * 29^{1} * 457^{1} * 13523^{1} * 19507^{1} * 978541^{1}$
1.17696	$2^{81} * 5^{6} * 7^{5} * 13^{4} * 61^{3}$	$11^{28} * 239672475606367^{1}$	$3^{57} * 448703^{1} * 54904681243^{1}$
1.18421	$2^{110} * 13^{1}$	$5^{10} * 11^{16} * 340859^{1} * 1130369041^{1}$	$3^{21} * 7^{14} * 1447^{2} * 72817^{1} * 159857^{1}$
1.19013	$3^{42} * 5^{11}$	$7^{43} * 978435539^{1} * 15968397404623^{1}$	$2^{15} * 11^{23} * 13^{3} * 17^{2} * 31^{3} * 242441^{1} * 15750167^{1} * 16099907^{1}$

Tabelle 5.1: Die 25 besten, neu gefundenen abc-Tripel, absteigend sortiert nach Qualität

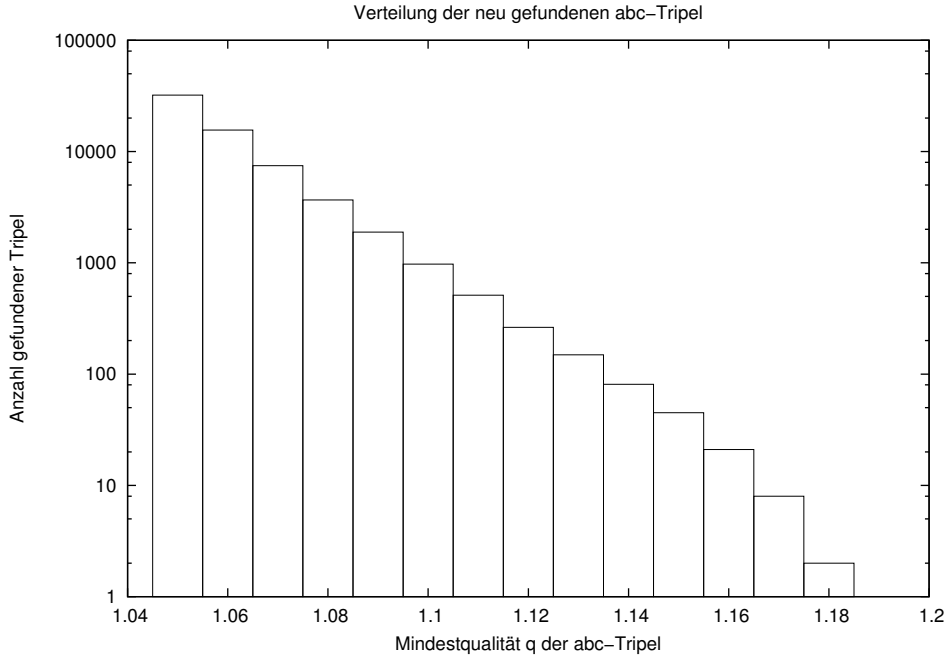

Abbildung 5.3: Anzahl der neu gefundenden abc-Tripel, aufgeschlüsselt nach Qualität

Kapitel 6

Schlussfolgerungen und Ausblick

Der LLL-ABC Algorithmus von Dokchister, ist neben dem ABC@Home Projekt ein sehr gut parallelisierbarer Algorithmus zur Suche nach abc-Tripeln. Das BOINC Projekt ABC Lattices @ Home mit Unterstützung der Hochschule RheinMain mindestens den Zahlenraum zwischen 2^{105} und 2^{106} weiter stichpunktartig nach neuen abc-Tripeln durchsuchen. Es sind ca. noch $\approx$ 1.000.000 Arbeitspakete zu betrachten und die um das Projekt gebildete Gruppe freiwilliger Helfer ist sehr motiviert und bereit, weiterzurechnen.

Wie auch ABC@Home und andere Projekte, die sich mit dem Thema beschäftigen, stößt LLL-ABC an seine Grenzen in großen Zahlenbereichen auf Grund des steigenden Faktorisierungsaufwands. Zukünftige Optimierungen des Algorithmus sollten sich auf die folgenden Dinge fokussieren:

- **Einsparen von Faktorisierungen**: In der aktuellen Implementierung könnten Faktorisierungsaufrufe eingespart werden. Zum Einen könnte der Code auf unnötige/doppelte Faktorisierungen untersucht werden, zum Anderen könnten versucht werden, ob die Anzahl der benötigten Basis-Tripel zur Betrachtung eines Zahlenbereichs verringert werden kann.

- **Verbesserung von msieve**: Die eingesetzte Faktorisierungslibrary hat Verbesserungspotenzial im Bereich der Fehlerbehandlung, sowie in der darunterliegenden Langzahlbibliothek. Bei dieser handelt es sich um eine nicht optimierte Eigenentwicklung des msieve-Autors. Eine Umstellung auf GMP/MPIR könnte eine Verbesserung darstellen. Auch der Einsatz von Grafikkarten zur Faktorisierung könnte helfen, die größeren Faktorisierungsaufgaben zu bewältigen.

- **Untersuchung der Faktorisierungsaufgaben**: Es könnte von Vorteil sein, die Natur der zu faktorisierenden Zahlen zu untersuchen. So könnten eventuell früh Faktorisierungsaufgaben abgebrochen werden, deren Ergebnis tendenziell zu schlechteren abc-Tripeln führt und andere beschleunigt werden.

Kapitel 7

Anhang

Dieser Arbeit liegt ein optischer Datenträger mit folgendem Inhalt bei:

- Dieses Dokument im PDF Format
- Die aktuelle Codebasis der Applikation sowie den Modifikationen an der Bibliothek msieve.
- Sämtliche referenzierte, frei verfügbare Literatur, zuzüglich Arbeitsmaterialien.
- Referenzierte Webseiten liegen unter dem entsprechenden Kürzel im Verzeichnis *Webseiten*
- Eine komprimierte CSV Datei mit allen bislang gefundenen abc-Tripeln. Außerdem liegt dieses Ergebnis aufgeschlüsselt in die folgenden Teilbereiche vor:
  - Ergebnissen aus dem laufenden Betrieb
  - Ergebnisse des Implementierungsvergleichs (siehe Kapitel 5.2).

Kapitel 8

Literaturverzeichnis

[ABC] ABC@home: *BOINC Projektwebseite*. – Verfügbar unter http://www.abcathome.com/, besucht: 22.11.2012.

[Ber] Berkeley, University of California: *BOINC Website*. – Erreichbar unter http://boinc.berkeley.edu, besucht: 03.05.2013.

[BOI] BOINC-Stats: *ABC@Home - Boincstats*. – Erreichbar unter http://boincstats.com/en/stats/41/project/detail, besucht: 03.05.2013.

[Dok04] Dokchitser, Tim: LLL & ABC. In: *Journal of Number Theory* 107 (2004), S. 161–167

[Flo] Flothow, Sebastian und Gampe, Jan und Löffel, Katrin und Wild, Robert: *Batman@Home*. – Projektauswertung erreichbar unter http://www.flothow.de/2011/2011-07-19.batman/, besucht: 06.05.2013.

[Hel10] Helfer, Etienne: *LLL lattice basis reduction algorithm*. 2010. – Studienarbeit. Verfügbar unter http://algo.epfl.ch/en/projects/bachelor_semester/bfm01, besucht: 24.02.2013.

[Hor10] Horst, Johannes P. d.: Finding ABC-triples using Elliptic Curves. (2010). – Masterthesis

[Mah10] Mahl, Matthias: *Konstruktion guter ABC-Tripel mit dem LLL-Algorithmus*. 1. Aufl. München : GRIN Verlag, 2010. – ISBN 978–3–640–68185–3

[Moc12] Mochizuki, Shinichi: *Inter-universal Teichmuller Theory IV: Log-volume Computations and Set-theoretic Foundations* http://www.kurims.kyoto-u.ac.jp/~motizuki/papers-english.html

[Nit] Nitaj, Abderrahmane: *THE ABC CONJECTURE HOME PAGE*. – Verfügbar unter `http://www.math.unicaen.fr/~nitaj/abc.html`, besucht: 22.11.2012

[Pap] Papadopoulos, Jason S.: *Msieve factoring library*. – Sourceforce Projekt verfügbar unter `http://sourceforge.net/projects/msieve/`, besucht: 22.11.2012.

[Sch10] Schnorr, Claus-Peter: Gitter und Kryptographie. (2010). – Skript zur Vorlesung