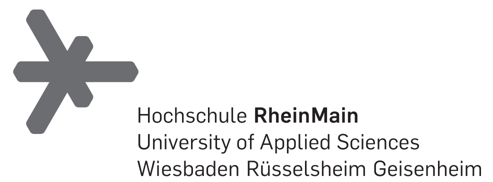

Fachbereich Design Informatik Medien
Studiengang Allgemeine Informatik

Master-Thesis

zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

Verteiltes Rechnen für Fragestellungen der Zahlentheorie

vorgelegt von Fabio Campos

am 3. September 2010

Referent: Prof. Dr. Steffen Reith

Korreferent: Prof. Dr. Jörn Steuding

Erklärung

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

Wiesbaden, 3. September 2010 Fabio Campos

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

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der FHW	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet	√	

Wiesbaden, 3. September 2010 Fabio Campos

Für Helaine, Lua, Malu und Viola.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Gliederung der Thesis 4

1.2. Notationen 5

2. Methoden und Techniken **7**

2.1. Globus-Toolkit 9

2.1.1. Architektur 10

2.2. BOINC 13

2.2.1. Überblick 14

2.2.2. Server-Architektur 16

2.2.3. Client-Architektur 19

2.3. Multicore Prozessoren 24

2.3.1. Cell Broadband Engine 25

2.4. Profiler 28

2.4.1. gprof - Der GNU Profiler 29

2.4.2. Shark 29

2.4.3. Valgrind 29

2.5. Mathematica 31

3. Zahlentheoretische Aspekte **33**

3.1. AKS-Algorithmus 33

3.2. AKS-Vermutung 37

3.3. Lenstras Heuristik 38

3.4. Popovychs Vermutung 39

4. Implementierung **41**

4.1. Grundlegende Überlegungen 41

4.1.1. Logische Formulierung der Vermutungen 42

4.1.2. Aufbau des Algorithmus 42

4.1.3. Ergebnismengen 43

4.2. Vorgehensweise 45

4.3. Algorithmus 46

4.3.1. Polynomarithmetik 47

4.3.2. x86-Architektur 54

4.3.3. Cell-BE-Architektur 55

4.4. BOINC 59

4.4.1. Das BOINC-Application Programming Interface 59

4.4.2. Definition der Arbeitspakete 61

4.4.3. Windows- und Linux-Applikation 63

4.4.4. Cell BE-Applikation 65

4.4.5. Projektspezifische Server-Prozesse 66

4.5. Algorithmus zu Lenstras Heuristik 69

5. Evaluation **73**

5.1. Worst Case 74

5.2. Maßnahmen zur Performanceoptimierung 76

5.3. BOINC 81

5.4. Performance und Vergleich 87

5.5. Sonstige Evaluationen 88

5.6. Umfrage 91

6. Fazit **95**

A. Literatur **97**

B. Inhalt der CD **101**

Abbildungsverzeichnis

2.1. Prozentualer Betriebssystemanteil der BOINC-Clients 9
2.2. Globus-Toolkit-Architektur 11
2.3. Weltweite erreichte Rechenleistung 14
2.4. BOINC-Architektur 15
2.5. BOINC-Kommunikation 16
2.6. Server-Status-Anzeige 19
2.7. Zeitlicher Workflow der Arbeitspakete 20
2.8. BOINC-Client-Architektur nach [ACA06] 21
2.9. BOINC-Manager-GUI 22
2.10. Kommunikation zwischen Core-Client und Applikation nach [ACA06] . . 22
2.11. Thread-Struktur des BOINC-Runtime-Systems nach [ACA06] 23
2.12. Höhere Performance bei geringem Verbrauch nach [Rei09] 25
2.13. Cell BE-Architektur aus [Sti07] 26
2.14. Shark-GUI 30

4.1. Vorgehensweise bei der Implementierung 45
4.2. Kürzeste Additionskette für $n < 101$ aus [Knu97] 49
4.3. Parallelisierungsmodelle nach [SG06] 57
4.4. Cell-BE Nachrichtenverlauf 58
4.5. www.primaboinca.com 60

5.1. Anzahl der passenden r-Werte bei zusammengesetzten Zahlen 75
5.2. Verteilung der r-Werte bei Primzahlen 76
5.3. Verteilung der r-Werte bei Primzahlen der Länge $128 - 1024$ Bits 77

5.4. Rechenzeit für zusammengesetzte Zahlen der Länge $128-1024$ Bits ohne Teiler < 100 . . . 78
5.5. Rechenzeit für Primzahlen der Länge $128-1024$ Bits . . . 79
5.6. Ausschnitt einer Shark-Analyse . . . 79
5.7. Vergleich für Zahlen der Länge 160 - 1024 Bits . . . 80
5.8. Speicherbedarf bei Überprüfung von Zahlen der Länge $128-1024$ Bits . 81
5.9. Anzahl der Arbeitspakete am Tag . . . 82
5.10. Anzahl der Client-Architekturen/OS bei primaboinca (Stand 06.08.2010) 82
5.11. Prozentuale weltweite Client-Verteilung (Stand 20.08.2010) . . . 83
5.12. Prozentuale weltweite Rechenleistungsverteilung (Stand 20.08.2010) . . . 83
5.13. Täglicher Teilnehmerzuwachs . . . 84
5.14. Anzahl der Teilnehmer . . . 85
5.15. Täglicher Rechnerzuwachs . . . 86
5.16. Anzahl der Rechner . . . 86
5.17. Täglicher Datentraffic des BOINC-Servers . . . 91
5.18. How many others BOINC projects do you support? . . . 92
5.19. How many of these projects are math based? . . . 93
5.20. How would you classify your mathematical skills? . . . 93
5.21. Do you know the AKS-Algorithm? . . . 94

Tabellenverzeichnis

4.1. Äquivalenz aussagenlogischer Formeln 43
4.2. Square and multiply-Methode 50
4.3. Interne Darstellung einer Zahl mittels MPM 58

5.1. Zeitgewinn durch Code-Anpassung 80
5.2. Intel und Cell BE-Vergleich 87
5.3. Vergleich der Laufzeiten zwischen AKS-Algorithmus und Vermutungen . 88

Verzeichnis der Quellcodes

2.1. XML-Konfigurationsdatei 17
2.2. Konfiguration der Dämonprozesse 17
2.3. Konfiguration von periodischen Tasks 18
2.4. Zufällige Primzahlerzeugung in Mathematica 31

4.1. Square and multiply-Methode für Polynome 50
4.2. Modulo-Rechnung für Polynome 51
4.3. Polynom-Definiton 54
4.4. Verwendung von *vector unsigned int* 58
4.5. Verwendung von *Branch Hints* 59
4.6. Kommunikation über Pipes 65
4.7. BOINC-Überprüfungen 66
4.8. BOINC-Main 67
4.9. BOINC-Make_Job 68
4.10. Lentras Heuristik 70
4.11. Lentras Heuristik - Variante 2 71

5.1. Anfangsversion (polyMult_Mod) 77
5.2. Aktuelle Version (polyMult_Mod) 78

Verzeichnis der Algorithmen

3.1. AKS-Algorithmus 36

4.1. Algorithmus zur Überprüfung der Vermutungen 47

4.2. Square and multiply-Methode 49

4.3. m-ary Methode 53

4.4. PPE-Algorithmus 56

4.5. SPE-Algorithmus 56

4.6. Windows und Linux BOINC-Algorithmus 64

4.7. Implementierung von Lenstras Satz 69

Kurzfassung

Diese Arbeit befasst sich mit zwei Hypothesen aus dem Bereich der Zahlentheorie. Hierbei handelt es sich jeweils um Vermutungen zur Identifizierung von Primzahlen. Die erste Vermutung (Agrawals Vermutung) war Ausgangspunkt zur Formulierung des ersten deterministischen Primzahltestalgorithmus mit polynomialer Laufzeit (AKS-Algorithmus). Hendrik Lenstras und Carl Pomerances Heuristik zu dieser Vermutung legt nahe, dass es unendlich viele Gegenbeispiele geben müsste. Jedoch ist bis dato kein Gegenbeispiel bekannt. Diese Hypothese wurde für $n < 10^{10}$ überprüft, ohne ein Gegenbeispiel zu finden. Die zweite Vermutung (Popovychs Vermutung) ergänzt Agrawals Vermutung um eine weitere Bedingung und stellt somit eine logische Verstärkung der Vermutung dar. Sollte eine dieser Hypothesen stimmen, würde die Laufzeit vom AKS-Algorithmus von $O((log\ N)^6)$ (aktuell schnellste Version des AKS-Algorithmus) zu $O((log\ N)^3)$ reduziert werden.

Auf Grundlage mathematischer Aussagen aus beiden Vermutungen wurde in dieser Arbeit ein Algorithmus zur Suche nach einem Gegenbeispiel formuliert. Um über mehr Rechenleistung zu verfügen und somit einen größeren Zahlenbereich überprüfen zu können, wurde ein Algorithmus zum verteilten Rechnen implementiert. Hierzu wurden zwei Architekturtypen untersucht: eine *Grid-Computing-* und eine *Volunteer-Computing-Architektur*. Bedingt durch die Voruntersuchungen erscheint der Grid-Computing Ansatz nicht attraktiv, da der Zugang zu den nötigen Ressourcen mit zu großem zeitlichen Aufwand verbunden ist. Aus diesem Grund wurde der Fokus auf den Volunteer-Computing-Ansatz gelegt. Als Implementierung dieser Architektur wurde das quelloffene *Berkeley Open Infrastructure for Network Computing* (kurz BOINC) verwendet. Hierbei sind eine Linux-, Windows- und Cell BE-Version entstanden, um das Rechnen auf unterschiedlichen Plattformen und Vergleiche zwischen diesen zu ermöglichen.

Die Rechenleistung, welche in der kurzen Zeit durch dieses BOINC-Projekts erreicht wurde, übertraf jegliche Erwartung. Mittlerweile werden weltweit von mehr als 600 Rechnern (mit steigender Tendenz) täglich etwa 150000000 Zahlen (≈ 431 GigaFLOPS) überprüft. Im Vergleich dazu überprüft ein Intel® Pentium® Dual Core E5200 bei einer Taktfrequenz von $2,5$ GHz etwa 320000 solcher Zahlen pro CPU-Kern am Tag. Trotz der durch das

BOINC-Projekt erreichten Rechenleistung würde die Überprüfung des anfangs erwünschten Zahlenbereichs $10^{10} < n < 10^{11}$ etwa 300 Tage benötigen.

Abstract

This thesis concerns itself with two hypotheses in number theory. Both are conjectures for the identification of prime numbers. The first conjecture (Agrawal's Conjecture) was the basis for the formulation of the first deterministic prime test algorithm in polynomial time (AKS algorithm). Hendrik Lenstras and Carl Pomerances heuristic for this conjecture suggests that there must be an infinite number of counterexamples. So far, however, no counterexamples are known. This hypothesis was tested for $n < 10^{10}$ without having found a counterexample. The second conjecture (Popovych's conjecture) adds a further condition to Agrawals conjecture and therefore logically strengthens the conjecture. If this hypothesis would be correct, the time of a deterministic prime test could be reduced from $O((log\ N)^6)$ (currently most efficient version of the AKS algorithm) to $O((log\ N)^3)$.

Using the mathematic statements in both conjectures, an algorithm to find counterexamples is formulated in this thesis. To be able to use more computing power and thus test a larger range of numbers, the algorithm was implemented for distributed computing. For this, two types of architecture were reviewed: grid-computing and volunteer-computing. After analysis of these architectures it was clear that grid computing would not be an attractive alternative, because access to the needed resources would require a large amount of time on the one hand, and would require less theorical work on the other hand. For these reasons the volunteer computing approach was chosen. The open-source Berkely Open Infrastructure for Network Computing (BOINC) toolkit was chosen to implement this architecture. Clients for Linux, Windows and Cell BE were implemented using this toolkit to enable calculations on multiple platforms.

The computing power achieved at short notice via the BOINC clients exceeded all expectations. At the moment, up to 600 globally distributed hosts (tendency increasing) test approximately 150000000 numbers daily. In comparison, a single Intel® Pentium® Dual Core E5200 clocked at 2.5 GHz can test 320000 numbers daily. Nonetheless, despite the computing power achieved by the BOINC clients, testing the originally planned range of $10^{10} < n < 10^{11}$ would require 300 days.

KAPITEL 1

EINLEITUNG

Diese Arbeit befasst sich mit zwei Hypothesen aus dem Bereich der Zahlentheorie und deren Implementierung. Entwickelt wurde ein Algorithmus zur Suche nach einem Gegenbeispiel für diese Vermutungen. Bei den Vermutungen handelt es sich jeweils um Algorithmen zur Identifizierung von Primzahlen. Primzahlen galten in der Mathematik schon immer als besondere Zahlen, Gauß nannte sie die "Königin der Mathematik". In Bezug auf Primzahlen gibt es zwei wichtige Bereiche: Primzahlerkennung und Primfaktorzerlegung. Effiziente Primzahltests sind für unser tägliches Leben von Bedeutung, da Primzahlen bei der Erzeugung kryptographischer Schlüssel verwendet werden. Zur Primzahlerkennung wurden in den letzten Jahrhunderten immer effizientere Algorithmen entwickelt.
Miller [Mil76] beschrieb 1975 basierend auf dem kleinen Satz von Fermat und in der Annahme der erweiterten Riemann-Hypothese einen deterministischen Primzahltest mit polynomialer Laufzeit. Rabin [Rab80] erweiterte 1980 diesen Test und erreichte somit einen zufallsbasierten Primzahltest mit polynomialer Laufzeit. Solovay und Strassen [SS77] beschrieben 1977 ebenso einen zufallsbasierten Primzahltest mit polynomialer Laufzeit. In der Annahme der erweiterten Riemann-Hypothese kann dieser Algorithmus ebenso deterministisch formuliert werden. Adleman, Pomerance und Rumely [APR83] erreichten 1983 mit einem deterministischen Algorithmus mit einer Laufzeit von $(log\ n)^{O(log\ log\ log\ n)}$ einen Durchbruch. Goldwasser und Kilian [GK86] beschrieben 1986 einen zufallsbasierten Algorithmus unter Verwendung von elliptischen Kurven mit polynomialer Laufzeit. Dieser Algorithmus gilt allerdings nicht für alle Zahlen. Adleman und Huang [AH92] veränderten 1992 den Algorithmus von Goldwasser und Kilian und erreichten dadurch einen zufallsbasierten Algorithmus mit polynomialer Laufzeit, welcher für alle Zahlen gilt. Das Hauptziel dieses Forschungsbereichs ist die Formulierung

eines deterministischen Primzahltestalgorithmus mit polynomialer Laufzeit. Der AKS-Algorithmus (bennant nach seinen Erfinder: M. Agrawal and N. Kayal and N. Saxena) [AKS02] beschreibt einen solchen Algorithmus ohne jegliche Annahme.

Der AKS-Algorithmus in einer der ersten Versionen hat eine Laufzeit von $O((log\ n)^{7,5})$.
In den Monaten nach der Entdeckung erschienen neue Varianten, die die AKS-Geschwindigkeit um Größenordnungen verbesserten. Wegen der großen Anzahl an Varianten sprechen Crandall und Papadopoulos in ihrem Aufsatz [CP03] von der Klasse der AKS-Algorithmen, statt vom AKS-Algorithmus. Eine veränderte Version des AKS-Algorithmus [Gra05] von Lenstra und Pomerance erreicht bereits eine Reduzierung der Laufzeit auf $O((log\ n)^6)$. Selbst die veränderte Version des AKS-Algorithmus spielt bedingt durch die relativ lange Laufzeit in der Praxis keine Rolle. Sollte eine der in dieser Arbeit überprüften Vermutungen stimmen, wäre die Laufzeit eines deterministischen Algorithmus auf $O((log\ n)^3)$ reduziert.
Die erste Vermutung, die AKS-Vermutung, wurde bereits vor der Veröffentlichung des AKS-Algorithmus aufgestellt. In dieser von Agrawal betreuten Arbeit [BP01b] wurde bereits der Zahlenbereich $n < 10^{10}$ ohne Gegenbeispiele überprüft. Hendrik Lenstras und Carl Pomerances Heuristik [LP02] zu dieser Vermutung legt nahe, dass es unendlich viele Gegenbeispiele geben müsste. Jedoch ist bis dato kein Gegenbeispiel bekannt. Basierend auf diese Heuristik können Zahlen, welche vorgegebene Eigenschaften erfüllen, anhand einer Vorberechnung gezielt überprüft werden. Popovych veröffentlichte die zweite Vermutung [Pop09] erst sieben Jahre später. Popovychs-Vermutung basiert auf der AKS-Vermutung und erweitert diese um eine weitere Bedingung. Die Laufzeit wird durch diese Erweiterung unwesentlich verändert.

Der in dieser Arbeit implementierte Algorithmus zur Suche nach einem Gegenbeispiel arbeitet nach der Exhaustionsmethode (engl. Brute-Force-Method). In dieser Methode werden alle (in diesem Fall ungeraden) Zahlen überprüft. Um mehr Rechenleistung zu erreichen und somit einen größeren Zahlenbereich überprüfen zu können, wurde ein Algorithmus zum verteilten Rechnen entwickelt. Als Middleware für die verteilte Anwendung wurde zum einem Berkeley Open Infrastructure for Network Computing (kurz BOINC[1])

[1] http://boinc.berkeley.edu/

und zum anderen Globus-Toolkit[1] eingesetzt. Hierbei sind BOINC-Client-Versionen der Anwendung für Windows-, Linux- und Cell BE-Systeme entstanden.

Folgende Fragen wurden zu Beginn dieser Arbeit formuliert:

F1 Wie hoch ist der Zeitaufwand, ein komplettes BOINC-Projekt (inkl. Server) aufzusetzen?

F2 Welche Zeit benötigt das Projekt, um einen bestimmten Zahlenbereich zu überprüfen?

F3 Wo und mit welchem Aufwand sind Zeitslots auf Grids, welche Globus-Toolkit verwenden, zu bekommen?

F4 Welcher Architekturtyp eignet sich am besten für ein solches Projekt?

F5 In wie weit unterscheiden sich die Implementierungen für unterschiedliche Betriebssysteme?

F6 Wie hoch ist der Zeitaufwand pro Version?

F7 Welche Dimensionen (Benutzer, Pakete, Traffic, usw.) nimmt ein BOINC-Projekt in der zur Verfügung stehenden Zeit an?

F8 Welche Voraussetzungen muss ein Server für ein BOINC-Projekt erfüllen?

F9 Ist die Verwendung der Cell BE-Prozessoren unter BOINC möglich?

F10 Welche obere Schranke hat r in dem überprüften Zahlenbereich, wenn gilt $n \in \mathbb{P}$?

Außerdem wurden folgende Hypothesen zu Anfang der Arbeit aufgestellt:

H1 Der Zahlenbreich zwischen $10^{10} < n < 10^{11}$ für $r < 100$ ist in der relativ kurzen Zeit berechenbar.

H2 Ohne Verwendung von Lenstras Heuristik wird kein Gegenbeispiel gefunden.

H3 Die Anzahl der Nutzer, welche das BOINC-Projekt unterstützen, wird > 20 sein.

[1] http://www.globus.org/toolkit/

1.1. Gliederung der Thesis

Diese Thesis unterteilt sich in einen theoretischen und einen praktischen Teil. Der theoretische Teil dieses Dokuments beschreibt zunächst im Kapitel 2 die verwendeten Methoden und Techniken. Im Anschluss daran werden im Kapitel 3 mathematische Aspekte und Grundlagen der Arbeit ausführlich beschrieben.
Der praktische Teil der Thesis beginnt mit Kapitel 4. Dieses Kapitel beschäftigt sich zunächst mit Überlegungen, welche auf mathematischen Aspekten basieren. Anschließend werden Details der Implementierung polynomarithmetischer Methoden und mögliche Maßnahmen zur Beschleunigung der Berechnung besprochen. Nach der Beschreibung der Implementierung des Algorithmus zum verteilten Rechnen werden abschließend zwei Implementierungsvarianten von Lentras und Pomerances Heuristik erläutert. Kapitel 5 präsentiert die Ergebnisse der Berechnungen sowie statistische Aussagen über die in der Thesis gewonnenen Daten. Abschließend wird im Kapitel 6 ein Fazit formuliert.

1.2. Notationen

Um Missverständnisse zu vermeiden und die Thesis nicht mit Definitionen und / oder Querverweisen auf solche zu überfüllen, soll diese Liste der verwendeten mathematischen Symbole und ihre Definitionen dienen.

Notation	Definition
$log\ x$	Logarithmus zur Basis 2: $log_2\ x$
$\mathbb{N}$	Die Menge der natürlichen Zahlen: $\{1, 2, 3, ...\}$
$\mathbb{N}_0$	Die Menge der natürlichen Zahlen mit Null: $\mathbb{N}_0 = \mathbb{N} \cup \{0\} = \{0, 1, 2, 3, ...\}$
$\mathbb{Z}$	Die Menge der ganzen Zahlen: $\{..., -2, -1, 0, 1, 2, ...\}$
$\mathbb{P}$	Die Menge der Primzahlen: $\{2, 3, 5, 7, 11, 13, ...\}$
$O(f(n))$	Die Menge aller Funktionen $g(n)$, für die es zwei Konstanten n_0 und c gibt, für alle $n \geqslant n_0$ gilt: $g(n) \leqslant c * f(n)$
$\#\mathbb{A}$	Die Anzahl der Elemente in der Menge $\mathbb{A}$: Sei $\mathbb{A} = \{1, 2, 5\}$, dann $\#\mathbb{A}$ = 3
$\lfloor x \rfloor$	Abrunden: $max\{n \vert n \leqslant x, n \in \mathbb{N}\}$
$m \mid n$	Die Zahl m teilt die Zahl n
$m \nmid n$	Die Zahl m teilt die Zahl n nicht
$ggT(x, y)$	größter gemeinsamer Teiler von x und y
$x \bmod y$	Rest bei Division: $x - y\lfloor x/y \rfloor$
$\mathbb{Z}_n$	Ring der ganzen Zahlen modulo n
$O_r(a)$	Die Ordnung von a modulo r: der kleinste Wert für k, für den gilt $a^k = 1 (\bmod\ r)$
$\phi(r)$	Eulersche ϕ-Funktion: Anzahl der Zahlen n, für die gilt $n < r$ und $ggT(n, r) = 1$
$x \equiv y \bmod n$	x ist kongruent zu y modulo n: falls $n \mid (x - y)$

$x \not\equiv y \mod n$	x ist nicht kongruent zu y modulo n
P	Die Klasse der deterministich in polynomieller Zeit lösbaren Probleme
NP	Die Klasse der nichtdeterministisch in polynomieller Zeit lösbaren Probleme

KAPITEL 2

METHODEN UND TECHNIKEN

In dieser Arbeit wurden *Grid-Computing* und *Volunteer computing* als Architekturtypen zum verteilten Rechnen untersucht. Die Hauptunterschiede beider Architekturtypen bestehen nach Anderson [And02] in den folgenden Punkten:

Volunteer Computing muss Amateure ansprechen. Ressourcen werden nicht von Fachleute freigegeben sondern von Anwendern mit geringen Fach- und Computerkenntnissen. Deshalb muss die Software einfach zu installieren sein und darf weder ein spezielles Betriebssystem benötigen, noch komplexe Konfigurationen verlangen.

Volunteer Computing muss Hacker tolerieren. Teilnehmer am Volunteer Computing bleiben anonym, somit besteht die Möglichkeit falsche Ergebnisse zurückzuliefern oder Credits für nicht berechnete Arbeitspakete zu beanspruchen. Eine Middleware für Volunteer Computing muss diese Möglichkeit berücksichtigen, indem Pakete redundant gerechnet werden.

Volunteer Computing verlangt kein sicheres Netz. Freiwillig zur Verfügung gestellte Ressourcen befinden sich meistens hinter einer Netzwerk-Firewall, die keine eingehenden Netzwerkverbindungen erlaubt. Dies erfordert ein Modell, bei dem teilnehmende Clients in regelmäßigen Abständen Aufgaben von einem zentralen Server anfordern. Grid Software dagegen verwendet ein Modell, bei dem der zentrale Server die zu bearbeitenden Daten an die Clients verschickt.

Volunteer Computing funktioniert asymmetrisch. Grid Computing ist in der Regel symmetrisch aufgebaut. Eine Organisation kann an einem Tag externe Ressourcen nutzen und am nächsten Tag selbst welche zur Verfügung stellen. Volunteer

Computing dagegen funktioniert asymmetrisch: freiwillige Teilnehmer stellen Ressourcen für Projekte zur Verfügung.

Volunteer Computing verlangt hervorragende Öffentlichkeitsarbeit. Volunteer Computing ist auf freiwillige Teilnehmer angewiesen. Ein Wissenschaftler erreicht eine höhere Rechenleistung nicht durch Anfragen oder Ankaufen von Rechenzeiten, sondern indem er die Öffentlichkeit davon überzeugt, dass seine Forschung den Einsatz wert ist.

Eine Unterscheidung der Architekturtypen ist anhand der Aussagen von Anderson somit eindeutig. Grid-Computing und Volunteer-Computing besitzen unterschiedliche Fähigkeiten, verwenden unterschiedliche Methoden und verlangen unterschiedliche Computeraffinität der Teilnehmer.
Als Implementierung einer Grid-Computing-Architektur standen folgende Middleware-Systeme zur Auswahl:

- Globus-Toolkit
- Unicore[1]
- gLite[2]

Das Globus-Toolkit, welches von der *Globus Alliance*[3] entwickelt wird, legt allgemeine Standards für eine Grid-Architektur fest. Aus diesem Grund wurde Globus Toolkit in dieser Thesis verwendet.
Aufgrund der weltweiten Verbreitung und starken Beteiligung wurde BOINC als Implementierung einer Volunteer-Computing-Architektur eingesetzt. Als Betriebssysteme für die Client-Plattformen wurden Windows und Linux vorgesehen. Wie Abbildung 2.1 zeigt, werden damit 95% der BOINC-Clients erreicht.

Weiterhin wurde Cell-BE als Zielplattform für die Berechnung ausgewählt. Diese Architektur erwies sich in [CP08] und [CS09] als besonders leistungsfähig für mathematische Berechnungen.

[1] http://www.unicore.eu/
[2] http://glite.web.cern.ch/glite/
[3] http://www.globus.org/

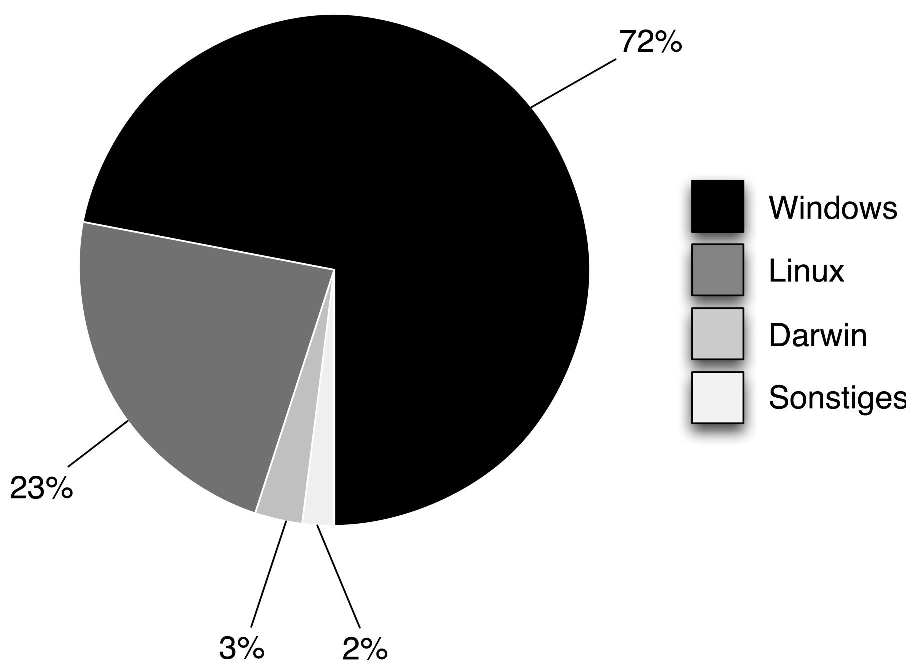

Abbildung 2.1.: Prozentualer Betriebssystemanteil der BOINC-Clients

Im Unterkapitel 2.1 werden Details des quelloffenen Softwarepakets namens Globus Toolkit besprochen. Unterkapitel 2.2 beschäftigt sich mit der Entstehung und Architektur des Berkeley Open Infrastructure for Network Computing (kurz BOINC[1]). Die Einzelheiten des Cell Broadband Engine (kurz Cell-BE[2]) werden im Unterkapitel 2.3 besprochen. Während sich Unterkapitel 2.4.2 mit dem Profiler Shark[3] beschäftigt, wird im Unterkapitel 2.4.3 die Werkzeugsammlung Valgrind[4] näher erläutert.

2.1. Globus-Toolkit

Das Globus-Toolkit[5] ist ein quelloffenes weiterverbreitetes Grid-Middleware. Es besteht aus einer Sammlung von Werkzeugen:

- zum Aufbau eines Computer-Grids
- zur Einrichtung und Verwaltung von Sicherheitsmechanismen

[1] http://boinc.berkeley.edu/
[2] https://www-01.ibm.com/
[3] http://developer.apple.com/tools/sharkoptimize.html
[4] http://valgrind.org/
[5] http://www-unix.globus.org/toolkit/

- zur Einrichtung und Verwaltung von Ressourcen
- zur Kommunikationsverwaltung

Das Globus-Toolkit wird von der *Globus Alliance*[1] entwickelt, es ist konform zur *Open Grid Services Architecture* (kurz OGSA[2]). Ursprung dieses Projekts war die *Supercomputing 95 conference Projekt IWAY*. Hauptziele dieses Projekts sind:

Gemeinsame Nutzung von Ressourcen Die globale Nutzung von Ressourcen ist der Grundstein des Grid-Computings.

Sicherer Zugang Vertrauen zwischen den Ressourcenanbietern und Grid-Nutzern ist in einem Grid essenziell. Gemeinsame Verwendung von Ressourcen sollte in einem Grid die lokalen Sicherheitsrichtlinien der einzelnen Teilnehmergruppen berücksichtigen.

Effiziente Nutzung Effiziente und ausgeglichene Nutzung der freigegebenen Ressourcen sollte in einem Grid gewährleistet sein.

Entfernungsunabhängigkeit Entfernung sollte in einem Grid keine Rolle spielen. Freigegebene Ressourcen sollten überall im Grid erreichbar sein.

Offene Standards Interoperabilität zwischen unterschiedlicher Grids ist ein weiteres Ziel dieses Projekts. Offene Standards (wie OGSA[3]) ermöglichen eine konstruktive Zusammenarbeit bei der Entwicklung.

2.1.1. Architektur

Das Konzept eines Computer-Grids erlaubt eine abstrakte, logische Sicht auf physische Ressourcen und umfasst sowohl Server, Datenspeicher, Netzwerke als auch Software. Die Architektur eines solchen Grids wird oft anhand von *Schichten* (engl. *layers*) beschrieben. Jede Schicht hat eine spezifische Funktion. Die oberen Schichten einer solchen Systemarchitektur sind im allgemein eher Benutzer fokussiert, während die unteren Schichten den

[1] http://www.globus.org/
[2] http://www.globus.org/ogsa/
[3] Open Grid Services Architecture

Fokus auf die Hardware setzten.

Die unterste Schicht ist das Netzwerk, welches die Grid-Teilnehmer verbindet. Darüber befindet sich die Ressourcen-Schicht (Computer, Storage-Systeme, usw.). In der Middleware-Schicht werden Werkzeuge zur Grid-Teilnahme zur Verfügung gestellt. Die oberste Applikationsschicht stellt Dienste zur Grid-Interaktion zur Verfügung und beschreibt somit die Benutzerschnittstelle.

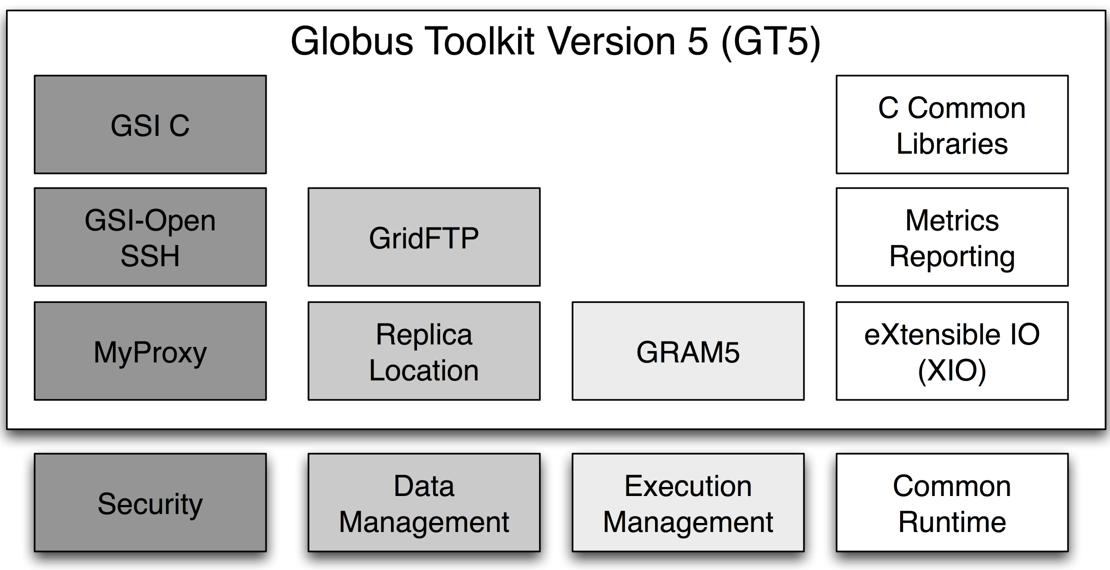

Abbildung 2.2.: Globus-Toolkit-Architektur

Die wesentlichen Komponenten des Globus-Toolkits sind folgende:

GSI C Die *Globus Toolkit Pre-Web Services Authentication and Authorization*-Komponente stellt Interfaces und Werkzeuge für Authentifizierung und Verwaltung von Berechtigungen und Zertifikaten zur Verfügung.

GSI-OpenSSH ist eine modifizierte Version von OpenSSH, welche zusätzliche Unterstützung für X.509[1] bietet. Dieses Modul ermöglicht eine *Single Sign-On*-Anmeldung.

MyProxy ist eine quelloffene Software für die Verwaltung von sicherheitsrelevanten Daten (Zertifikate und Schlüssel). Diese Komponente kombiniert online zugängli-

[1] ist ein Standard für eine Public-Key-Infrastruktur

chen Speicher für diese Daten mit einer Online-Zertifizierungsstelle, so dass Grid-Teilnehmer von überall sicher auf Ihre Zertifikate und Schlüssel zugreifen können, wenn sie diese benötigen.

GridFTP ist ein hochperformantes, sicheres und zuverlässiges Datenübertragungsprotokoll, das für weit reichende Netzwerke mit hohen Bandbreiten optimiert wurde. GridFTP basiert auf dem FTP-Protokoll.

GRAM5 Die *Grid Ressource Allocation and Management*-Komponente wird verwendet, um Jobs auf Grid-Computing-Ressourcen ausfindig zu machen, zu übermitteln, überwachen und zu beenden. GRAM5 ist kein Scheduler, sondern eine Sammlung von Kommunikationsdiensten.

C Common Libraries Diese Bibliotheken stellen eine Abstraktionsschicht für Datentypen und Datenstrukturen zur Verfügung, welche von Globus Toolkit genutzt werden.

eXtensible IO Globus XIO ist eine in C geschriebene erweiterbare Bibliothek für Input/Output.

Wider Erwarten verwendet weder der *Hessische Hochleistungsrechner* (kurz HHLR[1]) noch das *Center for Scientific Computing* (kurz CSC[2]) das Globus Toolkit. Das HHLR setzt eine proprietäre Middleware namens *IBM High Performance Computing Toolkit V5.1.1.0* ein. Das CSC ist ebenso im HHLR als Ressource verwendbar. Bedingt durch die nicht vorhandene Infrastruktur wurde die Arbeit mit dem Globus Toolkit nach der Installation und Konfiguration zwei virtueller Maschinen nicht weitergeführt. Der Fokus wurde auf die nach den Voruntersuchungen Erfolg versprechendere Volunteer-Computing-Architektur gelegt.

[1] http://www.hhlr.tu-darmstadt.de/organisatorisches/startseite/index.de.jsp
[2] https://csc.uni-frankfurt.de/web/index.php

2.2. BOINC

Zwei fundamentale Entwicklungen für verteiltes Rechnen haben in den 90er Jahren stattgefunden:

- die Verdoppelung der Komplexität integrierter Schaltkreise mit minimalen Komponentenkosten etwa alle zwei Jahre [Moo65];
- das rasante Wachstum der Anzahl der mit dem Internet verbundener Rechner.

Wissenschaftliche Projekte machten sich das aus diesen Rechnern entstandene Netz zu Nutze. Das Projekt *Great Internet Mersenne Prime Search* initiiert von G. Woltmann (kurz GIMPS[1]), welches nach großen Primzahlen sucht, wurde ebenso wie das Projekt *Distributed.net*[2] im Jahr 1997 gestartet. Das Distributed.net-Projekt beschäftigt sich mit der Entschlüsselung verschlüsselter Nachrichten. Ein weiteres Projekt namens *Search for extraterrestrial intelligence at home* (kurz SETI@home[3]) [ACK+02], welches im Jahr 1999 gestartet wurde, hat als Ziel, bestimmte vielversprechende Abschnitte des Himmels gezielt nach Radiosignalen von Außerirdischen abzusuchen.
BOINC [And04], ein quelloffenes Software-Framework für verteiltes Rechnen, wurde basierend auf Erfahrungen aus dem SETI@home-Projekt entwickelt. Dieses Framework wird seit 2003 an der Universität Berkeley[4] entwickelt. Seit dem 18. November 2003 steht BOINC unter der GNU General Public License[5]. Das Hauptziel des BOINC-Projekts ist, ungenutzte Rechenleistung über das Internet oder Intranet nutzbar zu machen. Anwender können durch BOINC an Projekten, welche meist einen wissenschaftlichen oder gemeinnützigen Hintergrund haben, teilnehmen. Diese Teilnahme erfolgt über die Installation eines Client-Programms und die Auswahl eines oder mehrerer zu unterstützender Projekte durch den Anwender. BOINC wird inzwischen von vielen Projekten benutzt, neben SETI@home seien an dieser Stelle noch Einstein@home [Col10] und Climaprediction.net [Cli10] genannt. Aktuell unterstützen etwa 299.182 Anwender mit 532.517 Rechnern BOINC-Projekte. Dadurch wird täglich eine Rechenkapazität von ∅ 2.493,81 TeraFLOPS

[1] http://www.mersenne.org/
[2] http://www.distributed.net/
[3] http://setiathome.ssl.berkeley.edu/
[4] http://berkeley.edu/
[5] http://www.gnu.de/documents/gpl.de.html

erreicht.

Im folgende Unterkapitel werden die Einzelheiten der BOINC-Architektur besprochen. Während im Unterkapitel 2.2.2 die Server-Architektur im Detail erläutert wird, beschäftigt sich Unterkapitel 2.2.3 mit Einzelheiten der Client-Architektur. Die weltweite Verteilung der durch BOINC-Projekte erreichten Rechenleistung wird in Abbildung 2.3 dargestellt.

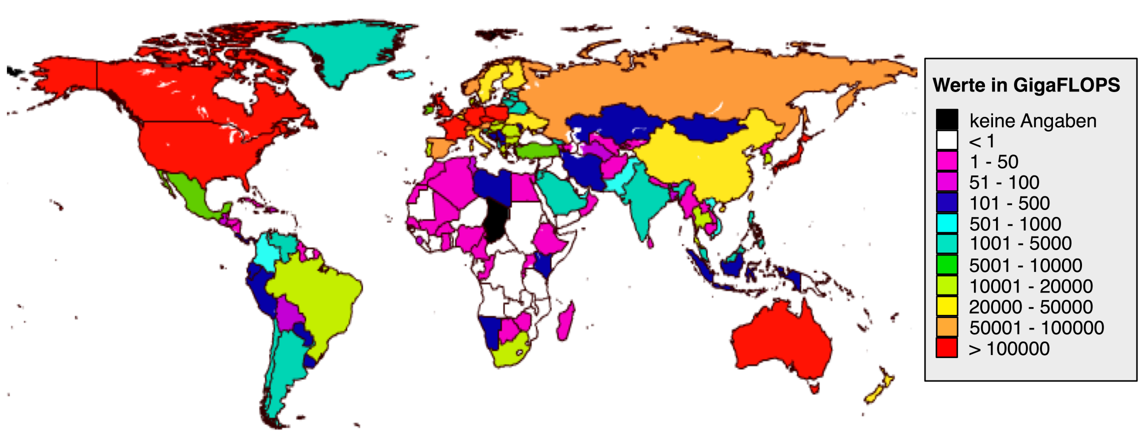

Abbildung 2.3.: Weltweite erreichte Rechenleistung

2.2.1. Überblick

Das BOINC-Framework besteht aus Client- und Server-Komponenten. Sowohl die Client- als auch die Server-Software besteht einerseits aus einigen Komponenten, die für jedes Projekt identisch sind und andererseits aus Teilen, die projektspezifisch angepasst oder implementiert werden müssen. Die einzelnen Komponenten und deren Beziehungen sind in Abbildung 2.4 dargestellt.

Die Kommunikation zwischen Client und Server (siehe Abbildung 2.5) verläuft in mehreren Phasen. Zunächst kontaktiert der Core Client den Scheduling-Server, bekommt von

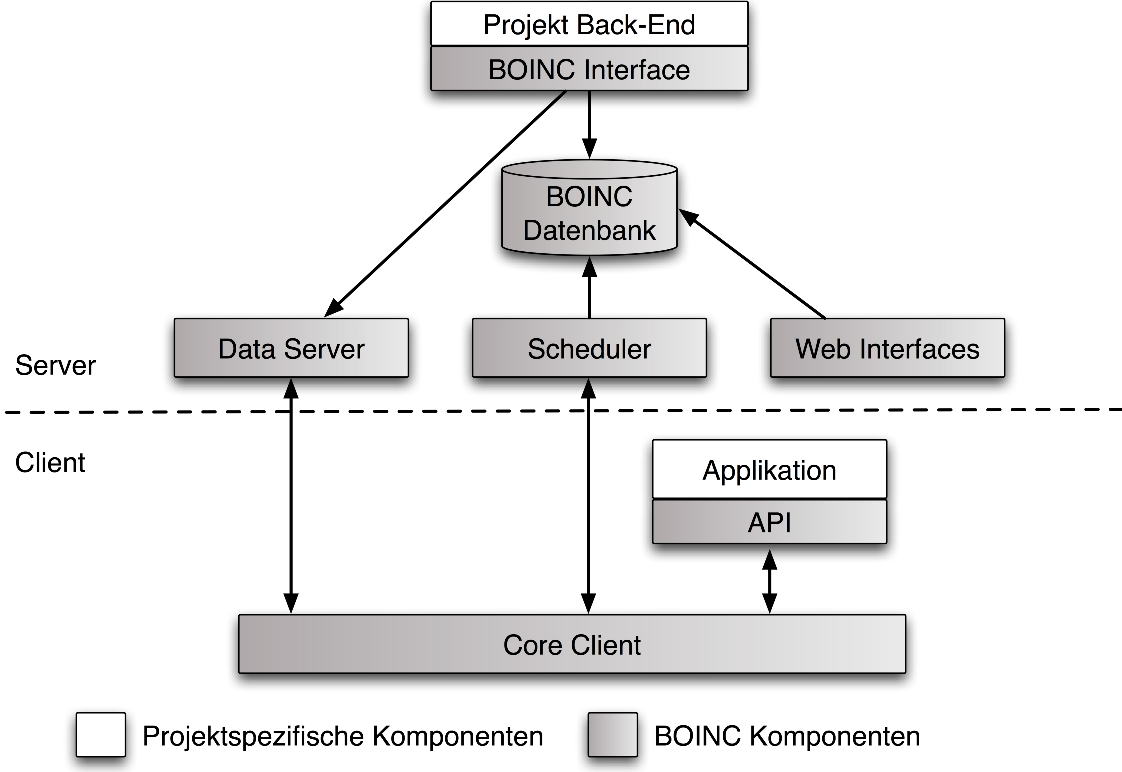

Abbildung 2.4.: BOINC-Architektur

diesem eine Arbeitspaketdefinition zugeteilt und lädt die darin angegebenen Daten vom Datenserver herunter. Eine solche Arbeitspaketdefinition beinhaltet sowohl Verweise auf Input-Daten als auch auf Updates der projektspezifischen Applikation. Im Anschluss wird ein projektspezifisches Programm, welches die eigentliche Berechnung durchführt, gestartet und überwacht. Die einzelnen Bestandteile (Core Client und projektspezifisches Programm) sind dabei über das BOINC-Application Programming Interface (kurz BOINC-API) gekoppelt. Dadurch ist es möglich, das Programm zu stoppen oder anwendungsspezifische Grafiken im Rahmen eines Bildschirmschoners darzustellen. Nach der Durchführung einer Berechnung werden die Ergebnisse zum Server übertragen. Ein Client ist nicht auf ein einzelnes Projekt beschränkt, sondern kann (auch parallel) an mehreren BOINC-Projekte teilnehmen. Bedingt durch das zugrunde liegende heterogene und unzuverlässiges Netz liefert BOINC jedes Arbeitspaket in mehreren Instanzen an mehrere Clients aus, von deren Ergebnissen dann dasjenige als das Korrekte gewertet wird, das von der Mehrheit der Clients erzeugt wurde.

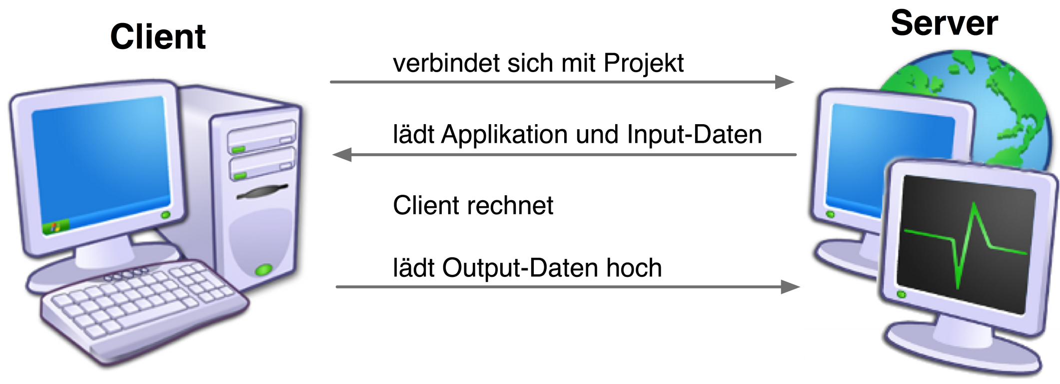

Abbildung 2.5.: BOINC-Kommunikation

2.2.2. Server-Architektur

Ein BOINC-Projekt wird identifiziert anhand einer *Master URL*. Diese URL dient sowohl als Homepage des Projekts als auch als Adresse zur Verbindung zum Server. Die Daten eines BOINC-Projekts (wie Beschreibung der Applikationen, Versionen, Arbeitspakete, Ergebnisse, Benutzerkonten, usw.) werden, wie in Abbildung 2.4 dargestellt, in einer relationalen Datenbank gespeichert. Die Server-Funktionen werden mithilfe von Web-Services und Dämonprozesse[1] gewährleistet. Der Scheduling-Server ist für die Abarbeitung der Client-RPCs verantwortlich und verwaltet somit die logische Verteilung der Arbeitspakete und Berichte über abgeschlossene Berechnungen. Der Daten-Server verwaltet unter Verwendung einer Zertifizierungsmethode das Hochladen der Ergebnisse. Das Herunterladen von Daten erfolgt vom Client aus über reines HTTP.
Die wichtigsten Server-Dämonprozesse sind:

Work Generator Dämon Dieses Programm erzeugt die Arbeitspakete, welche für die Clients zur Berechnung bereitgestellt werden. Die Daten eines Arbeitspaketes sind je nach wissenschaftlicher Anwendung verschieden, somit muss der Work-Generator-Dämon projektspezifisch implementiert werden.

Transitioner Dämon Dieser Dämonprozess verwaltet den Status der Arbeitspakete und Ergebnisse. Der Transitioner ist somit das Modul, welches dafür sorgt, dass Arbeitspakete die einzelnen Phasen bis hin zum fertigem Ergebnis durchlaufen.

[1]Prozesse, die unabhängig von der Steuerung im Hintergrund ausgeführt werden

Feeder Dämon Dieser Dämon füllt die *"Ready To Send"*-Warteschlange mit Arbeitspaketen. Dieser Jobanteil wird vom Feeder Daemon übernommen, da der Scheduler selbst mit Client-Transaktionen zu beschäftigt ist.

Validator Dämon Der Validator Dämon ist ein Back-End-Programm, welches die Validierung der Ergebnisse und Vergabe der Credits übernimmt.

Assimilator Dämon Der Assimilator Dämon ist ein projektspezifisches Modul, welches die Ergebnisse fachlich überprüft.

File Deleter Dämon Der File Deleter Dämon löscht Arbeitspakete und Ergebnisse vom Server, welche nicht mehr benötigt werden. Dadurch sind Download- und Upload-Verzeichnisse eines Projekts nicht mit unnötigen Dateien befüllt.

DB Purge Dämon Dieses Modul entfernt Einträge, welche nicht mehr benötigt werden, aus der relationalen Datenbank. Dadurch wird die Datenbank eines Projekts so klein wie möglich gehalten, was niedrigere Zugriffszeiten ermöglicht.

Die Konfiguration der Dämonprozesse erfolgt über eine *XML*-Datei. Quellcode 2.1 zeigt das Format dieser XML-Datei.

```
1  <boinc>
2    <config>
3      [ configuration options ]
4    </config>
5    <daemons>
6      [ list of daemons ]
7    </daemons>
8    <tasks>
9      [ list of periodic tasks ]
10   </tasks>
11
12 </boinc>
```

Quellcode 2.1: XML-Konfigurationsdatei

Dämonprozesse werden wie in Quellcode 2.2 gezeigt konfiguriert.

```
1  <daemon>
```

```
2    <cmd> feeder -d 3 </cmd>
3    [ <host>host.domain.name</host> ]
4    [ <disabled> 0|1 </disabled> ]
5    [ <output>filename</output> ]
6    [ <pid_file>filename</pid_file> ]
7  </daemon>
8  <daemon>
9  ...
10 </daemon>
```

Quellcode 2.2: Konfiguration der Dämonprozesse

Periodische Tasks werden wie in Quellcode 2.3 darstellt konfiguriert.

```
1      <task>
2        <cmd> get_load </cmd>
3        <output> get_load.out </output>
4        <period> 5 min </period>
5        [ <host> host.ip </host>       ]
6        [ <disabled>  0|1 </disabled>   ]
7        [ <always_run> 0|1  </always_run> ]
8      </task>
9      <task>
10       <cmd> db_dump -d 2 -dump_spec ../db_dump_spec.xml </cmd>
11       <period> 24 hours </period>
12     </task>
13     <task>
14     ...
15     </task>
```

Quellcode 2.3: Konfiguration von periodischen Tasks

Diese Server-Prozesse werden mittels *start*- und *stop*-Befehlen gestartet bzw. gestoppt. Zur Status-Überprüfung wird der *status*-Befehl (s. Abbildung 2.6) verwendet.

Der zeitlicher Ablauf der Abarbeitung eines Arbeitspakets (engl. *Workunit*) soll durch Abbildung 2.7 erläutert werden.

1. Ein Arbeitspaket wird durch einen projektspezifischen Work-Generator erzeugt
2. Ein Client (oder mehrere bei redundanter Berechnung) erhält dieses Arbeitspaket zur Berechnung

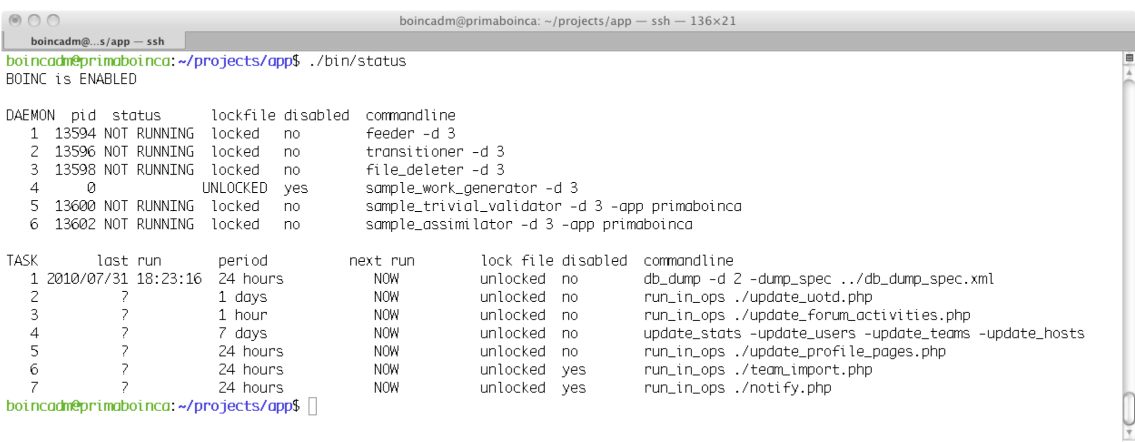

Abbildung 2.6.: Server-Status-Anzeige

3. Ein Client (oder mehrere) führen die Berechnung durch
4. Die Ergebnisse werden durch den Client zum Server hochgeladen
5. Der Server validiert die Ergebnisse
6. Der Server überprüft die Ergebnisse fachlich
7. Das Arbeitspaket wird gelöscht
8. Datenbankeinträge zu diesem Arbeitspaket werden entfernt

Die serverseitige Abarbeitung dieser Schritte wird von den beschriebenen Dämonprozesse geleistet.

2.2.3. Client-Architektur

Die BOINC-Client-Software besteht aus mehreren Komponenten (siehe Abbildung 2.8). Die Applikation ist für die eigentliche Berechnung verantwortlich und somit projektspezifisch. Eine Applikation kann sowohl aus einer ausführbaren Datei (simple application) als auch aus mehreren Dateien (compound application) bestehen.
Der Core-Client ist für folgende Aufgabe verantwortlich:

- Kommunikation mit dem Scheduling Server
- Lokales Scheduling

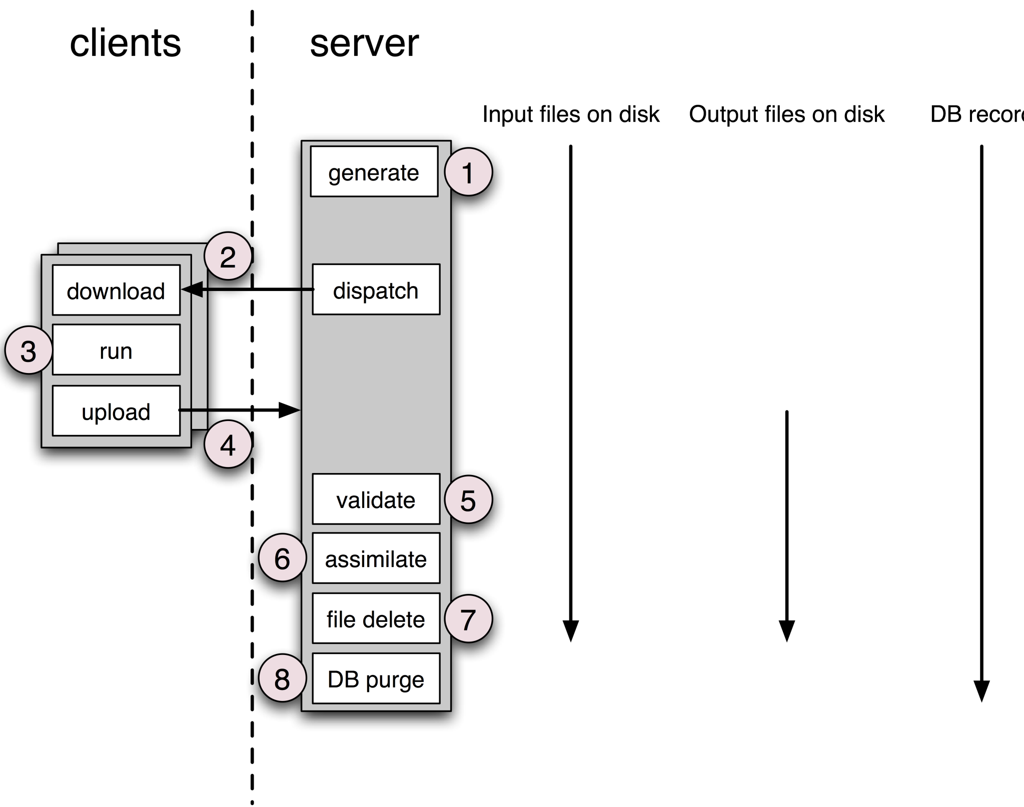

Abbildung 2.7.: Zeitlicher Workflow der Arbeitspakete

- Hoch- und Herunterladen von Dateien
- Ausführung und Kontrolle der Applikation

Der Core-Client startet (sofern von Anwender eingestellt) in einem Rechner mit n Kernprozessoren n Instanzen einer Applikation. Der BOINC-Client kann, wie bereits im Kapitel 2.2.1 beschrieben, mehrere Projekte gleichzeitig unterstützen. Das Scheduling zwischen den Projekten wird vom Core-Client nach dem präemptiven Round-Robin-Prinzip gewährleistet. Somit können Anwender, welche mehrere Projekte parallel unterstützen, einen periodischen Wechsel wahrnehmen. Periodische Überprüfung der laufenden Instanzen auf Speicherverbrauch, Plattenplatzverbrauch und Anzahl FLOPS ist ebenso Aufgabe des Core-Clients. Das lokale Scheduling des Core-Clients, welches festlegt zu welchem Projekt und zu welchem Zeitpunkt Arbeitspakete heruntergeladen und/oder berechnet werden sollen, hat folgende Ziele:

- maximale Ausnutzung der vorhandenen Ressourcen

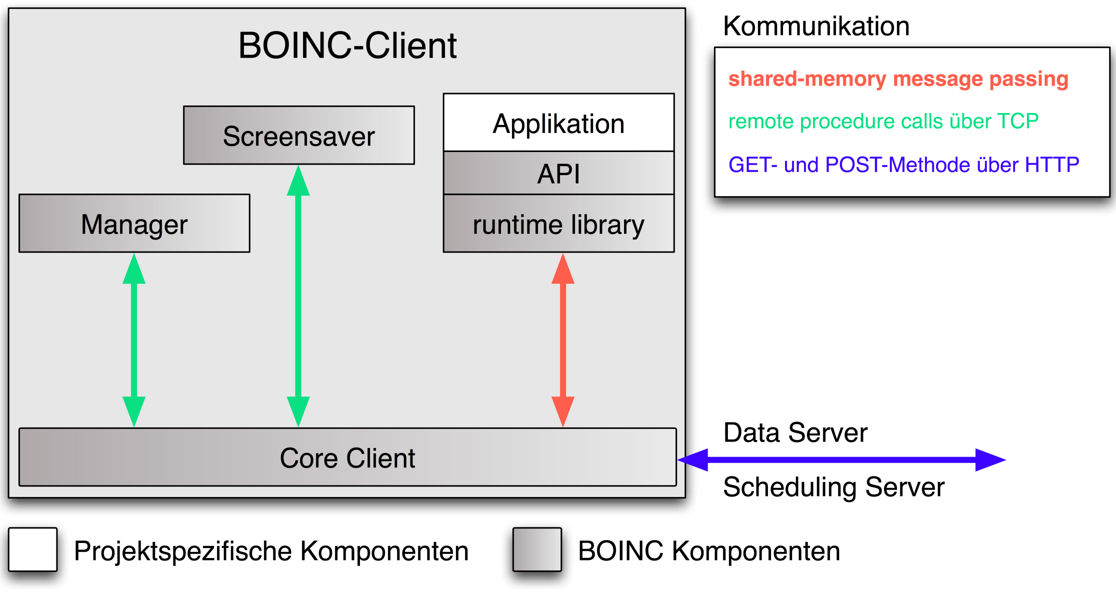

Abbildung 2.8.: BOINC-Client-Architektur nach [ACA06]

- Einhaltung der Ergebnis-Deadlines
- Einhaltung der Ressourcenverteilung zwischen Projekten

Der BOINC-Manager (nur in der grafischen Client-Version vorhanden) ist eine GUI zur Darstellung und Kontrolle der Applikationen (siehe Abbildung 2.9). Die Kommunikation zwischen Core-Client und BOINC-Manager erfolgt über RPC. Durch die Verwendung dieses Kommunikationswegs besteht die Möglichkeit, dass ein BOINC-Manager parallel mehrere BOINC-Client-Instanzen, welche auf unterschiedlichen Rechnern laufen, verwaltet.
Im BOINC-Screensaver (ebenso nur in der grafischen Client-Version vorhanden) werden Grafiken nicht erzeugt, sondern dieser erhält die darzustellenden Bildern über die Kommunikation mit dem Core-Client, welcher wiederum die Grafiken aus der projektspezifischen Applikation anfordert.

Die Kommunikation zwischen einzelnen Client-Komponenten erfolgt unter Verwendung unterschiedlicher Methoden (siehe Abbildung 2.8). Die bidirektionale Kommunikation zwischen Core-Client und Applikation erfolgt über Shared-memory message passing. Für jede gestartete Applikationsinstanz erzeugt der Core-Client ein Shared-Memory-

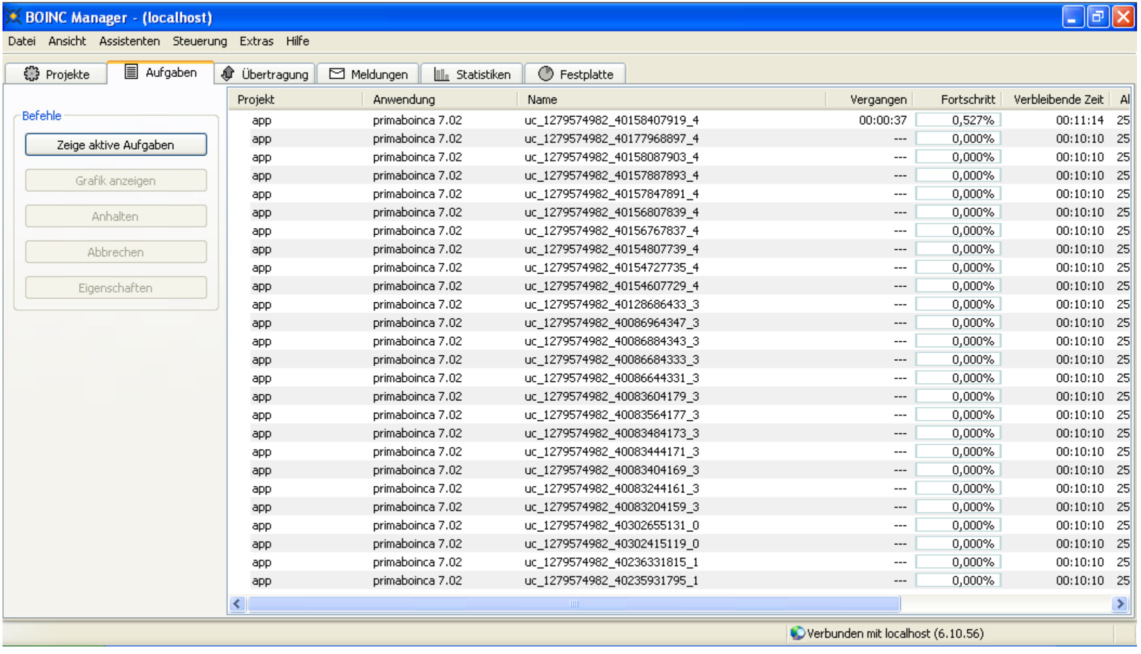

Abbildung 2.9.: BOINC-Manager-GUI

Segment. Jedes Segment beinhaltet eine Datenstruktur mit einer Anzahl unidirektionaler Nachrichtenkanäle. Die Kanäle besitzen eine feste Pufferlänge und einen *present*-Flag. Nachrichtenwarteschlangen werden, wenn notwendig, auf einer höheren Programmebene abgehandelt. Alle Nachrichten werden im XML-Format übertragen.
Das BOINC-Runtime-System verwendet acht Nachrichtenkanäle, jeweils 4 pro Richtung (siehe Abbildung 2.10). Shared-memory hat im Vergleich zu Alternativen wie Pipes,

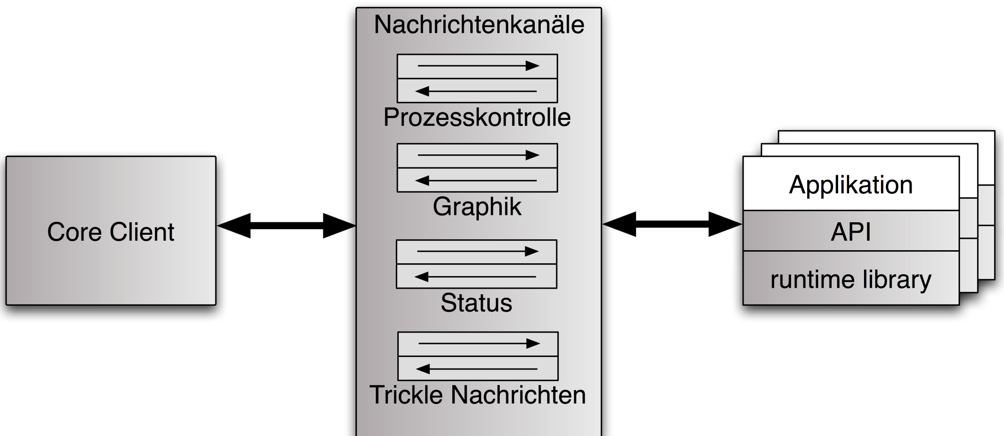

Abbildung 2.10.: Kommunikation zwischen Core-Client und Applikation nach [ACA06]

Sockets oder Signalen folgende Vorteile:

- Alle von BOINC unterstützten Betriebssysteme verwenden Shared-Memory mit ähnlicher Semantik
- Das Handling von Compound-Applikationen ist leichter implementierbar.

BOINC verwendet Threads um nebenläufige Aktivitäten der Applikation zu gewährleisten. Unter Unix werden *pthreads* und auf Windows-Systemen *native threads* eingesetzt. Die Threads-Struktur eines BOINC-Clients unter Unix-Systeme ist wie in Abbildung 2.11 dargestellt aufgebaut.

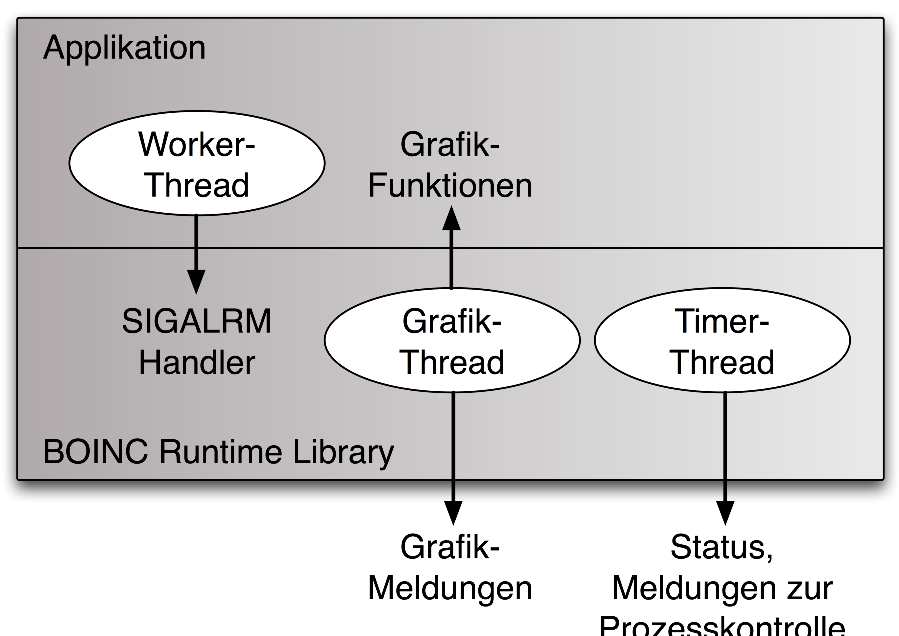

Abbildung 2.11.: Thread-Struktur des BOINC-Runtime-Systems nach [ACA06]

Worker-Thread Der Worker-Thread führt die Hauptapplikation aus und arbeitet 1 Hz Signale (SIGALRM-Handler) ab. Die Abarbeitung dieser Signale erfolgt aus zwei Gründen. Erstens sind CPU-Zeit-Abfragen bei der Verwendung von pthreads zwischen einzelnen Threads nicht erlaubt. In diesem Fall wird die Funktion *getrusage()* verwendet und das Ergebnis in Form einer Variablen gespeichert. Zweitens erlauben pthreads keine suspend- oder resume-Befehle zwischen Threads. In diesem Fall legt sich ein Thread schlafen, wenn ein bestimmtes Flag gesetzt ist.

Grafik-Thread Der Grafik-Thread arbeitet im Loop und fragt im zeitlichen Zyklus die Render-Funktion der Applikation ab.

Timer-Thread Dieser Thread führt periodische Aktivitäten in einem zeitlichen Zyklus von einer Sekunde aus.

Die Thread-Struktur unter Windows ist ähnlich. Lediglich der Timer-Thread und der SIGALRM-Handler werden durch periodische *Multimedia-Timer* ersetzt.

2.3. Multicore Prozessoren

Die Entwicklung der Halbleiterbausteine folgt seit mehr als 40 Jahren dem Gesetz von Moore [Moo65]. Das auf empirischen Beobachtungen beruhende Gesetz von Gordon Moore besagt, dass die Anzahl der Transistoren eines Prozessorchips sich alle 18 bis 24 Monate verdoppelt. Die Nutzung des steigenden Transistorangebots sorgte für zusätzliche Funktionalität des Prozessors und gleichzeitige Erhöhung der Taktfrequenz proportional zum Gesetz von Moore. Bedingt durch die zur Taktfrequenz proportionale Steigung der Leistungsaufnahme der Prozessoren führte diese Entwicklung zum doppelten Problem:

- zu hohe Wärmeentwicklung pro Flächeneinheit im Chip (verursacht durch Leckströme und hohe Taktraten)
- nicht ausreichende Leistung der Batterien tragbarer Geräte

Somit ist eine Erhöhung der Taktrate im bisherigen Umfang für die Zukunft nicht zu erwarten. Das Gesetz von Moore scheint aber bis auf weiteres seine Gültigkeit zu behalten. Prozessorhersteller setzen aus diesen Gründen auf eine explizite Parallelverarbeitung innerhalb eines Prozessors, um eine weitere Leistungssteigerung der Prozessoren im bisherigen Umfang zu ermöglichen. Explizite Parallelität auf Prozessorebene erfolgt durch folgende Architekturorganisation des Prozessorchips:

Mehrkern-Prozessoren Platzierung mehrerer Prozessorkerne mit jeweils unabhängigen Ausführungseinheiten

Hyperthreading Gleichzeitige Ausführung mehrere Kontrollflüsse auf einem Prozessor

Die Relation zwischen Performance und Leistungsaufnahme soll anhand Abbildung 2.12 verdeutlicht werden.

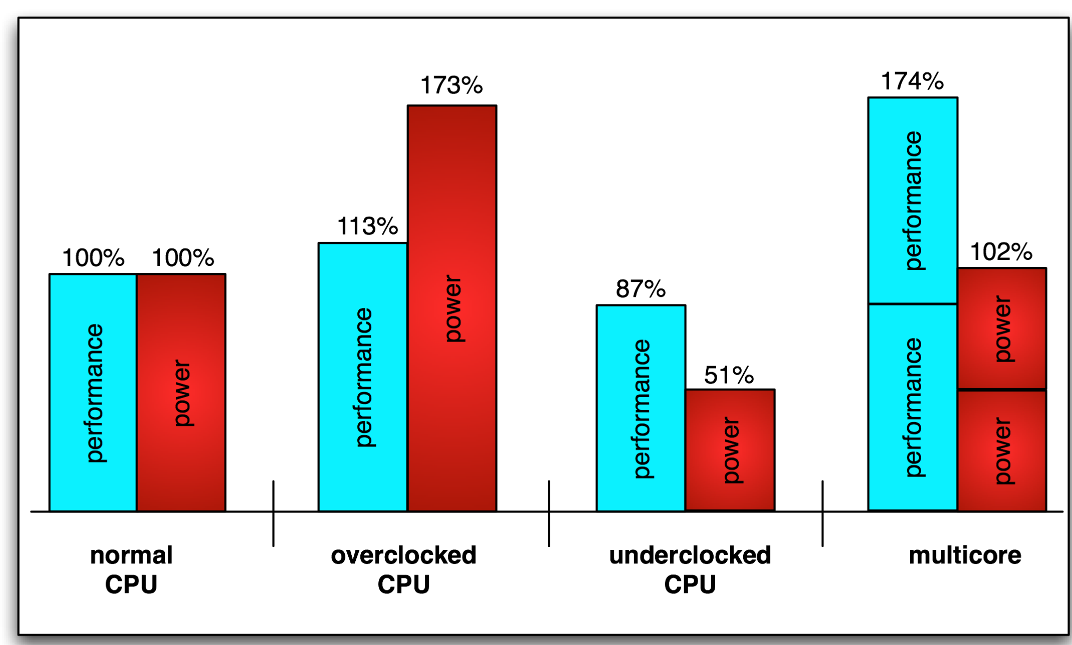

Abbildung 2.12.: Höhere Performance bei geringem Verbrauch nach [Rei09]

Für die Realisierung von Mehrkern-Prozessoren gibt es verschiedene Implementierungsvarianten, welche sich in der Anzahl der Prozessorkerne, der Größe und Anordnung der Caches, den Zugriffsmöglichkeiten auf die Caches und dem Einsatz von heterogenen Komponenten unterscheiden.

2.3.1. Cell Broadband Engine

Der Cell-Broadband Engine (kurz Cell BE) wurde im Februar 2005 [KDH+05] vom *STI Design Center*, einer Kooperation der Unternehmen Sony, Toshiba und IBM, vorgestellt. Im Gegensatz zu herkömmlichen Mehrkern-Prozessoren besteht der Cell-Prozessor nicht

aus mehreren gleichartigen Kernen. Der seit 2001 entwickelte Mehrkern-Prozessor basiert auf 8 einzelnen synergetischen Prozessorkernen (kurz SPE), welche über einen 64-Bit PowerPC-Prozessorkern (kurz PPE) gesteuert werden. Abbildung 2.13 zeigt den schematischen Aufbau des Cell-Prozessors.

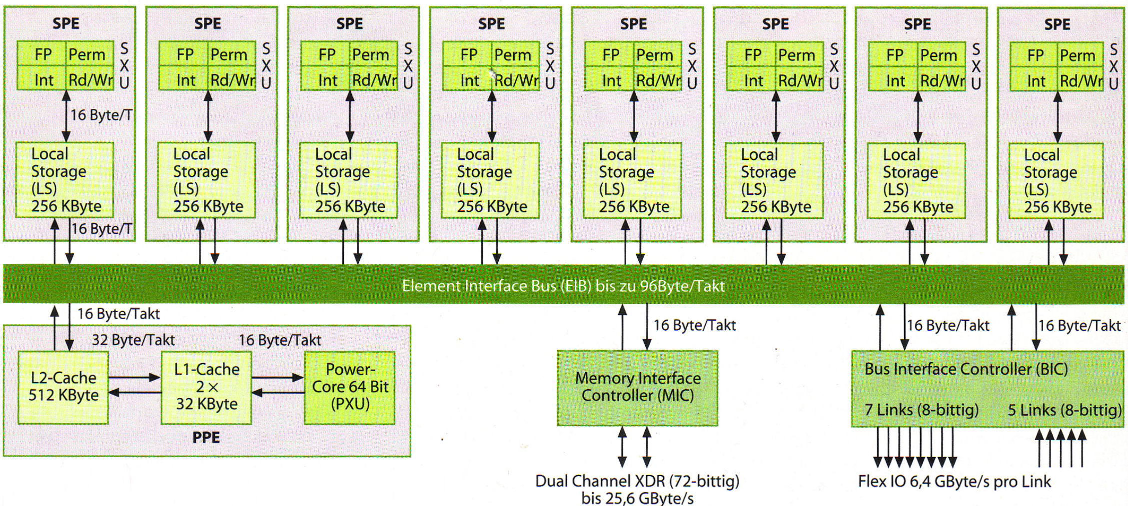

Abbildung 2.13.: Cell BE-Architektur aus [Sti07]

Die folgende Beschreibung der wesentlichen Komponenten dieses Prozessors nimmt Bezug auf die Darstellung aus [KDH+05].

PowerPC-Prozessorkern Der PowerPC-Prozessorkern verfügt über einen 32 kB L1-Instruktions- und Daten-Cache, sowie eine Taktfrequnez von 3,2 GHz. Durch Hyperthreading können zwei Threads nach dem Round-Robin Prinzip[1] zur gleiche Zeit verarbeitet werden. Das Hauptmerkmal des Architekturdesigns ist eine reduzierte Komplexität, um eine höhere Taktfrequenz und geringe Verlustleistung zu gewährleisten.

Synergetische Prozessorkerne Jede SPE besteht aus einer mehrfach parallel arbeitenden Recheneinheit mit 128 Registern, einem lokalen Speicher (256 KB) und einem leistungsfähigen Memory-Flow-Controller (kurz MFC). Dieser Controller tauscht Daten über den Element Interconnect Bus (kurz EIB) mit den anderen SPEs, dem PowerPC-Prozessor, dem Speichercontroller und den beiden Bus-Interfaces aus. Diese SPEs sind nicht zu

[1] ist ein Scheduling-Verfahren

Steuerungsaufgaben, wie z.B. der Ausführung eines Betriebssystems fähig, aber insbesondere für Fließkommaberechnungen hocheffizient. Durch die 8 SPEs erreicht der Cell-Prozessor eine maximale (theoretische) Fließkommaleistung (bei einfacher Genauigkeit) von 230 GigaFLOPS[1].

Element Interconnect Bus Die zwölf Hauptelemente auf dem Cell-Chip sind durch den Element Interconnect Bus verbunden. Dieser ringförmige Bus besteht aus vier Ringe (je zwei in beiden Richtungen) mit jeweils 128 Bit Breite. Jedes der Hauptelemente ist bidirektional mit zwei ebenfalls 128-bittigen Ports an den Bus angeschlossen. Bei halbem Prozessortakt von 1,6 GHz erreicht dieser Bus einen Durchsatz von 25,6 GByte pro Sekunde pro Richtung. Der Speichercontroller (kurz MIC) läuft unabhängig vom Kerntakt mit 1,6 GHz und erreicht eine Speicherbandbreite von 25,6 GByte pro Sekunde.

Der Cell-Prozessor findet sich inzwischen nicht nur in der aktuellen Spielekonsole PlayStation 3 von Sony wieder, sondern wird aufgrund seiner außergewöhnlichen Architektur und theoretischer Leistungsfähigkeit auch in der Welt des wissenschaftlichen Rechnens intensiv diskutiert oder bereits praktisch eingesetzt [CS09].

Für diese Arbeit standen sowohl Playtsation®-Konsolen der Marke Sony Computer Entertainment Europe als auch ein IBM BladeCenter samt sieben QS20 Blades zur Verfügung. Die Hauptbestandteile der QS20 Blades sind folgende:

- 2 Cell-Prozessoren (somit 16 SPEs)
- 1 GB XDRAM Speicher (512 MB pro Prozessor)
- 2 Gigabit Ethernet Ports
- Infiband (4x)

[1] 1 GigaFLOPS = 10^9 FLOPS (*Floating point operations per second*), bezeichnet 1 Milliarde Fließkommaoperationen pro Sekunde.

2.4. Profiler

Profiler werden zur Analyse des Laufzeitverhaltens von Anwendungen verwendet. Ein solches Werkzeug hilft Entwicklern durch Analyse und Vergleich von laufenden Programmen, mögliche Problembereiche aufzudecken. Profiler werden häufig zur Bemessung folgende Kenngrößen angewendet:

Laufzeit Die häufigste Anwendung eines Profilers ist das Zählen und Messen der Aufrufe und Durchläufe von Funktionen. Dies ermöglicht es dem Programmierer herauszufinden, wo sich eine Optimierung des Programms lohnt. Eine Optimierung von Funktionen, die nicht häufig verwendet werden, ist der Gesamtleistung des Programms nicht sonderlich zuträglich und erschwert in der Regel die Wartbarkeit des Quellcodes. Deshalb wird das Hauptaugenmerk auf Funktionen gelegt, die oft aufgerufen werden und in der Summe der Aufrufe viel Zeit benötigen.

Speichernutzung Ein weiterer Aspekt ist die Verfolgung von Speichernutzung durch ein Programm. Der Profiler soll dabei helfen, den Ge- und Verbrauch von Arbeitsspeicher zu optimieren und ggf. Fehler in der Programmierung aufzudecken, durch die ungenutzte Speicherbereiche nicht freigegeben werden (Speicherleck).

Nebenläufigkeit Moderne Profiler bieten die Möglichkeit, nebenläufige Prozesse (Threads) in ihrem Lebenszyklus grafisch (zum Beispiel als Balken- oder Netzdiagramm) darzustellen. Diese optische Aufbereitung soll einem Programmierer helfen, das Laufzeitverhalten von nebenläufigen Prozessen besser zu interpretieren und Fehler durch Deadlock aufzudecken.

In dieser Thesis wurden Profiler zum einem genutzt um die rechenintensiveren Stellen im Algorithmus zu lokalisieren und zum anderen, um die Speichernutzung zu optimieren. Als Werkzeuge kamen Shark, gProf und Valgrind zur Verwendung. Diese Werkzeuge werden in den folgenden Unterkapiteln genauer beschrieben.

2.4.1. gprof - Der GNU Profiler

Mit gprof[1] kann die Verteilung von Laufzeiten innerhalb eines Programms festgestellt werden. Die Häufigkeit der Aufrufe einzelner Funktionen lässt sich ebenso bemessen. Anhand solcher Aussagen sind gezielte Maßnahmen zur Beschleunigung des Algorithmus möglich. gprof verwendet Informationen, welche durch das zugehörige Laufzeitsystem während der Ausführung des Programms gesammelt werden. Die Erzeugung eines solchen Profils erfolgt in mehreren Schritten. Das Programm muss zunächst mit der Profiling-Option *pg* kompiliert und gelinkt werden. Anschließend wird das kompilierte Programm ausgeführt. Während der Ausführung wird eine Profiling-Datei, welche die im Programmlauf erfassten Informationen enthält, erzeugt. Die Auswertung der erstellten Profiling-Datei wird durch den Aufruf von *gprof* samt Optionenparameter erstellt.

2.4.2. Shark

Shark[2] ist ein proprietäres Werkzeug und Bestandteil des XCode developer tools[3] von Mac OS X[4]. Shark ermöglicht eine Beobachtung des Laufzeitverhaltens eines Algorithmus sowohl im C/C++- als auch im Assembler-Code. Dies erwies sich als sehr hilfreich. Da einzelne Zeilen im C/C++-Code mehreren Assembler-Anweisungen entsprechen, lassen sich gezieltere Optimierungen, welche den Assembler-Code mitberücksichtigt, leichter verwirklichen. Die graphische Oberfläche dieses Werkzeugs wird in Abbildung 2.14 dargestellt.

2.4.3. Valgrind

Valgrind[5][6] ist ein Analyseframework für unixbasierte Systeme, welches als Open-Source-Software unter GNU Public License verfügbar ist. Valgrind kann sowohl als flexibles

[1] http://sourceware.org/binutils/docs/gprof/index.html
[2] http://developer.apple.com/
[3] http://developer.apple.com/technologies/tools/
[4] http://www.apple.com/de/macosx/
[5] in der Mythologie der Haupteingang zur Walhalla (grind = "Gitter")
[6] http://valgrind.org/

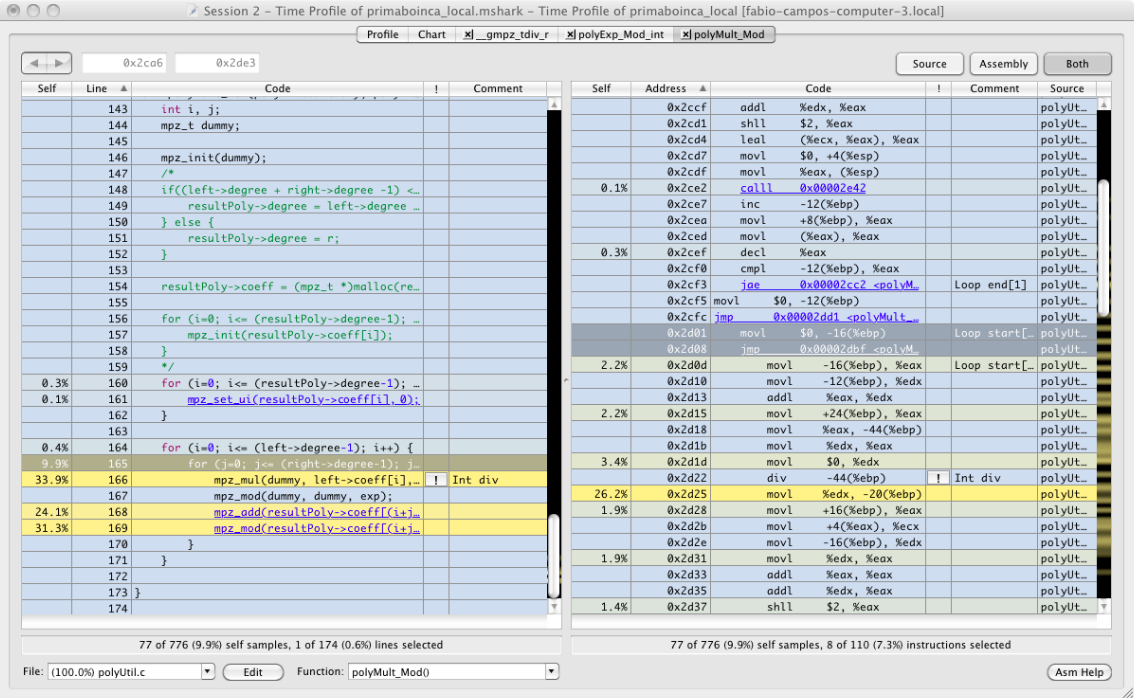

Abbildung 2.14.: Shark-GUI

Framework zur Erstellung eigener Werkzeuge als auch als fertige Werkzeugsammlung zur Analyse des Laufzeitverhaltens verwendet werden.

Valgrind führt Programme zur Analyse vollständig in einer virtuellen Maschine aus. Programme werden in temporären plattform-unabhängigen Byte-Code übersetzt. Jedes Werkzeug aus dieser Sammlung arbeitet in verschiedenen Detaillierungsstufen. Auf diese Weise kann die Art und technische Tiefe der Analyse exakt vorgegeben werden. In dieser Thesis wurde lediglich das Werkzeug *Memcheck* aus der Sammlung zur Speicherüberwachung und Beseitigung von Speicherlecks verwendet. Das Memcheck-Tool kann Fehler folgender Art entdecken:

- Benutzung von nicht initialisiertem Speicher
- Lese- und Schreibzugriffe auf freigegebenen Speicher
- Schreiben über die Speichergrenzen hinaus
- Speicherlecks

2.5. Mathematica

Mathematica wurde in dieser Arbeit zum einen zur Erzeugung von Zahlen mit bestimmten Eigenschaften zu Testzwecken und zum anderen zur Überprüfung bestimmter mathematischer Sachverhalte eingesetzt. Der Quellcode 2.4 soll als Beispiel eines Mathematica-Algorithmus dienen. Dieser Algorithmus diente zur Erzeugung von Primzahlen der Länge von 128 bis 1024 Bits.

```
1  createPrimFile[ step ] :=
2    Module[ {z = 0, n, y = 32},
3     Do [
4
5      While [z < 1,
6       n = RandomInteger[{2^y, 2^(y + 1)}];
7       While[(! (PrimeQ[n])), n = n + 1;];
8       s = ToString[n] <> "  " <> ToString[Log[2., n]];
9       s >>> "/Users/sopmac/Desktop/fh/Master/prime_128_1024.txt" ;
10      z++;
11      ];
12     z = 0;
13     , {y, 128, 1024, step}];
14    Print["finished"]
15     ];
```

Quellcode 2.4: Zufällige Primzahlerzeugung in Mathematica

KAPITEL 3

ZAHLENTHEORETISCHE ASPEKTE

Die Zahlentheorie ist seit etwa zwanzig Jahren ein sehr wichtiges Gebiet der Informatik (algorithmische Zahlentheorie). Die Garantie für die Sicherheit fast aller Verfahren der Public-Key-Kryptographie[1] basiert auf der Schwierigkeit, Berechnungsprobleme der Zahlentheorie zu lösen. Viele kryptographische Algorithmen, wie zum Beispiel das RSA-Verfahren, benötigen relativ große Primzahlen. Bei den meisten kryptographischen Protokollen, wie dem Schlüsseltausch nach Diffie-Hellman, steigt der Sicherheitsgrad proportional zur Länge der verwendeten Primzahlen. Der AKS-Algorithmus, welcher im Unterkapitel 3.1 besprochen wird, beschreibt den ersten effizienten Algorithmus zur Primzahlerkennung. Selbst aktuell schnellste Version dieses Algorithmus spielt in der Praxis, bedingt durch die relativ lange Laufzeit, keine Rolle. Während sich Unterkapitel 3.2 mit der AKS-Vermutung befasst, wird die Heuristik von H. Lenstra und C. Pomerance zu dieser Vermuntung im Unterkapitel 3.3 besprochen. Abschließend wird im Unterkapitel 3.4 Popovychs-Vermutung als logische Verstärkung der AKS-Vermutung beschrieben.
Die zahlentheoretischen Aspekte aus diesem Kapitel dienten als Grundlagen zur Formulierung des implementierten Algorithmus.

3.1. AKS-Algorithmus

Ausgangspunkt vieler Verfahren zur Primzahlüberprüfung ist der kleine Satz von Fermat. Dieser Satz besagt, dass:

[1] ist ein Kryptosystem, bei dem jede der kommunizierenden Parteien ein Schlüsselpaar besitzt, das aus einem geheimen Teil (privater Schlüssel) und einem nicht geheimen Teil (öffentlicher Schlüssel) besteht.

Satz 1 (Der kleine Satz von Fermat) *Sei $n \in \mathbb{P}$, dann gilt*

$$a^n \equiv a \mod n. \tag{3.1}$$

Dieser Satz ist eine Implikation und keine Äquivalenz, somit gilt die Umkehrung des Satzes nicht. Zusammengesetzte Zahlen, welche die Gleichung 3.1 erfüllen, werden als Pseudoprimzahlen bezeichnet. Im Allgemeinen spricht man von Pseudoprimzahlen zur Basis a , also zerlegbare Zahlen $n > a$, welche die Gleichung 3.1 erfüllen. Die kleinste Pseudoprimzahl zur Basis 2 ist die Zahl 341, denn es gilt $341 = 11 \times 31$ und weiterhin gilt $2^{341} \equiv 2 \pmod{341}$. Primzahlen lassen sich anhand dieses Satzes somit nicht charakterisieren. Eine Verallgemeinerung dieses Satzes auf Polynome, lautet wie folgt:

Satz 2 (Verallgemeinerung des kleinen Satz von Fermat) *Sei $ggT(n, a) = 1$ dann gilt*

$$(x - a)^n \equiv (x^n - a) \mod n \tag{3.2}$$

genau dann, wenn $n \in \mathbb{P}$

Diese Verallgemeinerung des Satzes führt zu einem nicht effizienten Primzahltest. Denn die Anwendung dieses Satzes als Primzahltest würde mit dem *Square and multiply-Potenzalgorithmus* (siehe Unterkapitel 4.3.1) etwa $log\, n$ Multiplikationen von Polynomen erfordern, welche aber immer aufwendiger werden. Im letzten Schritt sind zwei Polynome vom Grad etwa $\frac{n}{2}$ zu multiplizieren, was einen Aufwand der Größenordnung n erfordert. Dieser Satz kann anhand der binomischen Kongruenz wie folgt bewiesen werden:
Hilfssatz: Seien $a, b \in \mathbb{Z}$ und $p \in \mathbb{P}$ eine Primzahl, dann gilt:

$$(a + b)^p \equiv a^p + b^p \bmod p.$$

Beweisskizze: Nach der binomischen Formel gilt:

$$(a + b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p + \sum_{i=1}^{p-1} \binom{p}{i} a^i b^{p-i}.$$

Weiterhin gilt, dass $\binom{p}{i}$ für $0 < i < p$ durch p teilbar, denn:

$$\binom{p}{i} = \frac{p!}{i!(p-i)!}$$

beim Kürzen von $p!$ durch $i!(p-i)!$ bleibt ein Faktor p stehen.
Somit gilt:

$$(x-a)^n \equiv (x^n - a) \mod n$$

Der kleine Satz von Fermat kann ebenso anhand dieses Hilfssatzes induktiv bewiesen werden.

Agrawal und Biswas entwickelten einen Gleichheitstest, welcher auf die vollständige Aufstellung der Polynome verzichtet. Dadurch ist es möglich, die Gleichung 3.2 zu überprüfen, ohne das Polynom $(x-a)^n$ vollständig berechnet zu haben. Aus dieser Überlegung ist der erste deterministische Primzahltestalgorithmus mit polynomialer Laufzeit entstanden. Die Formulierung eines solchen Algorithmus war lange Zeit das Hauptziel eines Forschungsbereichs. Der AKS-Algorithmus (benannt nach seinen Erfindern: M. Agrawal und N. Kayal and N. Saxena) [AKS02] beschreibt einen solchen Algorithmus ohne jegliche Annahme. Die Formulierung des AKS-Algorithmus lautet wie folgt:

Dieser Algorithmus kann nach [CPCP05] in drei Hauptteile unterteilt werden:

1. Power-Test Im ersten Schritt des AKS-Algorithmus wird überprüft, ob es sich bei der zu untersuchenden Zahl um eine höhere Potenz handelt. Die genaue Formulierung innerhalb des AKS-Algorithmus lautet:
`if` $(N = a^b$ `for` $a \in \mathbb{N}$ `and` $b > 1)$, `output ZUSAMMENGESETZT.`

Dieser Schritt kann unter Verwendung der Newton-Iteration aus [Oev96], wie in [CP08] beschrieben, effizient implementiert werden.

2. Setup Im zweiten Schritt des AKS-Algorithmus wird der Wert r berechnet. Die Komplexität des Algorithmus ist von diesem Wert abhängig. Dieser Wert ist wie folgt zu wählen:
`Finde das kleinste` r `mit` $O_r(N) > (log\ \ N)^2$

Algorithmus 3.1 AKS-Algorithmus

```
Require: n > 1
  if (n = a^b for a ∈ ℕ and b > 1) then
    output COMPOSITE.
  end if
  Find the smallest r such that o_r(n) > log^2 n.
  if 1 < (a, n) < n for some a ⩽ r then
    output COMPOSITE.
  end if
  if n ⩽ r then
    output PRIME.
  end if
  for a = 1 to ⌊√φ(r) log n⌋ do
    if ((X + a)^n ≠ X^n + a (mod X^r − 1, n)) then
      output COMPOSITE.
    end if
  end for
  output PRIME.
```

Mit r baut man das Polynom $(x^r - 1)$ auf, in dessen Restklassenring die Polynomoperationen aus dem folgenden Schritt berechnet werden.

3. Binomische Kongruenz Im dritten Schritt wird anhand des im zweiten Schritt errechneten r-Wertes mit Hilfe der binomischen Kongruenz die Zahl n auf Primalität getestet. Dieser Schritt wird im AKS-Algorithmus wie folgt formuliert:

```
for a = 1 to ⌊√φ(r) log N⌋ do
    if ((X + a)^N ≠ X^N + a mod (X^r − 1, N)), output ZUSAMMENGESETZT;
output PRIM.
```

Der AKS-Algorithmus verkürzt diese Berechnung anhand des im zweiten Schritt errechneten r-Wert, so dass die Überprüfung der binomischen Kongruenz mit einem Polynom des Grads r möglich ist.

Der AKS-Algorithmus basiert, wie die meisten in der Einleitung angesprochenen Primzahltests, auf Satz 1 (kleiner Satz von Fermat). Dies ist nicht sofort ersichtlich, kann aber wie folgt gezeigt werden:

Angenommen $a = 1$ und $r = 1$, dann gilt:

$$\underbrace{(x+1)\ldots(x+1)}_{n} \bmod(x-1, n) = x^0 + 1,$$

und da $(x+1) \bmod (x-1) = 2$, folgt:

$$2^n \bmod n = 2. \tag{3.3}$$

Gleichung 3.3 stellt nichts anders als den kleinen Satz von Fermat zur Basis 2 (siehe Gleichung 3.1) dar.

3.2. AKS-Vermutung

Die folgende Vermutung wurde bereits in [BP01a] aufgestellt. In [AKS02] wurde diese Vermutung erneut veröffentlicht. Eine in [AKS02] vorgeschlagene Variante, welche die Vermutung um die Bedingung $r > logn$ erweitert, wurde in dieser Arbeit nicht berücksichtigt. In der Originalversion lautet die Formulierung dieser Vermutung wie folgt:

Vermutung 1 (AKS-Vermutung) *Sei r eine Primzahl, welche n nicht teilt und sei*

$$(X-1)^n = X^n - 1 \ (mod\ X^r - 1, n) \tag{3.4}$$

dann entweder n eine Primzahl oder es gilt

$$n^2 = 1(mod\ r). \tag{3.5}$$

.

Zu dieser Vermutung gibt es bis heute laut H. Lenstra aus [Cam10a] und R. Popovych aus [Cam10b] weder einen Beweis noch einen Gegenbeweis.

3.3. Lenstras Heuristik

Der folgende Satz von H. Lenstra und C. Pomerance aus [LP02] legt nahe, dass die AKS-Vermutung so nicht wahr sein sollte.

Satz 3 (Lenstras Heuristik) *Seien $p_1, \dots, p_k$ mit $k \equiv 1 \mod 4$ verschiedene Primzahlen und n deren Produkt. Für jede solche Primzahl p_j gelte ferner $p_j \equiv 3 \mod 80$ sowie $(p_j \pm 1) \mid (n \pm 1)$. Dann ist*

$$(X-1)^n \equiv X^n - 1 \mod (n, X^5 - 1)$$

sowie $n^2 \not\equiv 1 \mod 5$.

Dieser Satz kann wie folgt bewiesen werden:

Beweisskizze: Nach Wahl der p_j ist $n \equiv 3 \mod 5$ und also $n^2 \not\equiv 1 \mod 5$. Ferner ist das fünfte Kreisteilungspolynom $P = X^4 + X^3 + X^2 + X + 1$ im Ring $\mathbb{Z}[X]$ teilerfremd zu $X - 1$ und also das Ideal

$$(P, X-1) = (1)$$

ein Hauptideal gleich $\mathbb{Z}[X]$. Damit genügt es also,

$$(X-1)^n \equiv X^n - 1 \mod (n, P)$$

zu zeigen.

Mit dem chinesischen Restsatz besteht der Isomorphismus von Ringen

$$\mathbb{Z}[X]/(n, P) \cong \prod_{j=1}^{k} \mathbb{F}_{p_j}[X]/P$$

und jeder Faktor rechts ist ein Körper, nämlich der Zerfällungskörper von $\mathbb{F}_{p_j}(\zeta)$, wobei ζ eine primitve fünfte Einheitswurzel ist (etwa $\zeta = \exp(2\pi i/5)$), denn jeweils ist p_j teilerfremd zu 5. Insbesondere genügt es also,

$$(\zeta - 1)^n = \zeta^n - 1$$

in $\mathbb{F}_p(\zeta)$ für jede Primzahl $p = p_j$ zu verifizieren.

Nach Wahl der Primzahlen p_j errechnet sich mit der Binomialentwicklung

$$(\zeta - 1)^{p^2} = \zeta^{p^2} - 1 = \zeta^{-1} - 1 = -\zeta^{-1}(\zeta - 1)$$

bzw.

$$(\zeta - 1)^{p^2 - 1} = -\zeta^{-1}$$

und mittels $(-\zeta^{-1})^{10} = 1$ folgt, dass die Ordnung von $(\zeta - 1)$ in $\mathbb{F}_p(\zeta)$ die Zahl $10(p^2 - 1)$ teilt. Damit verbleibt nachzuweisen, dass

$$n \equiv p \bmod 10(p^2 - 1)$$

gilt. Mit Blick auf den chinesischen Restsatz faktorisiert man hierzu

$$10(p^2 - 1) = 5 \cdot 2^4 \cdot \frac{p-1}{2} \cdot \frac{p+1}{4}$$

und weist die Kongruenz faktorweise nach; hier geht wiederum die spezielle Wahl der Primzahlen p_j ein.

Ein Algorithmus zur Berechnung von Zahlen, welche die Anforderungen des Satzes von Lenstra erfüllen, kann wie in Algorithmus 4.7 aus Unterkapitel 4.5 formuliert werden.

3.4. Popovychs Vermutung

Die AKS-Vermutung wird durch Popovychs-Publikation [Pop09] um eine weitere Bedingung erweitert. In dieser Publikation lautet die Vermutung wie folgt:

Vermutung 2 (Popovychs Vermutung) *Sei r eine Primzahl, welche n nicht teilt und sei*

$$(X - 1)^n = X^n - 1 \; (mod\; X^r - 1, n) \tag{3.6}$$

und

$$(X+2)^n = X^n + 2 \; (mod \, X^r - 1, n) \tag{3.7}$$

dann ist entweder n eine Primzahl oder es gilt

$$n^2 = 1 (mod \, r). \tag{3.8}$$

Popovychs Erweiterung basiert auf einer monoton steigenden Anordnung der Untergruppen der Gruppe $(Z_p[X]/((C_r(X)))^*$.

KAPITEL 4

IMPLEMENTIERUNG

Um zunächst den Fokus auf den zahlentheoretischen Teil des Projekts zu legen, wurden die einzelnen Schritte des Algorithmus ohne den Ansatz für verteiltes Rechnen implementiert. Maßnahmen zur Beschleunigung der Berechnung wurden zum Teil bereits in dieser Version des Algorithmus formuliert.
Im folgenden Unterkapitel werden basierend auf den zahlentheoretischen Aussagen aus Kapitel 3 weitere Überlegungen formuliert. Diese Überlegungen und der Rechenaufwand einzelner Schritte führten schließlich zum Aufbau des Algorithmus. Eine Betrachtung der Relation zwischen den möglichen Ergebnismengen der Berechnung dient der Verdeutlichung der möglichen Fälle. Unterkapitel 4.2 soll die grobe Implementierungsvorgehensweise beschreiben. Anschließend werden im Unterkapitel 4.3 die Details der Implementierung besprochen. Unterkapitel 4.4 geht auf die Besonderheiten der BOINC-Version der Anwendung ein. Das letzte Unterkapitel 4.5 beschreibt Varianten der Implementierung eines Algorithmus zu Lenstras Heuristik.

4.1. Grundlegende Überlegungen

Die zu untersuchenden Vermutungen basieren auf Überprüfungen verschiedener Gleichungen. Anhand von aussagenlogischen Variablen wurde, wie im folgenden Unterkapitel beschrieben, eine logisch äquivalente Umstrukturierung des Algorithmus formuliert.

4.1.1. Logische Formulierung der Vermutungen

Popovychs- und die AKS-Vermutung lassen sich anhand folgender aussagenlogischer Formeln darstellen:

Definition 1 (Aussagenlogische Formel zur AKS-Vermutung) *Sei $n \in \mathbb{N}$ die zu überprüfende Zahl. Seien A, B, C und D aussagenlogische Variablen und weiterhin gilt:*

- $A \triangleq (r \in \mathbb{P}) \wedge (r \nmid n)$
- $B \triangleq (X-1)^n \equiv X^n - 1 \ (mod\, n, X^r - 1)$
- $C \triangleq n \in \mathbb{P}$
- $D \triangleq n^2 \equiv 1 \ (mod\, r)$

Dann gilt $(A \wedge B) \Rightarrow ((C \wedge \neg D) \vee (\neg C \wedge D))$

Definition 2 (Aussagenlogische Formel zu Popovychs-Vermutung) *Sei $n \in \mathbb{N}$ die zu überprüfende Zahl. Seien A, B, B_1, C und D aussagenlogische Variablen und weiterhin gilt:*

- $A \triangleq (r \in \mathbb{P}) \wedge (r \nmid n)$
- $B \triangleq (X-1)^n \equiv X^n - 1 \ (mod\, n, X^r - 1)$
- $B_1 \triangleq (X+2)^n \equiv X^n + 2 \ (mod\, n, X^r - 1)$
- $C \triangleq n \in \mathbb{P}$
- $D \triangleq n^2 \equiv 1 \ (mod\, r)$

Dann gilt $(A \wedge B \wedge B_1) \Rightarrow ((C \wedge \neg D) \vee (\neg C \wedge D))$

4.1.2. Aufbau des Algorithmus

Eines der Hauptziele dieser Arbeit ist die Suche nach einem Gegenbeispiel für diese Vermutungen. Somit sind nur Fälle relevant, in denen die Variablen A, B und C oder entsprechend A, B, B_1 und C gelten. Diese Überlegung führte zur folgenden Umformulierung

der aussagenlogischen Formel: $(A \wedge B \wedge \neg D) \Rightarrow C$ bzw. $(A \wedge B \wedge B_1 \wedge \neg D) \Rightarrow C$. Die Äquivalenz beider logischen Aussagen für die relevanten Fälle $(A \wedge B \wedge \neg D)$ soll anhand der Wahrheitstabelle 4.1 verdeutlicht werden.

A	B	C	D	$A \wedge B$	$(\neg C \wedge D) \vee (C \wedge \neg D)$	$(A \wedge B) \Rightarrow (\neg C \wedge D) \vee (C \wedge \neg D)$	$(A \wedge B \wedge \neg D) \Rightarrow C$
1	1	0	0	1	0	0	0
1	1	1	0	1	1	1	1

Tabelle 4.1.: Äquivalenz aussagenlogischer Formeln

Die Überprüfung der Bedingung $B \triangleq (X-1)^n \equiv X^n - 1 \pmod{n, X^r - 1}$ ist deutlich rechenintensiver als die Überprüfung der Bedingung $D \triangleq n^2 \equiv 1 \pmod{r}$. Diese Überlegung führte dazu, dass die Reihenfolge der Überprüfungen wie in Algorithmus 4.1 im Unterkapitel 4.3 implementiert wurden.

4.1.3. Ergebnismengen

Popovych's Vermutung stellt eine konjunktive Verstärkung der AKS-Vermutung dar. Mengentheoretisch betrachtet führt diese Aussage zur folgenden Überlegung:

Definition 3 (Ergebnismenge der AKS-Vermutung) $\mathbb{P}_{aks} =_{def}$ *die Menge der Zahlen, welche durch die AKS-Vermutung als Primzahlen erkannt werden.*

Definition 4 (Ergebnismenge der Popovychs-Vermutung) $\mathbb{P}_{pop} =_{def}$ *die Menge der Zahlen, welche durch Popovychs Vermutung als Primzahlen erkannt werden.*

Korollar: Dann gilt $\mathbb{P}_{pop} \subseteq \mathbb{P}_{aks}$.

Aus dieser Folgerung ergeben sich folgende mögliche Beziehungen zwischen dieser Mengen:

Sei $\mathbb{P} =_{def}$ die Menge der Primzahlen, dann:

1. Fall: Sei $\mathbb{P}_{pop} = \mathbb{P}_{aks}$ und $\mathbb{P}_{aks} = \mathbb{P}$, dann sind beide Vermutungen wahr und Popovychs-Vermutung kann vernachlässigt werden. In diesem Fall existiert kein Gegenbeispiel.

2. Fall: Sei $\mathbb{P}_{pop} = \mathbb{P}_{aks}$ und $\mathbb{P}_{aks} \neq \mathbb{P}$, dann sind beide Vermutungen falsch und Popovychs-Vermutung kann vernachlässigt werden. In diesem Fall existiert für beide Vermutungen ein Gegenbeispiel.

3. Fall: Sei $\mathbb{P}_{pop} \subset \mathbb{P}_{aks}$ und $\mathbb{P}_{pop} = \mathbb{P}$, dann werden durch die AKS-Vermutung zusammengesetzte Zahlen als Primzahlen erkannt. In diesem Fall existiert ein Gegenbeispiel für die AKS-Vermutung.

4. Fall: Sei $\mathbb{P}_{pop} \subset \mathbb{P}_{aks}$ und $\mathbb{P}_{aks} = \mathbb{P}$ folgt $\mathbb{P}_{pop} \subset \mathbb{P}$, dann werden durch Popovychs-Vermutung nur eine Teilmenge der Primzahlen erkannt. In diesem Fall existieren durch Popovychs-Vermutung nicht erkannte Primzahlen.

5. Fall: Sei $\mathbb{P}_{pop} \subset \mathbb{P}_{aks}$, $\mathbb{P}_{aks} \neq \mathbb{P}$ und $\mathbb{P}_{pop} \neq \mathbb{P}$, dann sind beide Vermutung falsch. In diesem Fall existiert für beide Vermutungen ein Gegenbeispiel.

Für den Zahlenbereich $n < 10^{10}$ und $r < 100$ gilt aus [BP01a] s $\mathbb{P}_{aks} = \mathbb{P}$. In dieser Arbeit soll im Zahlenbereich ab $n > 10^{10}$ und $r < 100$ die Beziehung zwischen diesen Mengen $(\mathbb{P}_{pop}, \mathbb{P}_{aks}, \mathbb{P})$ überprüft und somit für diesen errechneten Zahlenbereich der passende Fall bestimmt werden.

Bemerkung 1 (Fallunterscheidung anhand der Gruppenmächtigkeit) *Eine Fallunterscheidung anhand der Mächtigkeit wurde nicht in Betracht gezogen. Denn sei* $\mathbb{P}_{err} =_{def}$ *die Menge der durch eine der Vermutungen falsch erkannten Primzahlen und* $\mathbb{P}_{ok} =_{def}$ *die Menge der durch eine der Vermutungen erkannten Primzahlen, dann besteht die Möglichkeit, dass obwohl* $\mathbb{P}_{err} U \mathbb{P}_{ok} \neq \mathbb{P}$*, dennoch gilt* $\#\mathbb{P}_{err} + \#\mathbb{P}_{ok} = \#\mathbb{P}$ *.*

4.2. Vorgehensweise

Als Programmiersprache wurde C/C++ verwendet. Zum Umgang mit großen Zahlen wurde die GNU Multiple Precision Arithmetic-Bibliothek (kurz GMP[1]) (sowohl für die Windows- als auch für die Linux-Version) verwendet. GMP gilt als Standard C/C++-Bibliothek für den Umgang mit großen Zahlen. Weiterhin verfügt es über alle für die Implementierung notwendigen mathematischen Funktionen und wurde aus diesen Gründen ausgewählt.
In der Cell BE-Version kamen sowohl die GMP-Bibliothek (für Berechnungen auf der PPE) als auch die IBM Multi-Precision-Bibliothek (kurz MPM) (für Berechnungen auf den SPUs) zum Einsatz. Die Vorgehensweise der Implementierung in dieser Arbeit erfolgte wie in Abbildung 4.1 dargestellt. Um den Fokus auf den theoretischen Teil der Implementierung zu setzen, wurden zunächst BOINC unabhängige Versionen implementiert. Die Implementierungsschritte in Abbildung 4.1 sollen hierbei keine zeitliche Reihenfolge darstellen.

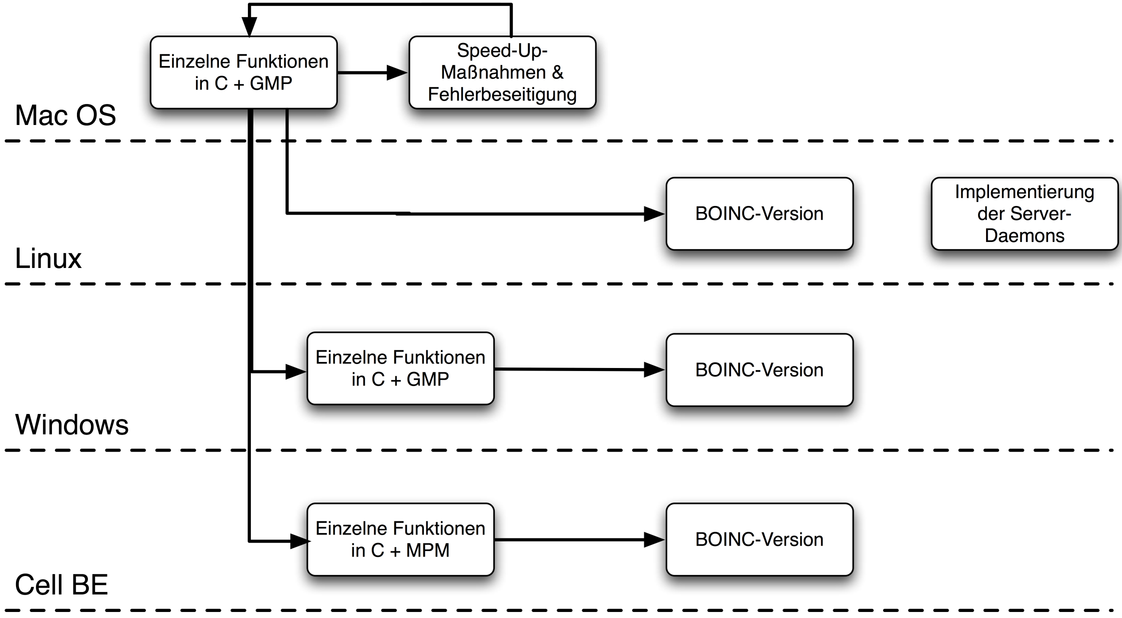

Abbildung 4.1.: Vorgehensweise bei der Implementierung

[1] http://gmplib.org/

Als Entwicklungs- und Testplattform dienten:

- ein Macbook mit einem 1.83 GHz Intel Core Duo-Prozessor unter Mac OS X (10.6.4)
- Spielkonsolen (Playstation 3) der Marke Sony Computer Entertainment Europe mit Ubuntu 9.10 (Karmic Koala)
- eine Linux-VM mit Ubuntu 10.04 (Lucid Lynx)
- eine Linux-VM mit Ubuntu 8.04 (Hardy Heron)
- eine Linux-VM mit Ubuntu 6.10 (Edgy Eft)
- eine Windows-VM mit Windows-XP-Professional
- ein IBM Cell BE-Blade-Server

Zur Virtualisierung wurde VirtualBox[1] verwendet. Die älteren Linux-VMs dienten zur Überprüfung der Kernel-Abwärtskompatibilität der Anwendung.
Als Server wurde ein vServer mit folgende Eigenschaften verwendet:

- Intel(R) Core(TM) i7 CPU 920 2.67 GHz
- 20 GB Plattenplatz
- 2 GB Arbeitsspeicher
- Debian 5.0

4.3. Algorithmus

Der Algorithmus besteht wie bereits im Unterkapitel 4.1 beschrieben, aus der Überprüfung einzelner Gleichungen. Die folgende Formulierung (Algorithmus 4.1) basiert auf den vorhergehenden Überlegungen und führte zum in dieser Phase der Arbeit implementierten Algorithmus.

[1] http://www.virtualbox.org/

Algorithmus 4.1 Algorithmus zur Überprüfung der Vermutungen

```
Require: n ⩾ 0
  r ← 2
  while r ⩽ √2 do
    // Überprüfung der Bedingung A
    if r ∤ n then
      // Überprüfung der Bedingung ¬D
      if n^2 ≢ 1(mod r) then
        // Überprüfung der Bedingung B1
        if (x + 2)^n ≡ x^n + 2( mod n, x^r − 1) then
          POPPRIME ← true
        end if
        // Überprüfung der Bedingung B
        if (x − 1)^n ≡ x^n − 1( mod n, x^r − 1) then
          AKSPRIME ← true
        end if
        if ((AKSPRIME = true) ‖ ( POPPRIME = true)) then
          isPrime ← mpz_probab_prime(n, 10)
          print Ergebnisse
          return 0
        end if
      end if
    end if
    r ← mpz_nextprime(r)
  end while
```

4.3.1. Polynomarithmetik

Zur algorithmischen Überprüfung des Vermutungen ist eine Bibliothek zur Berechnung polynomarithmetischer Gleichungen (sowohl für die x86- als auch für die Cell-BE-Architektur) entstanden. Folgende Überlegung aus [Knu97] spielten eine große Rolle bei der Implementierung dieser Bibliothek:

> *"The reader should note the similarity between polynomial arithmetic and multiple-precision arithmethic .... The chief difference is that the coefficient u_k for x^k in polynomial arithmethic bears no essential relation to its neighboring coefficients $u_{k\pm1}$, so the idea of carrying from one place to the next is absent."*

Der rechenintensivere Teil des Algorithmus besteht aus der Überprüfung der Gleichungen 3.6 und 3.7. Dadurch konzentrieren sich die Maßnahmen zur Beschleunigung auf diesen Teil des Algorithmus. Beide Vermutungen verwenden Polynome der Form:

$$u(x) = u_n x^n + ... + u_1 x + u_0, \tag{4.1}$$

wobei die Koeffizienten $u_n, ..., u_1, u_0 \in \mathbb{N}_0$. Ein Polynom dieser Form besitzt den Grad n, wenn $u_n \neq 0$. Um die Potenzberechnung der Polynome zu beschleunigen wurden unterschiedlichen Methoden untersucht. Grundsätzlich gilt folgende Beobachtung aus [Gor98]:

Bemerkung 2 (Äquivalenz zwei Probleme) *Beide Fragestellungen sind äquivalent:*

1. *Welche ist die kleinste Anzahl an Multiplikation, die notwendig sind, um g^n zu berechnen? Hierbei sind nur Operationen erlaubt, welche zwei bereits errechnete Potenzen multiplizieren.*
2. *Wie lang ist die kürzeste Additionskette für n?*

Definition 5 (Additionsketten aus [Gor98]) *Eine Additionskette für ein $n \in \mathbb{N}$ ist eine endliche, monoton steigende Folge positiver, ganzer Zahlen*

$$a_1 = 1, a_2, ..., a_r = n,$$

und für alle $i = 1, ..., r$ gibt es j und k mit $k \leqslant j < i$, so dass $a_i = a_j + a_k$. r wird als Länge der Additionskette bezeichnet.

Eine kurze Additionskette für n reduziert die Anzahl der Multiplikationen bei der Berechnung von g^n durch die Berechnung von $g^{a_2}, g^{a_3}, ..., g^{a_{r-1}}$. Die Bestimmung der kürzesten Additionskette einer Zahl ist ein **NP**-vollständiges Problem. Die optimale Lösung kann somit durch Approximationsalgorithmen nur genährt werden. Sei $l(n)$ die Länge der kürzesten Additionskette für n, dann gilt für relativ große n aus [Erd60] folgendes:

$$l(n) = log\ n + (1 + o(1)) \frac{log\ n}{log\ log\ n}$$

Der genaue Wert für $l(n)$ ist nur (wie in Abbildung 4.2 dargestellt) für relativ kleine n bekannt.

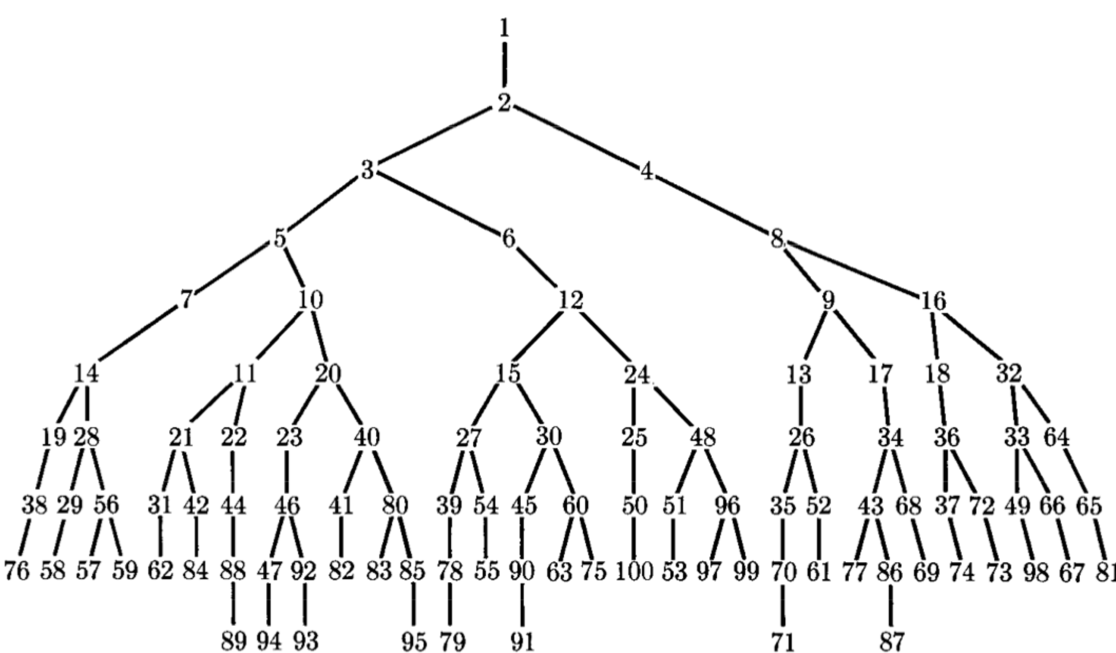

Abbildung 4.2.: Kürzeste Additionskette für $n < 101$ aus [Knu97]

Die obere Schranke dieser Funktion wird durch die *Square and multiply*-Methode aus [CF05] und [Knu97] beschrieben. Diese Methode soll anhand Algorithmus 4.2 verdeutlich werden. Sei

$$n = \sum_{i=0}^{l} c_i 2^i$$

die Darstellung der Zahl n zur Basis 2.

Algorithmus 4.2 Square and multiply-Methode

```
a ← 1
for d = l to 0 by −1 do
    a ← a × a
    if c_d = 1 then
        a ← a × x
    end if
end for
return a
```

Diese Methode basiert auf folgende Gleichung:

$$x^{(n_{l-1}\dots n_{i+1}n_i)_2} = \left(x^{(n_{l-1}\dots n_{i+1})_2}\right)^2 \times x^{n_i} \tag{4.2}$$

Die Methode aus Algorithmus 4.2 ist ebenso bekannt unter *left-to-right binary*-Methode, da die Berechnung vom *most* zu dem *least significant* Bit stattfindet. Die Anzahl der Quadrierungen ist in dieser Methode abhängig von der Länge von n ($\sim log\ n$). Die Anzahl der Multiplikationen hängt vom Hamming-Gewicht von n ab. Alle von Null verschiedenen Bits in der binären Darstellung von n sorgen für eine Multiplikation. Im Durchschnitt werden ½ $\log\ n$ Multiplikation benötigt. Diese Methode wird in dieser Arbeit für die Berechnung von $(X-1)^n$ aus Gleichung 3.6 und $(X+2)^n$ aus Gleichung 3.7 verwendet. Ein weiterer Vorteil dieser Methode in diesem Algorithmus beruht darauf, dass die Polynome schrittweise durch $(X-1)^n\ (mod\ X^r - 1, n)$ gekürzt werden. Dies sorgt dafür, dass der Speicherverbrauch konstant gering ist. Als Beispiel soll die Berechnung von x^{314} aus [CF05] in der Tabelle 4.2 dienen; $314 = (100111010)_2$ und $l = 9$.

i	8	7	6	5	4	3	2	1	0
n_i	1	0	0	1	1	1	0	1	0
y	x	x^2	x^4	x^9	x^{19}	x^{39}	x^{78}	x^{157}	x^{314}

Tabelle 4.2.: Square and multiply-Methode

Ausschnitte der für die x86-Architektur implementierten Methode zeigen die entsprechenden Quellcode 4.1 und Quellcode 4.2.

```
1  /*
2   * Implementation of the Square and multiply-Methode
3   *
4   * @param       a pointer to the result polynom,
5                  a pointer to a dummy polynom,
6                  the polynom that should be squared,
7                  the exponent,
8                  the r value
9   * @return      void
10  */
```

```
11 void polyExp_Mod_int(poly* resultPoly, poly* dummyPoly, poly* left, mpz_t exp, int r){
12
13     int k;
14     int myLimit;
15
16     polyCopy(resultPoly, left);
17     /* get the size of the exponent to base 2 */
18     myLimit = mpz_sizeinbase(exp, 2);
19
20     for(k=myLimit-2; k>-1; k--){
21         polyMult_Mod(dummyPoly, resultPoly, resultPoly, exp, r);
22
23         /* check if the k-th-bit equals 1 */
24         if(mpz_tstbit(exp,k)==1){
25             polyMult_Mod(resultPoly, dummyPoly, left, exp, r);
26         } else {
27             polyCopy(resultPoly, dummyPoly);
28         }
29     }
30 }
```

Quellcode 4.1: Square and multiply-Methode für Polynome

```
1  /*
2   * Calculate the remainder of polynom division
3   *
4   * @param       a pointer to the result polynom,
5                  a pointer to the dividend polynom,
6                  a pointer to the divisor polynom that,
7                  the exponent,
8                  the r value
9   * @return      void
10  */
11 void polyMult_Mod(poly* resultPoly, poly* left, poly* right, mpz_t exp, int r) {
12     unsigned int i, j, myR;
13     mpz_t dummy;
14
15     mpz_init(dummy);
16
17     /* set all coefficients from resultPoly = 0 */
18     for (i=0; i<= (resultPoly->degree-1); i++) {
19         mpz_set_ui(resultPoly->coeff[i], 0);
20     }
21
```

```
22    for (i=0; i<= (left->degree-1); i++) {
23        for (j=0; j<= (right->degree-1); j++) {
24            myR = (i+j)%r;
25
26            /* GMP : void mpz_addmul (mpz t rop, mpz t op1, mpz t op2) [Function] */
27            /* Set rop to rop + op1 x op2. */
28
29            mpz_addmul(resultPoly->coeff[myR], left->coeff[i], right->coeff[j]);
30            mpz_mod(resultPoly->coeff[myR], resultPoly->coeff[myR], exp);
31
32        }
33    }
34
35 }
```

Quellcode 4.2: Modulo-Rechnung für Polynome

An dieser Stelle seien folgende weitere Methoden, welche nicht implementiert wurden, kurz erwähnt:

Methoden basierend auf Addition-Subtraktionsketten Eine weitere Methode zur Verkürzung der Additionskette kann durch Verwendung eine weitere mathematische Operation erfolgen. Methoden[1], welche die Subtraktion zur Erzeugung einer Additionskette verwenden, führen bei einer Subtraktion der Potenzen zu einer Division der Polynome. Aufgrund der schlechten Laufzeiten einer Division im Vergleich zur Multiplikation kamen solcher Methoden für die implementierte Polynombibliothek nicht in Frage.

m-ary Methode aus [Gor98] Diese Methode stellt eine Verallgemeinerung der *Square and multiply*-Methode dar. Hierbei werden Zahlen > 2 als Basis verwendet. Diese Methode soll anhand Algorithmus 4.3 verdeutlicht werden. Hierbei soll

$$n = \sum_{i=0}^{l} c_i m^i$$

die Darstellung der Zahl n zur Basis m sein.

Diese Methode ist besonders effizient für Fälle $m = 2^k$, denn hierbei benötigt das

[1]z.B. die NAF-Methode aus [Gor98]

Algorithmus 4.3 m-ary Methode

Require: $x^2, x^3, \ldots, x^{m-1}$ vorgerechnet
  $a \leftarrow 1$
  for $d = l$ to 0 by -1 **do**
    $a \leftarrow a^m$
    $a \leftarrow a \times x^{c_d}$
  end for
  return a

Potenzieren von a zur m-ten Potenz nur k Quadrierungen.

Fenstermethoden Die *m-ary Methode* allgemein betrachtet verwendet ein k-Bit langes Fenster der binären Darstellung von x. Die Verallgemeinerung führt zur folgenden Überlegungen:

- Diese Fenster müssen nicht zwingend aufeinander folgen
- Aufeinander folgende Nullen müssen nicht berechnet werden
- Nur gerade Potenzen von x müssen vorgerechnet werden

Sei als Beispiel $x = 262359474289536631831 91$ aus [Gor98], dann hat x folgende binäre Darstellung:

$$1011000111001000000111010010100111010100000010111100000111110011001010101 11.$$

Unter Verwendung der Fenstermethode bei einer Länge von bis zu 4 Bits, würde die Berechnung von n^x 93 Multiplikation benötigen:

$$\underline{1011}_{11}000\underline{111}_{7}00\underline{1}_{1}000000\underline{111}_{7}0\underline{1001}_{9}0\underline{1001}_{9}\underline{1101}_{13}0\underline{1}_{1}000000\underline{1011}_{11}\underline{11}_{3}00000\underline{1111}_{15}\underline{1001}_{9}\underline{1001}_{9}0\underline{101}_{5}0\underline{111}_{7}$$

Diese Berechnung unter Verwendung der *m-ary Methode* bei $m = 8$ benötigt dagegen 102 Multiplikationen.

Einzelne Maßnahmen zur Beschleunigung des Algorithmus und architekturabhängige Besonderheiten sollen in den folgenden Unterkapiteln besprochen werden.

4.3.2. x86-Architektur

Die Implementierung des Algorithmus für diese Architektur erfolgte unter Verwendung der GMP-Bibliothek. Zum Umgang mit Polynomen wurde eine polynomarithmetische Bibliothek implementiert. Die Definition der Polynome werden wie in Quelcode 4.3 (unter Verwendung des Variablentyps der GMP-Bibliothek für Integer-Werte *mpz_t*) definiert.

```
1 typedef struct  {
2
3     unsigned int degree; /* this value is the real-poly-degree + 1
4       * for example: the poly x^5 has a degree value of 6
5       * in this struct
6       */
7     mpz_t * coeff; /* Array presenting the coefficients of the polynom */
8
9 } poly;
```

Quellcode 4.3: Polynom-Definiton

Weiterhin wurden folgende Funktionen entwickelt:

void polyPrint(poly* in) Zur Ausgabe der Polynome zu Testzwecken.

void polyClear(poly* inPoly) Zur Freigabe der allokierten *mpz_t*-Variablen mittels der GMP-Funktion *mpz_clear*.

void polyInit(poly* inPoly) Zur Initialisierung der *mpz_t*-Variablen mittels der GMP-Funktion *mpz_init*.

void polyZero(poly* inPoly) Hierdurch werden alle Koeffizienten eines Polynoms gleich Null gesetzt.

void polyCopy(poly* resultPoly, poly* inPoly) Zum Kopieren von Polynomen.

int polyIsEqual(poly* left, poly* right, mpz_t a) Zum Vergleich zweier Polynome im Ring $\mathbb{Z}_a$.

void polyExp_Mod_int(poly* resultPoly, poly* dummyPoly, poly* left, mpz_t exp, int r) Zur Potenzberechnung der Polynome unter Verwendung der *Square and Multiply*-Methode.

void polyMult_Mod(poly* resultPoly, poly* left, poly* right, mpz_t exp, int r)
Zur Modulo-Berechnung der Polynome im Ring $\mathbb{Z}_{exp}$

Der Unterschied zwischen der Windows- und der Linux-Version ist sehr gering und wurde mittels Präprozessor-Befehlen verwirklicht.

4.3.3. Cell-BE-Architektur

Bedingt durch den offiziell für die Cell-BE-Architektur nicht vorhandenen BOINC-Client wurde ein Algorithmus für die Cell-BE-Architektur implementiert, welcher BOINC zunächst nicht unterstützt. Die Implementierung auf solch einer speziellen Architektur verlangt ein *bodenständiges* Programmieren. Entscheidungen zur Implementierung wurden basierend auf der technischen Dokumentation des Cell-Prozessors von IBM [SG07] und [SG06] getroffen. Als Parallelisierungsmethode des Algorithmus wurden folgende Modelle in Betracht gezogen:

Multistage Pipeline Modell erfolgt durch eine stufenweise Berechnung (Beispiel: SPE1 berechnet Werte, die von SPE2 verwendet werden)

Parallel Modell alle SPEs arbeiten unabhängig voneinander auf Partitionen der Daten

Abbildung 4.3 stellt diese Modelle dar.

Die PPE dient in alle Modellen als Koordinator, somit werden diese als *PPE Centric* bezeichnet. Weiterhin sei an dieser Stelle das *Service Modell* erwähnt. In diesem Modell laufen auf den SPEs verschiedene Algorithmen. Eine Kombination der beschriebenen Modelle ist ebenso möglich. Der in dieser Arbeit implementierte Algorithmus verwendet das Parallel Modell als Parallelisierungsmethode. Der Ablauf kann anhand Algorithmus 4.4 und Algorithmus 4.5 zusammengefasst werden:

Kommunikation zwischen PPE und SPEs Die Kommunikation zwischen PPE und SPEs erfolgte in dieser Implementierung durch das Mailbox-System. Jede Recheneinheit besitzt einen Maileingang und -ausgang. Mittels Mailboxes ist es möglich 32-bit Nachrichten auszutauschen, die auch unterbrechenden Charakter haben können (Interrupt-Mails).

Algorithmus 4.4 PPE-Algorithmus

```
convert n
create SPE-context
load SPE-program to context
send n to all SPEs
while r ⩽ √2 do
   if r ∤ n then
      if n^2 ≢ 1(mod r) then
         read mailbox
         if message == PRIME then
            break
         else
            send r to SPEn
         end if
         r ← mpz_nextprime(r)
      end if
   end if
end while
send EXIT_MESSAGE to all SPEs
```

Algorithmus 4.5 SPE-Algorithmus

```
get n
intialize polynoms
r ← read mailbox
while r ≠ EXIT_MESSAGE do
   if (x + 2)^n ≡ x^n + 2( mod n, x^r − 1) then
      POPPRIME ← true
   end if
   if (x − 1)^n ≡ x^n − 1( mod n, x^r − 1) then
      AKSPRIME ← true
   end if
   if ((AKSPRIME = true) ‖ ( POPPRIME = true)) then
      send AKSPRIME und POPPRIME to PPE
   end if
end while
```

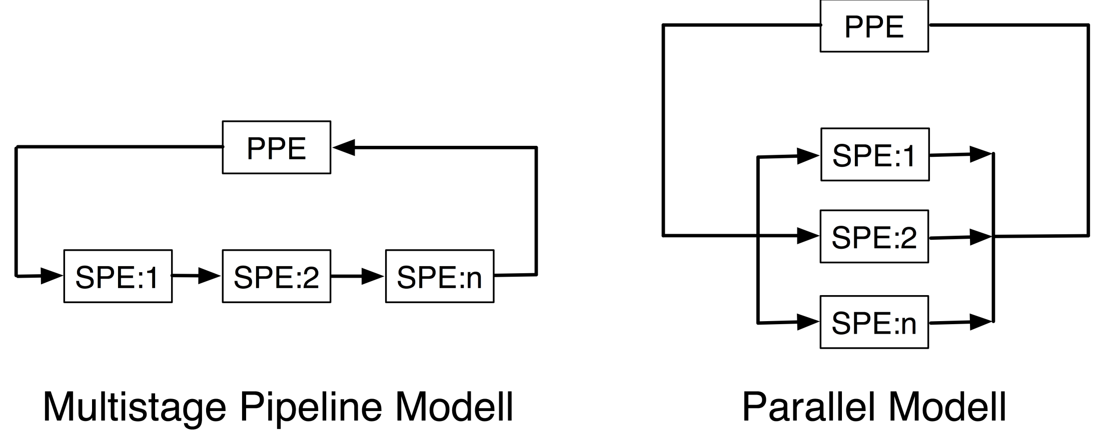

Abbildung 4.3.: Parallelisierungsmodelle nach [SG06]

Der prinzipielle Nachrichtenablauf des Algorithmus soll anhand Abbildung 4.4 erläutert werden.

1. zunächst werden die SPE-Kontexte mit der Übergabe der zu prüfende Zahl n als Parameter erzeugt
2. anschließend werden über Mailbox die r-Werte zur Berechnung übergeben
3. für den Fall, dass n durch eine SPE als Primzahl erkannt wurde, wird eine Nachricht über Mailbox an die PPE gesendet
4. die PPE sendet an alle SPEs über Mailbox eine *EXIT_MESSAGE*

Die gesamte Kommunikation erfolgt asynchron. Für den Fall einer leeren Nachrichtenschlange blockieren die SPEs.

Multi Precision Math Library Die *Multi Precision Math Library* (kurz MPM) ist eine Programmierbibliothek, welche arithmetische Funktionen auf der SPE für beliebig große Zahlen ermöglicht. Die Struktur dieser Bibliothek ist identisch mit der Struktur der GMP-Bibliothek. Sie enthält die Basis Operatoren wie *add, sub, mult, mod, div, cmp* sowie auch Hilfsoperationen wie *sizeof, neg*. Die interne Darstellung von großen Zahlen erfolgt anhand von Arrays von Vektoren vom Typ *unsigned int*. Diese Datenstruktur ermöglicht bei einfacher Genauigkeit SIMD[1]-Anweisungen. Tabelle 4.3 zeigt beispielhaft die interne

[1]Single instruction, multiple data

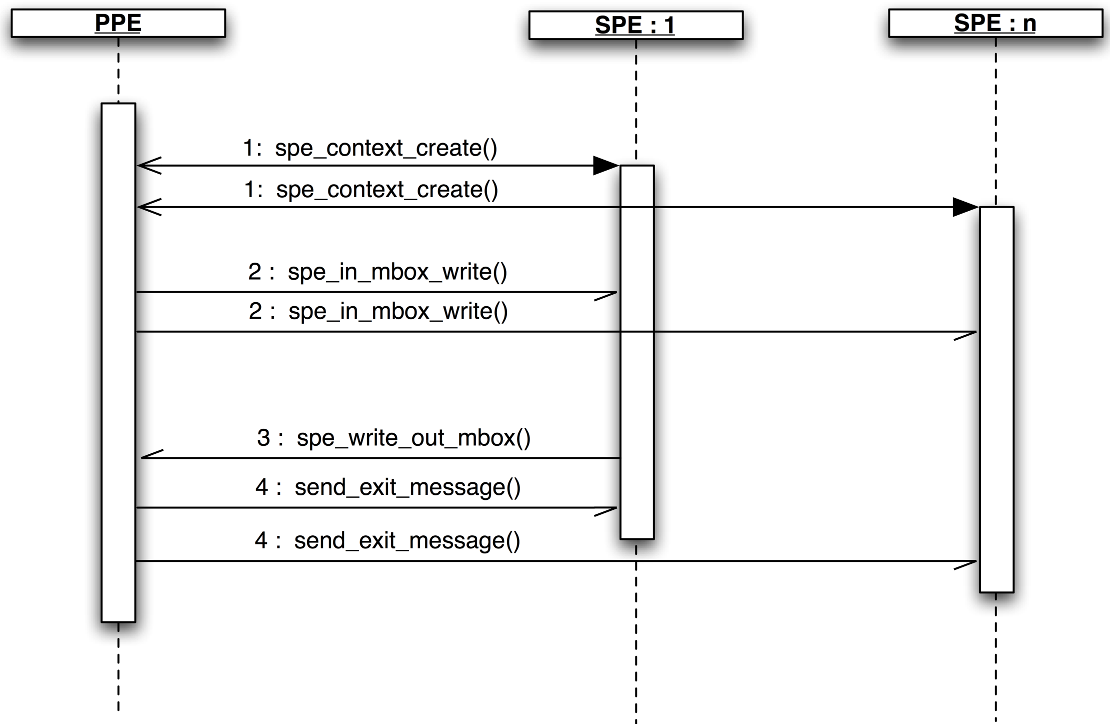

Abbildung 4.4.: Cell-BE Nachrichtenverlauf

Darstellung der Zahl 8589934595. Diese Darstellung entspricht:

$$2 \times 2^{32} + 3 \times 2^{0}$$

Die einzelnen Blöcke sind 32 Bit lang, somit der blockierte Speicher 128 Bit lang. In

0	0	2	3

Tabelle 4.3.: Interne Darstellung einer Zahl mittels MPM

der Implementierung sieht die Instanziierung einer solchen Zahl wie in Quellcode 4.4 dargestellt.

```
vector unsigned int a = (vector unsigned int){0, 0, 2, 3};
```

Quellcode 4.4: Verwendung von *vector unsigned int*

Sowohl PPE als SPEs setzen auf *In-Order-Execution*[1], dadurch werden Sprünge nicht in Hardware vorhergesagt. Mit Hilfe von *Branch Hints* können hohe Kosten für *Branch Misses* verhindert werden. *Branch Hints* wurden in dieser Implementierung lediglich auf den SPE-Anwendungen, wie in Quellcode 4.5 ersichtlich, eingesetzt.

```
1        /*
2         * check if (x + a)^n mod (x^r - 1) != x^(n mod r) + a
3         */
4        if (__builtin_expect(polyIsEqual(resultPoly, &rightPoly_coeffmodr,
5                &rightPoly_coeff0) == 1, 1)) {
6            popPrime=1;
7        }
```

Quellcode 4.5: Verwendung von *Branch Hints*

4.4. BOINC

Die projektspezifischen Server-Dämonprozesse und die unterschiedlichen BOINC-Versionen wurden basierend auf Quellcodebeispielen implementiert. Wesentliche Parameter, welche relevante Projekteigenschaften festlegen, sind oft nicht ausführlich genug dokumentiert. Projekteigenschaften werden sowohl im Quellcode als auch anhand XML-Konfigurationsdateien konfiguriert. Abbildung 4.5 stellt die Startseite des implementierten Projekts[2] dar.

4.4.1. Das BOINC-Application Programming Interface

Das *BOINC Application Programming Interface* (kurz BOINC-API) ist eine Sammlung in C++ implementierter Funktionen. Alle relevanten Funktionen haben ein C Interface, so dass eine Client-Implementierung in C (oder in anderen Programmiersprachen) ebenso möglich ist. Diese Funktionen sind nicht betriebssystemabhängig, was die Implementierung unterschiedlicher Versionen sehr erleichtert. Die API wird sowohl bei der Client-

[1] Befehle werden strikt nach Programmreihenfolge abgearbeitet
[2] www.primaboinca.com

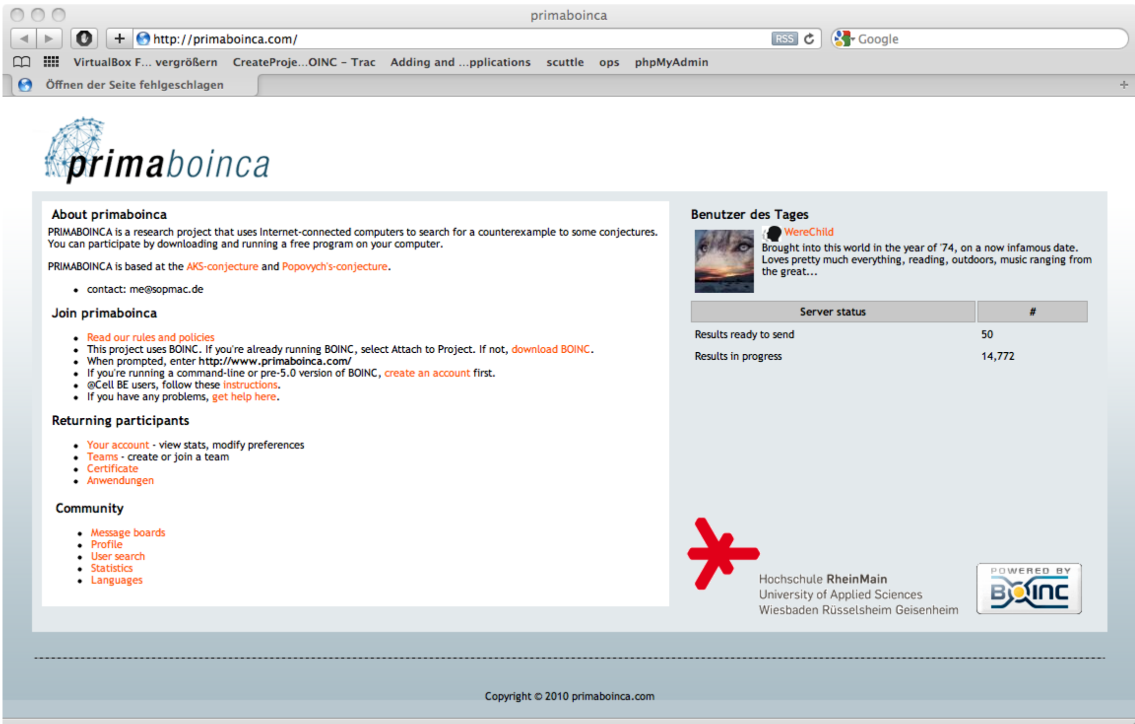

Abbildung 4.5.: www.primaboinca.com

Implementierung als auch bei der Implementierung der Server-Dämonprozesse verwendet. Im folgenden werden die verwendeten Funktionen und Klassen aus dieser API aufgelistet und erläutert.

boinc_init() Initialisiert eine Single-thread-BOINC-Anwendung

boinc_finish(int status) Beendet einer BOINC-Anwendung, wobei $status \neq 0$, wenn ein Fehler aufgetreten ist.

boinc_resolve_filename(char *logical_name, char *physical_name, int len) Ersetzt den logischen durch den physikalischen Dateinamen.

boinc_fopen(char* path, char* mode) BOINC-Anwendungen sollen diese betriebssystemunabhängige Methode anstelle von *fopen()* verwenden.

boinc_time_to_checkpoint() Anwendungen, welche pro Arbeitspaket lange Rechenzeit benötigen, sollten vorhandene Teilergebnisse in Form einer Datei (engl. state file) periodisch sichern. Diese Datei sollte alle Informationen beinhalten, welche für das Fortsetzen der Berechnung im Falle eines Neustarts notwendig sind. Bei einem

Start überprüft die Anwendung den Inhalt dieser Datei und legt somit fest, welcher Anteil eines Arbeitspaket bereits berechnet wurde.

boinc_fraction_done() Teilt dem Core-Client mit, welcher Anteil in Prozent eines Arbeitspaket abgearbeitet wurde.

class MFILE Diese Klasse sollte verwendet werden, um eine atomare[1] Sicherung der Teilergebnisse zu ermöglichen. Die *MFILE*-Klasse puffert die Daten im Speicher und schreibt diese beim Aufruf von *flush()* auf die Festplatte.

4.4.2. Definition der Arbeitspakete

Die Arbeitspakete beinhalten jeweils 20000 ungerade Zahlen aus dem Zahlenbereich $n > 10^{10}$, dies führt zu einer Paketgröße von 235 kB. Die Entscheidung für diese Paketgröße erfolgte basierend auf zwei Überlegungen:

- Die Zeit für die Berechnung eines Arbeitspaket muss in Relation zur Deadline (maximale Zeit für eine Berechnung) stehen.
- Die Größe der errechneten Arbeitspakete soll < 1 MB sein.

Die Deadline wurde für dieses Projekts zunächst auf 24 Stunden gesetzt. Bei einer durchschnittlichen Berechnungszeit von 4,4 Zahlen pro Sekunde (pro CPU-Kern) werden 20000 Zahlen (ein Paket) in etwa 1,5 Stunden gerechnet. Die errechneten Arbeitspakete erreichen eine Größe von etwa 520 kB.
Das Format der errechneten Arbeitspakete wurde wie folgt festgelegt:

<Überprüfte Zahl>,<r>,<AKS-prime>,<Popovych-prime>,<MR-prime>,<Zeit>

Die einzelnen Werte innerhalb einer solchen Datei haben folgende Bedeutung:

<Überprüfte Zahl> Zahl, welche überprüft wurde

<r> ist entweder:

- der Teiler einer zusammengesetzten Zahl

[1] ein Sachverhalt, der nicht weiter in seine Bestandteile zerlegbar ist.

- -1, falls kein Teiler < 100 existiert
- Zahl, welche dazu führt, dass eine/beide Vermutungen zutreffen

<AKS-prime> hat den Wert 1 für den Fall, dass die überprüfte Zahl laut AKS-Vermutung prim ist, 0 sonst

<Popovych-prime> hat den Wert 1 für den Fall, dass die überprüfte Zahl laut Popovychs-Vermutung prim ist, 0 sonst

<MR-prime> hat den Wert 2, wenn nach der *mpz_probab_prime_p*-Methode die überprüfte Zahl sicher prim ist; 1 wenn die überprüfte Zahl wahrscheinlich prim ist und 0 wenn es sich bei der überprüften Zahl sicher um eine zusammengesetzte Zahl handelt

<Zeit> für die Berechnungen dieser Zahl benötigte Zeit in Sekunden

Es folgt ein kleiner Ausschnitt eines Arbeitspakets.

```
# linux output file #
42946107285,3,0,0,0,0.000063
42946107287,5,1,1,1,0.000444
42946107289,-1,0,0,0,0.726943
42946107291,3,0,0,0,0.000017
42946107293,5,1,1,1,0.000398
42946107295,5,0,0,0,0.000006
42946107297,3,0,0,0,0.000007
42946107299,-1,0,0,0,0.721519
42946107301,-1,0,0,0,0.717713
42946107303,3,0,0,0,0.000021
42946107305,5,0,0,0,0.000006
```

Die Überprüfung, ob ein Gegenbeispiel gefunden wurde, lässt sich bedingt durch das gewählte Format leicht durchführen. Mittels Ausführung von *grep*-Befehlen sind folgende Überprüfungen möglich:

grep ,0,0,1, * sucht nach Zahlen, welche laut Miller-Rabin prim (wahrscheinlich) sind, aber laut AKS-Vermutung und Popovychs-Vermutung nicht

grep ,0,0,2, * sucht nach Zahlen, welche laut Miller-Rabin prim (sicher) sind, aber laut AKS-Vermutung und Popovychs-Vermutung nicht

grep ,0,1,0, * sucht nach Zahlen, welche laut Popovychs-Vermutung prim sind, aber laut AKS-Vermutung und Miller-Rabin nicht

grep ,0,1,1, * sucht nach Zahlen, welche laut Popovychs-Vermutung und Miller-Rabin (wahrscheinlich) prim sind, aber laut AKS-Vermutung nicht

grep ,0,1,2, * sucht nach Zahlen, welche laut Popovychs-Vermutung und Miller-Rabin (sicher) prim sind, aber laut AKS-Vermutung nicht

grep ,1,0,0, * sucht nach Zahlen, welche laut AKS-Vermutung prim sind, aber laut Popovychs-Vermutung und Miller-Rabin nicht

grep ,1,0,1, * sucht nach Zahlen, welche laut AKS-Vermutung und Miller-Rabin (wahrscheinlich) prim sind, aber laut Popovychs-Vermutung nicht

grep ,1,0,2, * sucht nach Zahlen, welche laut AKS-Vermutung und Miller-Rabin (sicher) prim sind, aber laut Popovychs-Vermutung nicht

grep ,1,1,0, * sucht nach Zahlen, welche laut AKS-Vermutung und Popovychs-Vermutung prim sind, aber laut Miller-Rabin nicht

grep ,-1,0,0,1, * sucht nach Zahlen, für die kein $r < 100$ gefunden wurde, die aber laut Miller-Rabin (wahrscheinlich) prim sind

grep ,-1,0,0,2, * sucht nach Zahlen, für die kein $r < 100$ gefunden wurde, die aber laut Miller-Rabin (sicher) prim sind

4.4.3. Windows- und Linux-Applikation

Der Unterschied zwischen der Windows- und der Linux-Version ist sehr gering und wurde, ebenso wie in der von BOINC unabhängigen Version des Algorithmus, mit-

tels Präprozessor-Befehle verwirklicht. Algorithmus 4.6 zeigt den Aufbau des für das BOINC-Projekt implementierten Algorithmus.

Algorithmus 4.6 Windows und Linux BOINC-Algorithmus

```
boinc_init()
boinc_resolve_filename(INPUT_FILE, INPUT_FILE_PATH)
boinc_fopen(INPUT_FILE_PATH)
boinc_resolve_filename(OUTPUT_FILE, OUTPUT_FILE_PATH)
boinc_resolve_filename(CHECKPOINT_FILE, CHECKPOINT_FILE_PATH)
if CHECKPOINT_FILE vorhanden then
  fopen(CHECKPOINT_FILE, append)
else
  fopen(CHECKPOINT_FILE, write)
end if
while fgets(INPUT_FILE_PATH) != EOF) do
  n ← fgets(INPUT_FILE_PATH)
  r ← 2
  while r < 100 do
    if r ∤ n then
      if n^2 ≢ 1(mod r) then
        if (x + 2)^n ≡ x^n + 2( mod n, x^r − 1) then
          POPPRIME ← true
        end if
        if (x − 1)^n ≡ x^n − 1( mod n, x^r − 1) then
          AKSPRIME ← true
        end if
        if ((AKSPRIME = true) ‖ ( POPPRIME = true)) then
          break
        end if
      end if
      r ← mpz_nextprime(r)
    else
      break
    end if
  end while
  isPrime ← mpz_probab_prime(n, 10)
  print(CHECKPOINT_FILE, r, AKSPRIME, POPPRIME, isPrime)
  if boinc_time_to_checkpoint() then
    do_checkpoint()
  end if
end while
boinc_finish(0);
```

Der Algorithmus wird zunächst anhand BOINC-API-Funktionen initialisiert. Anschließend überprüft der Algorithmus den Status der Checkpoint-Datei. Die Hauptberechnung erfolgt, solange Zahlen aus einer Input-Datei (Arbeitspaket) gelesen werden. Nach jeder Berechnung werden die Ergebnisse in die Checkpoint-Datei geschrieben. Das Schreiben in diese Datei erfolgt atomar anhand der Funktion *flush()* der verwendeten Klasse *MFILE*.

4.4.4. Cell BE-Applikation

Der für Cell-BE implementierte Algorithmus unterscheidet sich von der x86-Version insofern, dass die eigentliche Berechnung durch einen separaten Prozess ausgeführt wird. Die implementierte BOINC-Anwendung für diese Architektur liest ebenso wie die x86-Version die zu berechnende Zahl aus der Input-Datei und startet mit der gelesenen Zahl als Argument den Prozess zur Berechnung. Die Kommunikation zwischen der BOINC-Anwendung und dem Berechnungsalgorithmus erfolgt über *Pipes*[1]. Ein Ausschnitt aus dem Quellcode 4.6 soll diesen Sachverhalt verdeutlichen.

```
1     ...
2
3     /* get the numbers to be proofed */
4     while(fgets(linebuf, sizeof(linebuf), infile)!= NULL) {
5
6    char command[1024]="conjecture";
7    strcat(command, " SPU_NUM ");
8    strcat(command, linebuf);
9
10   /* start conjecture command using pipes */
11   if ( !(fpipe = (FILE*)popen(command,"r")) )
12   {
13    (void) out.printf("# problems with the pipe #\n");
14     fprintf(stderr, "APP: pipe error\n");
15     exit(1);
16   }
17
18   /* read the results out of the pipe */
19   while ( fgets( line, sizeof(line), fpipe)!=NULL)
20   {
```

[1]bezeichnet einen gepufferten uni- oder bidirektionalen Datenstrom zwischen zwei Prozessen

```
21          /* write the results into the output file */
22          (void) out.printf("%s", line);
23      }
24      /* close the pipe */
25      pclose(fpipe);
26
27      ...
```

Quellcode 4.6: Kommunikation über Pipes

Als BOINC-Client wurde eine für Cell-BE angepasste Version aus [1] verwendet.

4.4.5. Projektspezifische Server-Prozesse

Die Implementierung der projektspezifischen Server-Prozesse erfolgte unter Linux.
Der Work-Generator-Prozess, welcher die Arbeitspakete erzeugt, läuft als Dämon und arbeitet wie folgt.

Überprüfungen Zunächst wird die Verbindung über die BOINC-API zur Datenbank (Zeile 1-7) aufgebaut. Anschließend wird die Existenz der Anwendung (Zeile 9-12) und Templates (14-17) überprüft. Templates sind XML-Dateien, welche die Eigenschaften der Input- und Output-Dateien einer Anwendung festlegen.

```
1   retval = boinc_db.open(config.db_name, config.db_host, config.db_user,
2           config.db_passwd);
3
4   if (retval) {
5       log_messages.printf(MSG_CRITICAL, "can't open db\n");
6       exit(1);
7   }
8
9   if (app.lookup("where name='primaboinca'")) {
10      log_messages.printf(MSG_CRITICAL, "can't find app\n");
11      exit(1);
12  }
13
14  if (read_file_malloc("templates/uc_wu", wu_template)) {
15      log_messages.printf(MSG_CRITICAL, "can't read WU template\n");
```

[1]http://www.dotsch.de/boinc/BOINC Clients.html

```
16        exit(1);
17    }
```

Quellcode 4.7: BOINC-Überprüfungen

Main-Loop Der Main-Loop besteht aus einer Endlosschleife. In dieser Schleife wird zunächst überprüft, ob die notwendigen Server-Dämonprozesse (Zeile 5) laufen. Sollte anschließend die Anzahl der nicht gesendeten Arbeitspakete eine bestimmte voreingestellte Untergrenze erreichen (Zeile 8-11), so werden neue Arbeitspakete erzeugt (Zeile 15-24).

```
1  void main_loop() {
2      int retval;
3
4      while (1) {
5          check_stop_daemons();
6          int n;
7          /* count the unsent results */
8          retval = count_unsent_results(n, 0);
9          log_messages.printf(MSG_DEBUG, "unsent results = %d\n",n);
10         /* if enough ... sleep */
11         if (n > CUSHION) {
12             sleep(60);
13         } else {
14             /* if not, make (CUSHION - n) jobs */
15             int njobs = (CUSHION - n) / REPLICATION_FACTOR;
16             log_messages.printf(MSG_DEBUG, "Making %d jobs\n", njobs);
17             for (int i = 0; i < njobs; i++) {
18                 retval = make_job();
19                 if (retval) {
20                     log_messages.printf(MSG_CRITICAL, "can't make job: %d\n",
21                             retval);
22                     exit(retval);
23                 }
24             }
25             /* Now sleep for a few seconds to let the transitioner
26              * create instances for the jobs we just created.
27              * Otherwise we could end up creating an excess of jobs.
28              */
29             sleep(5);
30         }
31     }
32 }
```

Quellcode 4.8: BOINC-Main

Make_Job Diese Prozedur ist für Erzeugung der Input-Dateien und Verweise auf diese in der Datenbank verantwortlich. Die Dateien werden zunächst im Server-Download-Verzeichnis erzeugt (Zeile 1-26) und anschließend anhand von BOINC-API-Funktionen in die Datenbank eingetragen (Zeile 30-47).

```
1   /* make a unique name (for the job and its input file) */
2   (void) gmp_sprintf(myN, "%Zd", seqNo);
3
4   sprintf(name, "uc_%d_%s", start_time, myN);
5
6   /* Create the input file.
7    * Put it at the right place in the download dir hierarchy
8    */
9   retval = config.download_path(name, path);
10  if (retval)
11      return retval;
12  FILE* f = fopen(path, "w");
13  if (!f)
14      return ERR_FOPEN;
15
16  fprintf(f, "### primaboinca - input file ###\n");
17
18  for (k = 0; k <= counter; k++) {
19      (void) gmp_sprintf(myN, "%Zd", seqNo);
20      mpz_add_ui(seqNo, seqNo, 2);
21      fprintf(f, "%s\n", myN);
22  }
23  fclose(f);
24
25
26  /* Fill in the job parameters */
27  wu.clear();
28  wu.appid = app.id;
29  strcpy(wu.name, name);
30  wu.rsc_fpops_est = 1e12;
31  wu.rsc_fpops_bound = 1e14;
32  wu.rsc_memory_bound = 1e8;
33  wu.rsc_disk_bound = 1e8;
34  wu.delay_bound = 7*86400;
```

```
35      wu.min_quorum = REPLICATION_FACTOR;
36      wu.target_nresults = REPLICATION_FACTOR;
37      wu.max_error_results = REPLICATION_FACTOR * 4;
38      wu.max_total_results = REPLICATION_FACTOR * 8;
39      wu.max_success_results = REPLICATION_FACTOR * 4;
40      infiles[0] = name;
41
42      /* Register the job with BOINC */
43      return create_work(wu, wu_template, "templates/uc_result",
44              config.project_path("templates/uc_result"), infiles, 1, config);
```

Quellcode 4.9: BOINC-Make_Job

4.5. Algorithmus zu Lenstras Heuristik

Hendrik Lenstras und Carl Pomerances Heuristik [LP02] zur AKS-Vermutung legt nahe, dass es unendlich viele Gegenbeispiele geben müsste. Ein Algorithmus zur Berechnung von Zahlen, welche den Anforderungen des Satzes von Lenstra genügen, kann wie Algorithmus 4.7 formuliert werden.

Algorithmus 4.7 Implementierung von Lenstras Satz

$p_0 \leftarrow 3$
$p_1 \leftarrow 83$
$p_2 \leftarrow 163$
$p_3 \leftarrow 563$
$p_4 \leftarrow 564$
while $(p_j \pm 1) \nmid (n \pm 1)$ **do**
  $p_4 \leftarrow$ mpz_nextprime(p_4)
  if $p_4 \equiv 3 \bmod 80$ **then**
    $n \leftarrow p_0 * p_1 * p_2 * p_3 * p_4$
  end if
end while
output n

Hierbei werden für vier der fünf Primfaktoren der zu bestimmenden Zahl n die kleinsten Primzahlen p, für die gilt $p \equiv 3 \mod 80$ verwendet. Der letzte Primfaktor wird anhand der GMP-Funktion *mpz_nextprime(pj[4], pj[4]);* bestimmt. Möglicherweise wird durch

diesen Ansatz keine Zahl n gefunden, welche die weiteren Eigenschaften des Satzes 3 (siehe Unterkapitel 3) erfüllt, da die ersten vier Primfaktoren dieser Zahl in dieser Version des Algorithmus nicht variiert werden.

Quellcode 4.10 zeigt einen Ausschnitt des in C implementierten Algorithmus.

```
1     ...
2
3     /* endless ... */
4     while (1==1) {
5             /* endless ... */
6             while(1==1) {
7                 mpz_nextprime(pj[4], pj[4]);
8                 /* check if pj[4] = 3 mod 80 */
9                 if(((mpz_congruent_ui_p(pj[4], 3, 80))!=0)) {
10                        gmp_printf("\npj[4] = %Zd\n", pj[4]);
11                        mpz_add_ui(pj_plus1[4], pj[4], 1);
12                        mpz_sub_ui(pj_minus1[4], pj[4], 1);
13                        /* break to check ... */
14                        break;
15                    }
16            }
17
18            /* set n = pj[0] * pj[1] * pj[2] * pj[3] * pj[4] */
19            mpz_mul(n, pj[0], pj[1]);
20            mpz_mul(n, n, pj[2]);
21            mpz_mul(n, n, pj[3]);
22            mpz_mul(n, n, pj[4]);
23
24            gmp_printf("\nn = %Zd\n", n);
25
26            mpz_add_ui(n_plus1, n, 1);
27            mpz_sub_ui(n_minus1, n, 1);
28
29            /* check if for all pjs, pj + 1 divdes n + 1 */
30            for (j=0; j<5; j++) {
31                if (((mpz_divisible_p(n_plus1, pj_plus1[j]))!=0) &&
                      ((mpz_divisible_p(n_minus1, pj_minus1[j]))!=0)) {
32                    if(j==4) {
33                        for (i=0; i<5; i++) {
34                                gmp_printf("\npj[%d] = %Zd\n", j, pj[4]);
35                        }
36                        /* output the n found !!! */
37                        gmp_printf("\n\nFOUND: n = %Zd\n", n);
```

```
38                          return 0;
39                      }
40                  } else {
41                      break;
42                  }
43              }
44
45      }
46      ...
```

Quellcode 4.10: Lentras Heuristik

Basiert auf diesen Algorithmus wurde der Zahlenbereich $pj[4] = 43697931923$ und somit $n = 998518763145804963$ erfolglos durchsucht.
Eine weitere Implementierungsvariante zeigt Quellcode 4.11. In dieser Implementierung werden drei der fünf Primafaktoren der zu bestimmende Zahl n auf die kleinsten Primzahlen p gesetzt, für die gilt $p \equiv 3 \mod 80$. Der vierte Primfaktor wird zufällig errechnet und der letzte Primfaktor wird ebenso anhand der GMP-Funktion bestimmt. Eine weitere Variante wäre eine zufallsbedingte Bestimmung aller Primfaktoren.

```
1   ...
2   /* endless ... */
3   while (1==1) {
4           /* endless ... */
5           while(1==1) {
6               mpz_nextprime(pj[4], pj[4]);
7               /* check if pj[4] = 3 mod 80 */
8               if(((mpz_congruent_ui_p(pj[4], 3, 80))!=0)) {
9                       mpz_add_ui(pj_plus1[4], pj[4], 1);
10                      mpz_sub_ui(pj_minus1[4], pj[4], 1);
11                      /* Generate a random integer in the range 0 to pj[4] - 1 */
12                      mpz_urandomm(pj[3],state,pj[4]);
13                      while(1==1) {
14                          mpz_nextprime(pj[3], pj[3]);
15                          if(((mpz_congruent_ui_p(pj[3], 3, 80))!=0)) {
16                              mpz_add_ui(pj_plus1[3], pj[3], 1);
17                              mpz_sub_ui(pj_minus1[3], pj[3], 1);
18                              break;
19                          }
20                      }
21                      gmp_printf("\npj[4] = %Zd\npj[3] = %Zd\n", pj[4], pj[3]);
```

```
22                          /* break to check ... */
23                          break;
24                      }
25              }
26              /* set n = pj[0] * pj[1] * pj[2] * pj[3] * pj[4] */
27              mpz_mul(n, pj[0], pj[1]);
28              mpz_mul(n, n, pj[2]);
29              mpz_mul(n, n, pj[3]);
30              mpz_mul(n, n, pj[4]);
31
32              mpz_add_ui(n_plus1, n, 1);
33              mpz_sub_ui(n_minus1, n, 1);
34
35              /* check if for all pjs, pj + 1 divdes n + 1 and p - 1 | n - 1 */
36              for (j=0; j<5; j++) {
37                  if (((mpz_divisible_p(n_plus1, pj_plus1[j]))!=0) &&
                        ((mpz_divisible_p(n_minus1, pj_minus1[j]))!=0)) {
38                      if(j==4) {
39                          for (i=0; i<5; i++) {
40                              gmp_printf("\npj[%d] = %Zd\n", i, pj[i]);
41                          }
42                          /* output the n found !!! */
43                          gmp_printf("\n\n\n##################\nFOUND: n = %Zd\n", n);
44                          return 0;
45                      }
46                  } else {
47                      break;
48                  }
49              }
50
51      }
52      ...
```

Quellcode 4.11: Lentras Heuristik - Variante 2

KAPITEL 5

EVALUATION

Als x86-Client für die folgenden Messungen diente ein Intel Pentium Dual-Core E5200 2,50GHz (Wolfdale). Die Berechnungen fanden unter Ubuntu Version 10.04 (Lucid Lynx) statt. Weiterhin wurden Playstation 3-Konsolen von Sony und ein IBM BladeCenter samt sieben QS20 Blades als Testclients für die Cell-BE-Architektur verwendet.
Die in diesem Kapitel präsentierten Ergebnisse wurden anhand unterschiedlicher Methoden gewonnen. Im ersten Unterkapitel 5.1 wird der Worst Case des Algorithmus näher untersucht. Analysen mittels Profiler dienten ebenso wie gezielte Messungen (Zeit, Speicher) als Grundlagen für Maßnahmen zur Verbesserung des Algorithmus. Diese Maßnahmen und die dadurch erreichte Beschleunigung des Algorithmus werden im Unterkapitel 5.2 besprochen. Zur Analyse des BOINC-Projekts wurden gezielte SQL[1]-Abfragen an die BOINC-Server-Datenbank genutzt. Die Auswertung dieser Daten erfolgte für den Zeitraum vom 17.07. bis zum 03.08.2010. Diese Daten werden im Unterkapitel 5.3 präsentiert. Gerechnete, geschätzte bzw. hochgerechnete Werte dienen zum Vergleich bestimmter Algorithmen. Performance und Vergleiche zwischen der einzelnen Versionen der Anwendung werden im Unterkapitel 5.4 detailliert besprochen. Sonstige Evaluationen werden im Unterkapitel 5.5 präsentiert. Eine Online-Umfrage mittels *SurveyMonkey*[2] an die Teilnehmer des primaboinca.com-Projekts lieferte interessanten Ergebnissen in Bezug auf die Profile der Teilnehmer. Unterkapitel 5.6 zeigt abschließend einen repräsentativen Überblick über die Umfrageergebnisse.
Für die Messungen in diesem Kapitel (sofern nicht anders beschrieben) wurden zusammengesetzte Zahlen n verwendet, deren kleinster Primfaktor > 100 ist. Bei der Überprü-

[1] ist eine Datenbanksprache zur Definition, Abfrage und Manipulation von Daten in relationalen Datenbanken
[2] https://de.surveymonkey.com/

fung solcher Zahlen tritt der Worst Case des Algorithmus ein. Eine Begründung für diesen Sachverhalt ist im Unterkapitel 5.1 zu finden.

5.1. Worst Case

Der Worst Case beider Vermutungen tritt bei zusammengesetzte Zahlen ein, deren kleinster Teiler > 100 ist. Sei n die zu überprüfende Zahl, $r < 100$, und $r \nmid n$, dann werden bedingt durch den Aufbau des Algorithmus die Gleichungen

$$(X-1)^n = X^n - 1 \; (mod \; X^r - 1, n)$$

und

$$(X+2)^n = X^n + 2 \; (mod \; X^r - 1, n)$$

für alle Zahlen, für die gilt

$$n^2 \neq 1 (mod \; r) \tag{5.1}$$

überprüft. Die Gleichung 5.1 gilt recht oft. Der Grund dieses Sachverhalts kann wie folgt erklärt werden.

Bemerkung 3 (Selbstinverse Elemente in $\mathbb{Z}_r$)

$$n^2 \neq 1 \; (mod \; r) = n^2 - 1 = 0 \; (mod \; r)$$

daraus folgt

$$(n+1)(n-1) = 0 \; (mod \; r). \tag{5.2}$$

Die Gleichung 5.2 besitzt nur zwei Lösungen $n = 1$ und $n = -1$. Somit sinkt die Wahrscheinlichkeit mit steigendem r, dass die Gleichung 5.1 nicht gilt.

Korollar: Im Restklassenring $\mathbb{Z}_r$ sind nur die Elemente 1 und $r-1$ zu sich selbst invers.

Abbildung 5.1 zeigt die Anzahl der $r < 100$ und $r \in \mathbb{P}$, für die Gleichung 3.8 gilt, und zum Vergleich die Gesamtanzahl der Primzahlen < 100.

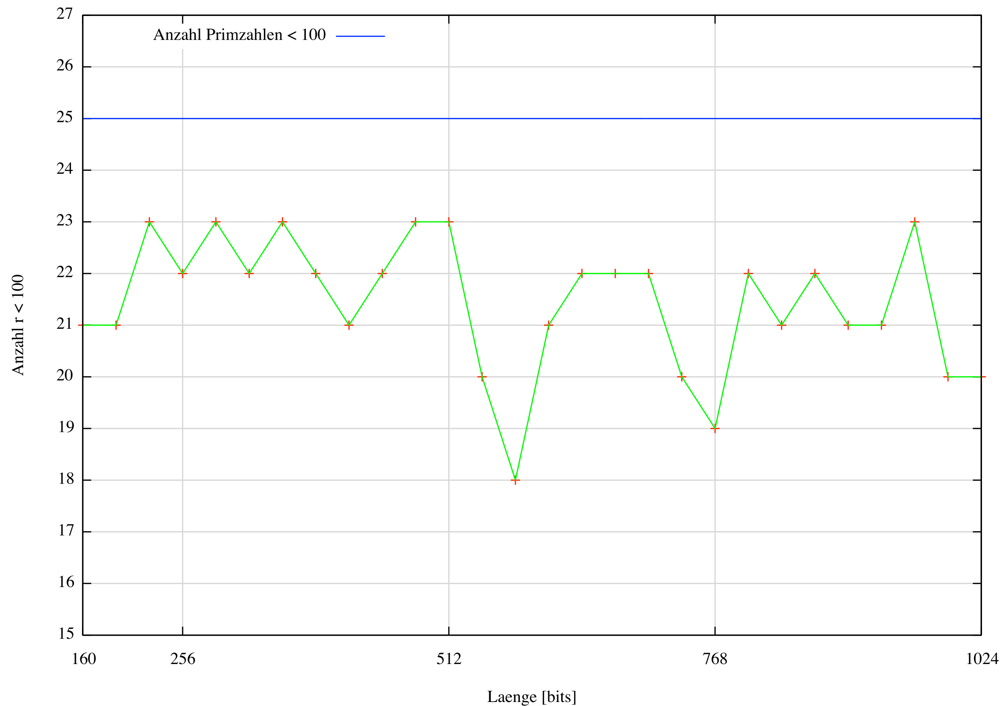

Abbildung 5.1.: Anzahl der passenden r-Werte bei zusammengesetzten Zahlen

Während die Anzahl der r-Werte, welche bei zusammengesetzten Zahlen die Gleichung 5.1 erfüllen, recht groß ist, werden Primzahlen bereits anhand relativ kleiner r-Werte als Primzahlen charakterisiert. Abbildung 5.2 zeigt für Zahlen eines Arbeitspakets die Verteilung der r-Werte, welche zur Identifizierung einer Zahl als Primzahl, geführt haben. In Abbildung 5.3 werden die r-Werte, welche zur Identifizierung einer Primzahl geführt haben, für Zahlen der Länge von 128 bis 1024 Bits dargestellt.

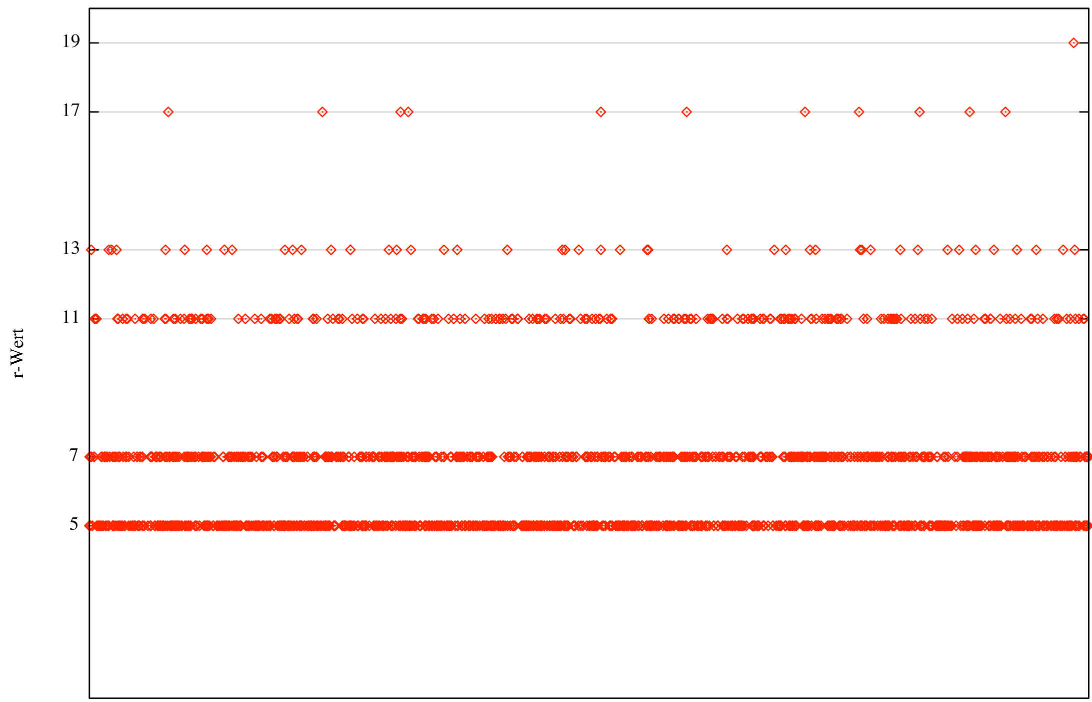

Abbildung 5.2.: Verteilung der r-Werte bei Primzahlen

Ein Vergleich zwischen Abbildung 5.4 und Abbildung 5.5 verdeutlicht die zeitliche Differenz bei der Überprüfung von Primzahlen und zusammengesetzten Zahlen, deren kleinster Primfaktor > 100 ist. Während die Charakterisierung einer Primzahl der Länge von 1024 Bits weniger als eine Sekunde benötigt, beträgt die benötigte Zeit bei zusammengesetzten Zahlen dieser Länge bereits etwa 1500 Sekunden.

5.2. Maßnahmen zur Performanceoptimierung

Anhand von Analysen mittels Profiler-Werkzeuge wurden Maßnahmen sowohl zur Beschleunigung des Algorithmus als auch zur Reduzierung des Speicherverbrauchs getroffen. Die Werkzeuge gprof und Shark dienten zur Lokalisierung der rechenintensiven Stellen im Algorithmus, während Valgrind zur Entdeckung von Speicherlecks und logischer Allokationsprobleme eingesetzt wurde.

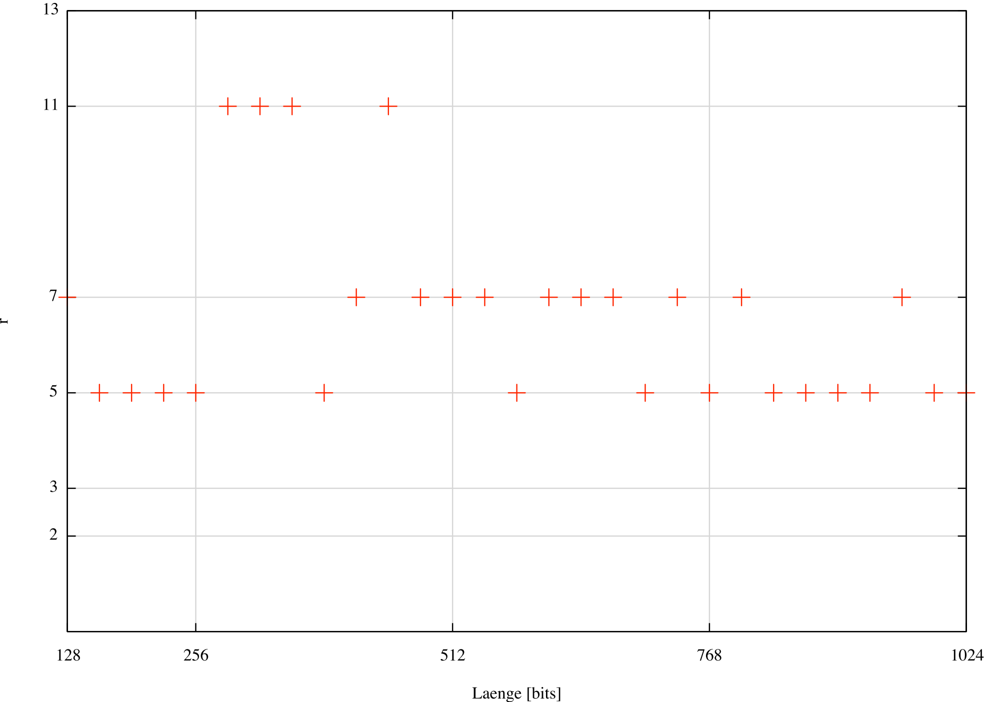

Abbildung 5.3.: Verteilung der r-Werte bei Primzahlen der Länge $128 - 1024$ Bits

Die am häufigsten aufgerufene und rechenintensivste Stelle im Algorithmus befindet sich in der Funktion *polyMult_Mod*, welche für die Modulo-Berechnung der Polynome verantwortlich ist. Diese Funktion ist Teil der in dieser Arbeit implementierten Polynom-Bibliothek. Abbildung 5.6 zeigt die prozentuale Zeitverteilung einzelner Anweisungen innerhalb der genannten Funktion. Ebenso sind in dieser Abbildung die laut Shark kritischen Anweisungen (gekennzeichnet durch ein Ausrufzeichen) und die mögliche Ursache zu sehen. Unterschiedliche Anpassungen führten von Quellcode 5.1 zu Quellcode 5.2.

```
1   ...
2   for (i=0; i<= (left->degree-1); i++) {
3       for (j=0; j<= (right->degree-1); j++) {
4           mpz_mul(dummy, left->coeff[i], right->coeff[j]);
5           mpz_mod(dummy, dummy, exp);
6           mpz_add(resultPoly->coeff[(i+j)%r], resultPoly->coeff[(i+j)%r], dummy);
7           mpz_mod(resultPoly->coeff[(i+j)%r], resultPoly->coeff[(i+j)%r], exp);
8       }
9   }
10  ...
```

Quellcode 5.1: Anfangsversion (polyMult_Mod)

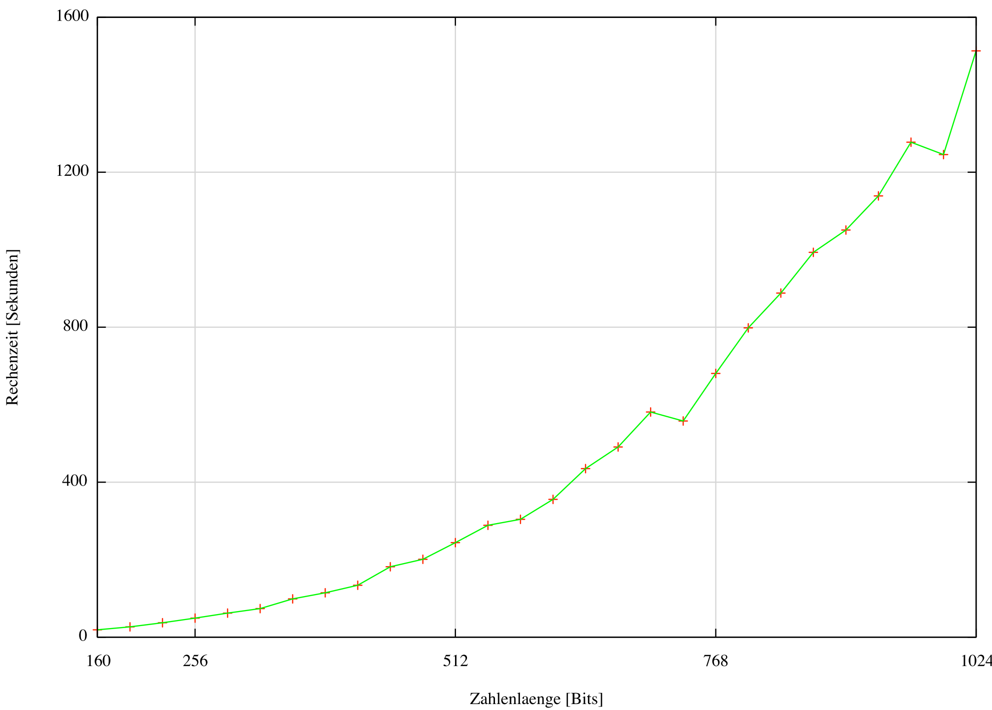

Abbildung 5.4.: Rechenzeit für zusammengesetzte Zahlen der Länge $128 - 1024$ Bits ohne Teiler < 100

```
1   ...
2
3   for (i=0; i<= (left->degree-1); i++) {
4       for (j=0; j<= (right->degree-1); j++) {
5           myR = (i+j)%r;
6
7           /* GMP : void mpz_addmul (mpz t rop, mpz t op1, mpz t op2) [Function] */
8           /* Set rop to rop + op1 x op2. */
9           mpz_addmul(resultPoly->coeff[myR], left->coeff[i], right->coeff[j]);
10          mpz_mod(resultPoly->coeff[myR], resultPoly->coeff[myR], exp);
11
12      }
13  }
14
15  ...
```

Quellcode 5.2: Aktuelle Version (polyMult_Mod)

Diese und weitere Anpassungen führten zu einer deutlichen Beschleunigung des Algorithmus. Der erreichte SpeedUp ist in Tabelle 5.1 ersichtlich. Durch die genannten Anpassun-

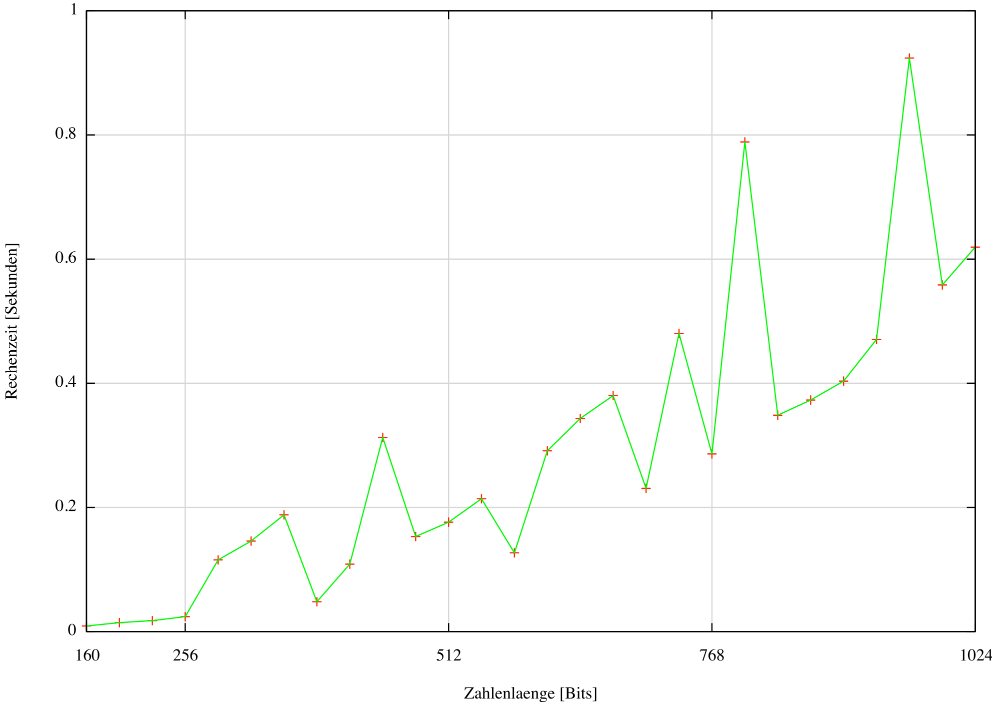

Abbildung 5.5.: Rechenzeit für Primzahlen der Länge $128 - 1024$ Bits

```
 0.1%  174  for (i=0; i<= (left->degree-1); i++) {
 5.9%  175    for (j=0; j<= (right->degree-1); j++) {
10.6%  176      mpz_mul(dummy, left->coeff[i], right->coeff[j]);
 4.7%  177      mpz_mod(dummy, dummy, exp);
40.8%  178      mpz_add(resultPoly->coeff[(i+j)%r], resultPoly->coeff[(i+j)%r], dummy);   ! Int div
37.5%  179      mpz_mod(resultPoly->coeff[(i+j)%r], resultPoly->coeff[(i+j)%r], exp);     ! Int div
       180
       181
       182    }
       183  }
```

Abbildung 5.6.: Ausschnitt einer Shark-Analyse

gen können im Primaboinca-Projekt bei einer Rechenleistung von etwa 430 GigaFLOPS[1] pro Tag etwa 26000000 Zahlen zusätzlich überprüft werden.

Weiterhin wurden Zahlen der Länge von 160 - 1024 Bits unter Anwendung beider Algorithmenversionen überprüft. Abbildung 5.7 zeigt die benötigte Rechenzeit. Während der Zeitgewinn bei Zahlen mit der Länge $\leqslant 512$ Bits wenige Sekunden beträgt, steigt die Zeitdifferenz bei Zahlen der Länge von 1024 Bits bereits auf über 500 Sekunden.

[1]Durchschnittliche Rechenleistung des Primaboinca-Projekts am 15.08.2010

	∅ RZ[a] pro Zahl	RZ[a] für 1024 bit	RZ[a] pro Pakete	Anz. Pakete pro Tag
Anfangsversion	0,3287	2097,45	6575,09	5200
Endversion	0,2240	1513,18	4480,60	7500[b]
Differenz	0,1047	583,82	2094,49	+1300

[a]Rechenzeit in Sekunden
[b]im Primaboinca-Projekt Stand 03.08.2010

Tabelle 5.1.: Zeitgewinn durch Code-Anpassung

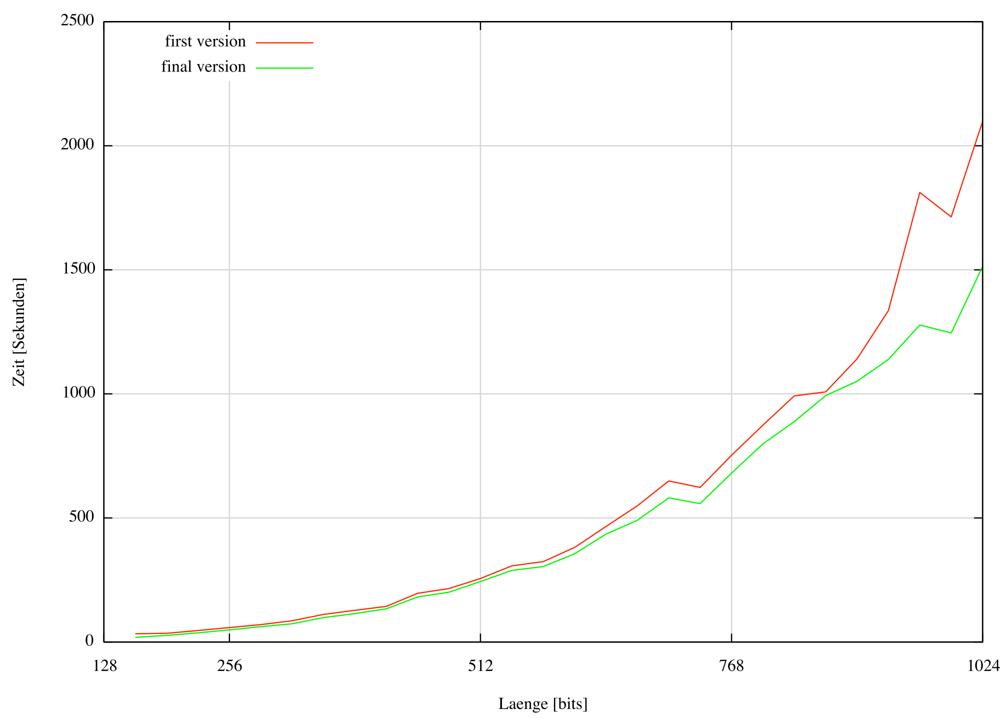

Abbildung 5.7.: Vergleich für Zahlen der Länge 160 - 1024 Bits

Unter Verwendung des *Memcheck*-Profiler aus der Valgrind-Werkzeugsammlung wurde der Speicherverbrauch des Algorithmus optimiert. Diese Anpassungen führten zu einem geringen und konstanten Speicherverbrauch. Der Speicherverbrauch des Algorithmus wurde für Zahlen der Länge von $128 - 1024$ Bits, wie in Abbildung 5.8 ersichtlich, gemessen. Während der virtuelle Speicherverbrauch konstant bei 1240 kB liegt, steigt der reale Speicherverbrauch gering von 612 auf 728 kB.

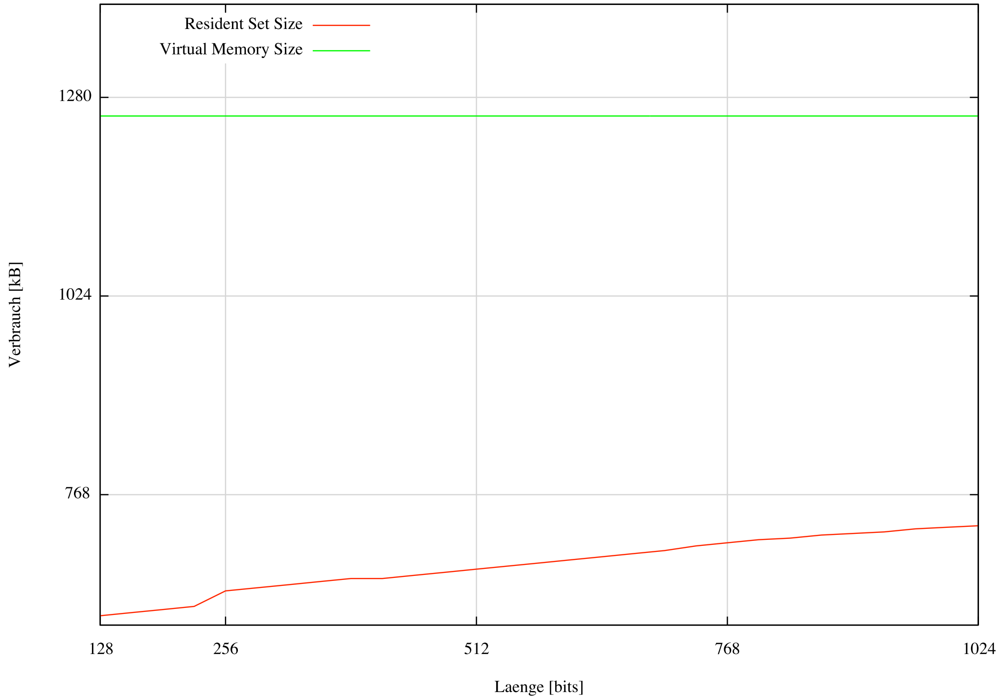

Abbildung 5.8.: Speicherbedarf bei Überprüfung von Zahlen der Länge $128 - 1024$ Bits

5.3. BOINC

Die Daten in diesem Unterkapitel wurden mittels Abfragen an die relationale BOINC-Server-Datenbank gewonnen.

Abbildung 5.9 stellt die Anzahl der täglich versendeten, gerechneten und abgebrochenen Arbeitspakete dar. Die Kurven wurden bewusst mittels kubischer Splines[1] gezeichnet, um das sinusähnliche Verhalten zu verdeutlichen. Alle drei Kurven sind sinusähnliche Kurven mit stetig steigender Amplitude. Die Frequenz dieser Sinuskurven lassen sich zeitlich (zum Beispiel als Wochenrhythmus) nicht eindeutig zuordnen.

[1]ist eine Funktion, welche durch gegebene Punkte im Koordinatensystem geht und eine minimale Gesamtkrümmung aufweist.

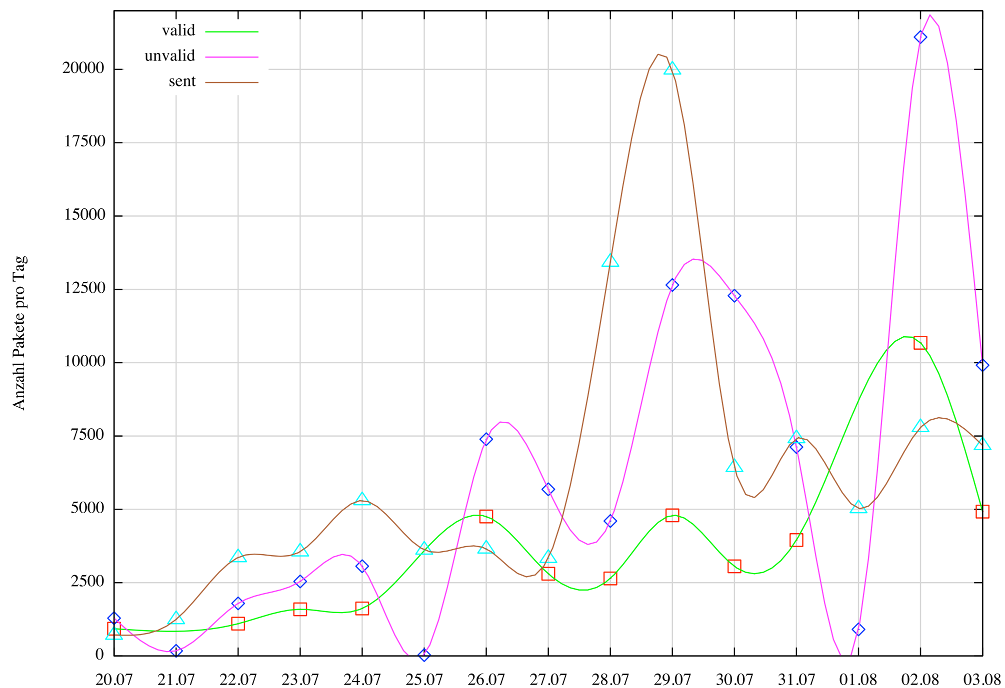

Abbildung 5.9.: Anzahl der Arbeitspakete am Tag

Die prozentuale Verteilung der Client-Betriebssysteme im primaboinca-Projekt in Abbildung 5.10 spiegelt die allgemeine Verteilung aus Abbildung 2.1 wider.

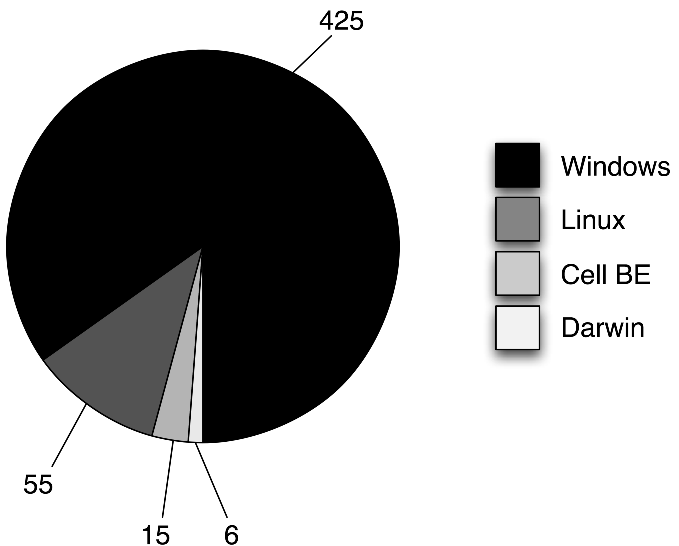

Abbildung 5.10.: Anzahl der Client-Architekturen/OS bei primaboinca (Stand 06.08.2010)

Abbildung 5.11 zeigt die weltweite Verteilung der am primaboinca-Projekt teilnehmenden Clients.

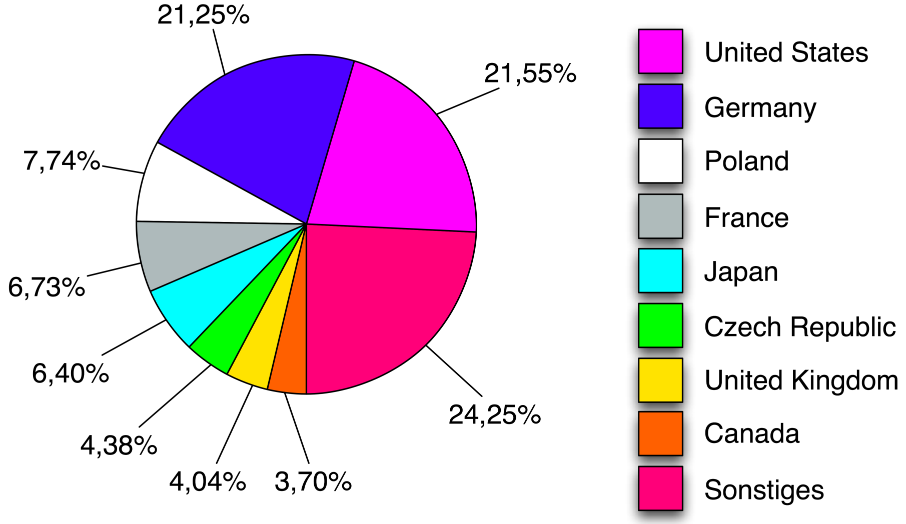

Abbildung 5.11.: Prozentuale weltweite Client-Verteilung (Stand 20.08.2010)

Obwohl die Differenz der teilnehmenden Clients zwischen den Vereinigten Staaten und Deutschland gering ist ($< 1\%$), beträgt die Differenz der Rechenleistung in Abbildung 5.12 etwa 20%. Obwohl Australien mit relativ wenigen Rechnern am Projekt beteiligt ist, erreichen diese Clients mehr als 6% der gesamten Rechenleistung.

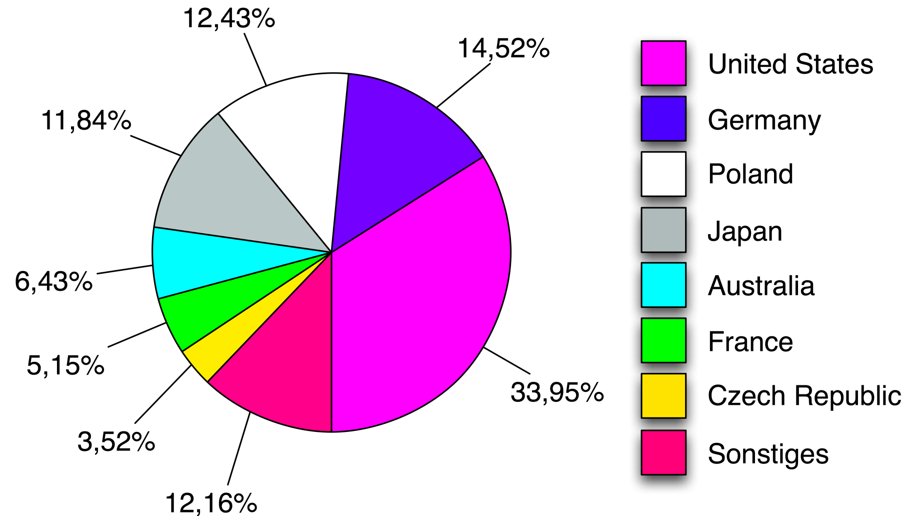

Abbildung 5.12.: Prozentuale weltweite Rechenleistungsverteilung (Stand 20.08.2010)

Der tägliche Zuwachs an Benutzer kann aus Abbildung 5.13 entnommen werden. Einrichtung eines Projektforums und Anpassungen bestimmter Projekteigenschaften führten zeitlich versetzt zu einem erhöhten Zuwachs.

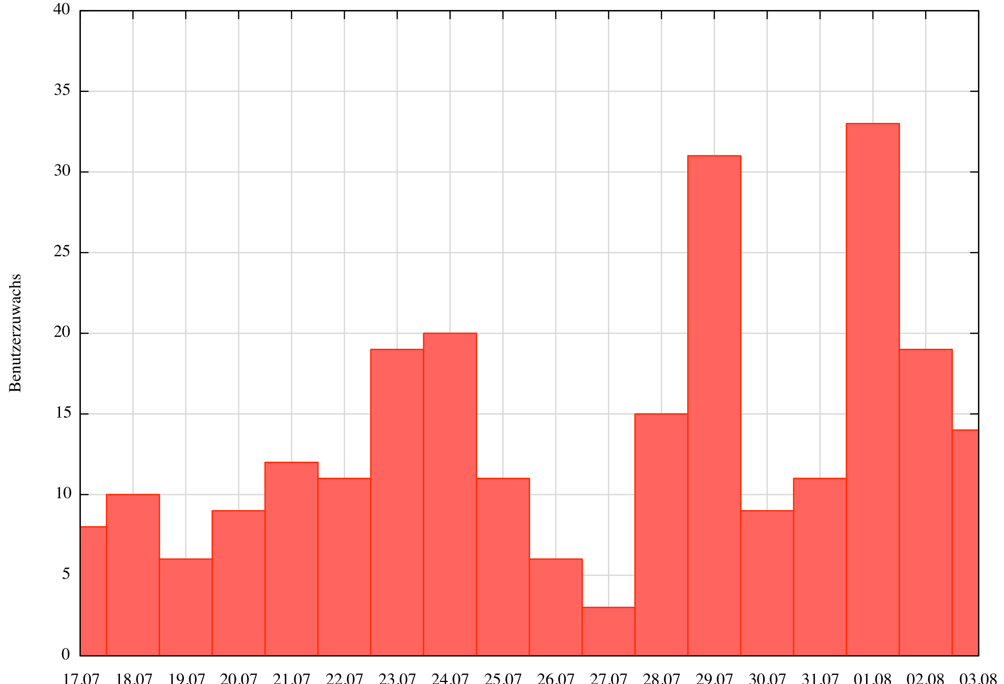

Abbildung 5.13.: Täglicher Teilnehmerzuwachs

Die Funktion, welche in Abbildung 5.14 dargestellt wird, verläuft nahezu linear (etwa 12,5 neue Benutzer pro Tag). Eine Hochrechnung (vom 02.08. auf den 20.08.2010) basierend auf diese Funktion entsprach mit geringer Abweichung der tatsächlichen Anzahl von 409 Benutzern.

Der tägliche Zuwachs an Rechnern wird in Abbildung 5.15 dargestellt. Die Relation zwischen Gesamtanzahl von Benutzern und Rechnern beträgt $1 : 2,23$. Aktuell (Stand 22.08.2010) wird primaboinca von 409 Benutzern und 915 Rechnern unterstützt. Die Erwartungen der freiwilligen Teilnehmer eines BOINC-Projekts können anhand folgender repräsentativer Liste beschrieben werden:

- ein funktionierendes Forum
- zeitnahe Antworten auf ihre Fragen innerhalb des Forum

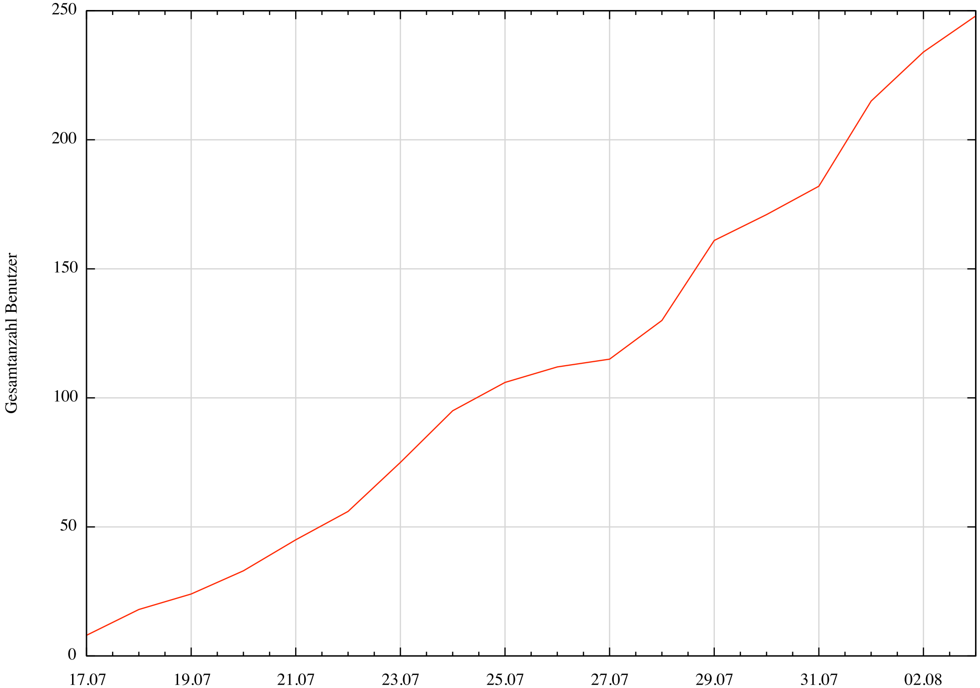

Abbildung 5.14.: Anzahl der Teilnehmer

- zeitnahe Lösung bestehender Probleme
- gerechte Verteilung der BOINC-Credits
- Anpassungen bestimmter Projekteigenschaften (Verschiebung von Deadlines, Erhöhung der maximalen Pakete pro Rechner, usw.)
- Kontakt zum Projektverwalter
- Zeitnahe Updates der Statistiken
- genaue Beschreibung der Projektziele

Die Gesamtanzahl der Rechner in Abbildung 5.16 verläuft ebenso wie die Funktion in Abbildung 5.14 nahezu linear.

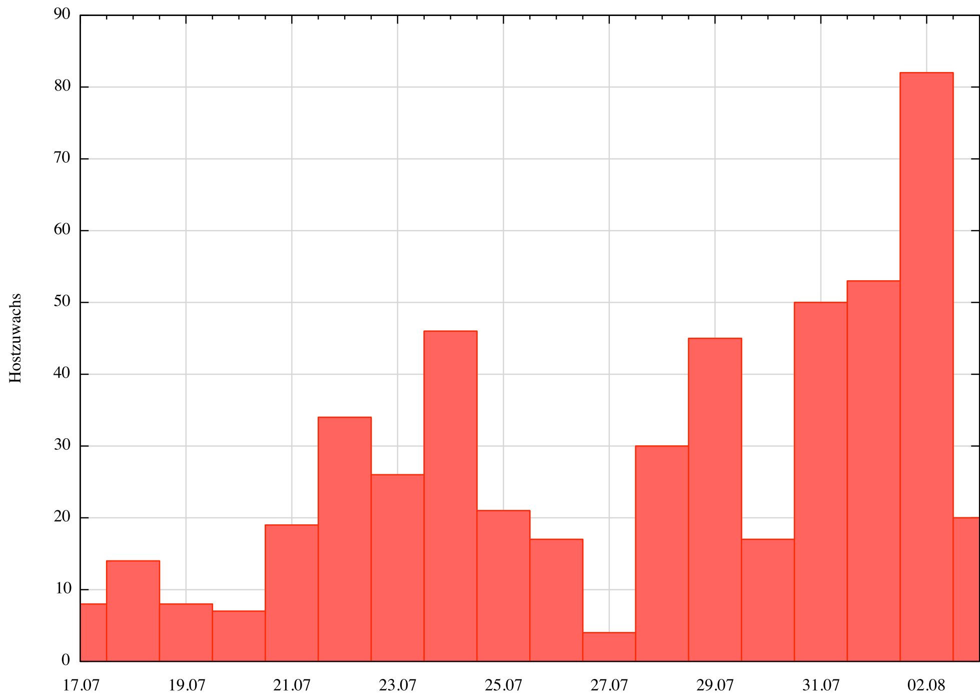

Abbildung 5.15.: Täglicher Rechnerzuwachs

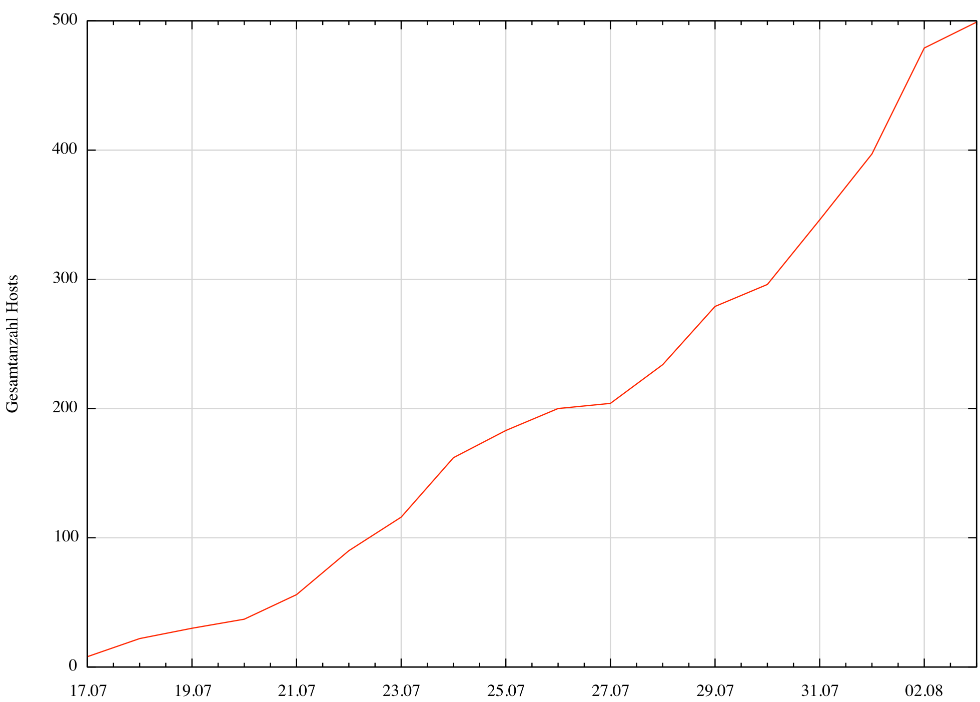

Abbildung 5.16.: Anzahl der Rechner

5.4. Performance und Vergleich

Ein x86-Client benötigt im Durchschnitt 1,5 Stunden pro Arbeitspaket, dies entspricht 2 Paketen pro 3 Stunden pro CPU-Kern. Somit errechnet ein solcher Client (mit 2 CPU-Kernen) 32 Pakete am Tag, dies entspricht 768000 Zahlen pro Tag (8,9 Zahlen pro Sekunde). Etwa 200 solcher Clients müssen durchgehend rechnen, um die durch dieses BOINC-Projekt täglich erreichte Rechenleistung zu realisieren. Diese Berechnung basiert auf der Anzahl der täglich errechneten Pakete auf primaboinca.com am 11.08.2010 von etwa 7500 Paketen.

Ein Vergleich zwischen der x86- und der Cell-BE-Architektur soll anhand von Tabelle 5.2 ermöglicht werden. Die in dieser Tabelle präsentierten Werte gelten für die Berechnung unter Verwendung eines CPU-Kerns. Auf der Playstation 3 Konsole kann anhand des Prozessor-Hyper-Threadings eine in etwa doppelt so große Leistung erreicht werden. Das IBM BladeCenter bestehend aus sieben QS20 Blades mit jeweils 2 PowerPC-Prozessoren ist ebenso unter Verwendung des Hyper-Threadings in der Lage etwa 400 Pakete (≈ 8000000 Zahlen) täglich zu überprüfen.

	∅ RZ[a] pro Zahl	RZ[a] pro Paket	Anz. Pakete pro Tag pro Core
x86	0,22	4480,60	19,28
Cell-BE[b]	0,06	1298,03	66,56
Differenz	0,16	3182,57	47,28

[a]Rechenzeit in Sekunden
[b]unter Verwendung von 3 SPUs

Tabelle 5.2.: Intel und Cell BE-Vergleich

Um einen Vergleich der Laufzeit zwischen AKS-Algorithmus und den in dieser Thesis überprüften Vermutungen zu ermöglichen, wurde Tabelle 5.3 mit errechneten Werten aus [CP08] formuliert. Hierbei ist zu bemerken, dass bei dieser Zeitmessung ausschließlich Primzahlen überprüft wurden.

	Rechenzeit (Sek.)				
Bit-Länge	17	32	44	63	128
AKS[a]@Intel	27,36	2697,60	-	-	25805[bc]
AKS[a]@Cell-BE	8,39	685,36	10131,58	124495,73	7373[bd]
Vermutung@Intel	0,000331	0,000739	0,001813	0,002457	0,018952
Vermutung@Cell-BE	0,004721	0,005941	0,007370	0,008956	0,089756

[a]Implementierung der Version mit Laufzeit $O((log\ N)^{7,5})$
[b]geschätzte Zeit
[c]≈ 1075 Tage
[d]≈ 307 Tage

Tabelle 5.3.: Vergleich der Laufzeiten zwischen AKS-Algorithmus und Vermutungen

5.5. Sonstige Evaluationen

Folgende Erkenntnisse wurden durch diese Thesis gewonnen:

F1 Wie hoch ist der Zeitaufwand ein komplettes BOINC-Projekt (inkl. Server) aufzusetzen?
Die Server-Dokumentation ist stellenweise lückenhaft und ungenau. Eine Step-by-Step-Installationsdokumentation ist nicht vorhanden. Falsche Einstellung wesentlicher Parameter konnten erst im laufenden Betrieb festgestellt werden.

F2 Welche Zeit braucht das Projekt, um einen bestimmten Zahlenbereich zu berechnen?
Ausgehend vom Zeitraum zwischen 17.07. und 03.08.2010 werden im Durchschnitt täglich etwa 3600 Pakete und somit 72000000 ungerade Zahlen gerechnet. Für den Zahlenbereich 10^{10} bis 10^{11} würden 625 Tage benötigt. Ausgehend vom Zeitraum zwischen 27.07. und 03.08.2010 werden durchschnittlich etwa 6250 Pakete pro Tag überprüft, somit wäre der zeitliche Aufwand für den gleichen Zahlenbereich nur noch halb so groß.

F3 Wo und mit welchem Aufwand sind Zeitslots auf Grids, welche Globus-Toolkit verwenden, zu bekommen? Folgende HPC[1]-Projekte unterstützen in Europa das Globus-Toolkit:

LRZ Das Leibniz-Rechenzentrum ist das Rechenzentrum für die Münchner Uni-

[1]High Performance Computing

versitäten und die Bayerische Akademie der Wissenschaften

ByGRID Gemeinsame Nutzung der Supercomputing-Ressourcen des LRZ und des RRZE (Regionales Rechenzentrum Erlangen)

D-Grid [1] Als gemeinsame Initiative mit der deutschen Wissenschaft und Wirtschaft fördert das Bundesministerium für Bildung und Forschung (BMBF) den Aufbau des D-Grids.

DEISA [2] Ein europäischer Verbund nationaler Supercomputing-Zentren für verteiltes Höchstleistungsrechnen.

PRACE [3] Eine pan-europäische Infrastruktur zum Höchstleistungsrechnen.

Der Zugang zu diesen Grids erfolgt meistens durch ein signiertes Zertifikat, welches beantragt werden muss. Leider sind Studenten der Hochschule RheinMain ohne weiteren Aufwand nicht berechtigt, eine solche Benutzerauthentifizierung zu beantragen.

F4 Welcher Architekturtyp eignet sich am besten für ein solches Projekt?

Das weltweit bereits vorhandene soziale BOINC-Netz sorgt für eine schnelle Verbreitung des Projekts, dadurch erreicht das Projekt in relativ kurzer Zeit eine gewisse Rechenleistung. Diese Rechenleistung ist nicht zeitbeschränkt. Die Rechenleistung in Grids, welche Globus Toolkit einsetzen, wird in der Regel für begrenzte Zeitslots vergeben. Somit eignet sich die *Volunteer-Computing-Architektur* für Projekte, deren Ziel nicht innerhalb solcher Zeitslots berechnet werden kann und für Projekte, welche in ihrer Laufzeit angepasst werden.

F5 In wie weit gehen die Implementierung der unterschiedlichen OS-Versionen auseinander?

Der Unterschied zwischen der Windows- und der Linux-Version ist sehr gering und mittels Präprozessor-Abfragen implementierbar. Die Cell-BE-Version verwendet bedingt durch die Architektur andere Bibliotheken und andere Methoden zur Berechnung der Pakete.

[1] http://www.d-grid.de/
[2] http://www.deisa.eu/
[3] http://www.prace-project.eu/

F6 Wie hoch ist der Zeitaufwand pro BOINC-Client-Version?

Die Implementierung der BOINC-Anwendungen war bedingt durch gut strukturierte Beispielanwendungen innerhalb des BOINC-Software-Pakets nicht besonders zeitaufwendig.

F7 Welche Dimensionen (Benutzer, Pakete) in welcher Zeit nimmt so ein BOINC-Projekt an?

Bereits vier Tage nach Freigabe des Projekts betrug die Anzahl gerechneter Pakete pro Tag im Durchschnitt 1200. Durchschnittlich kamen 10 neue Benutzer pro Tag hinzu. Mittlerweile sind bereits mehr als 300 Benutzer unter primaboinca.com angemeldet.

F8 Welche Voraussetzung muss ein Server für ein BOINC-Projekt erfüllen?

Der Ressourcenverbrauch eines BOINC-Servers wächst nur bedingt. Die Einrichtung von serverseitigen Cronjobs sorgten für Beseitigung veralteter Daten. Lediglich der durch BOINC verursachte Traffic wächst abhängig von der Größe der Arbeitspakete proportional zur erreichten Rechenleistung.

F9 In wie weit ist die Verwendung der Cell BE-Prozessoren unter BOINC möglich?

Die in dieser Thesis verwendeten PS3s laufen ziemlich stabil und arbeiten sehr performant.

F10 Welche Oberschranke hat r in dem überprüften Zahlenbereich, wenn gilt $n \in \mathbb{P}$, $r \nmid n$ und $n^2 = 1(mod\ r)$? Die höchste Zahl, welche die genannten Bedingungen erfüllt, war die 19.

Außerdem wurden folgende Hypothesen zu Anfang der Arbeit aufgestellt:

H1 Der Zahlenbereich zwischen $10^{10} < n < 10^{11}$ für $r < 100$ ist in der relativ kurzen Zeit berechenbar.

Nein. Die Überprüfung dieses Zahlenbereichs würde bei einer Rechenleistung von 430 GigaFLOPS (Stand 15.08.2010) etwa 300 Tage benötigen.

H2 Ohne Verwendung von Lenstras Heuristik wird kein Gegenbeispiel gefunden.

Ja. Die Implementierung dieser Heuristik würde die Erfolgswahrscheinlichkeit der Suche erhöhen.

H3 Die Anzahl der Nutzer, welche das BOINC-Projekt unterstützen, wird > 20 sein.

Ja. Die Anzahl der am Projekt teilnehmenden Nutzer war bereits nach wenigen Tagen > 50. Das weltweite BOINC-Netz bestehend aus Teams, Foren und statistikauswertenden Seiten sorgt für eine schnelle Verbreitung solcher Projekte.

Der durch das Projekt verursachte Datentraffic wird in Abbildung 5.17 dargestellt. Der Traffic wächst proportional zur Anzahl der versendeten und errechneten Arbeitspakete.

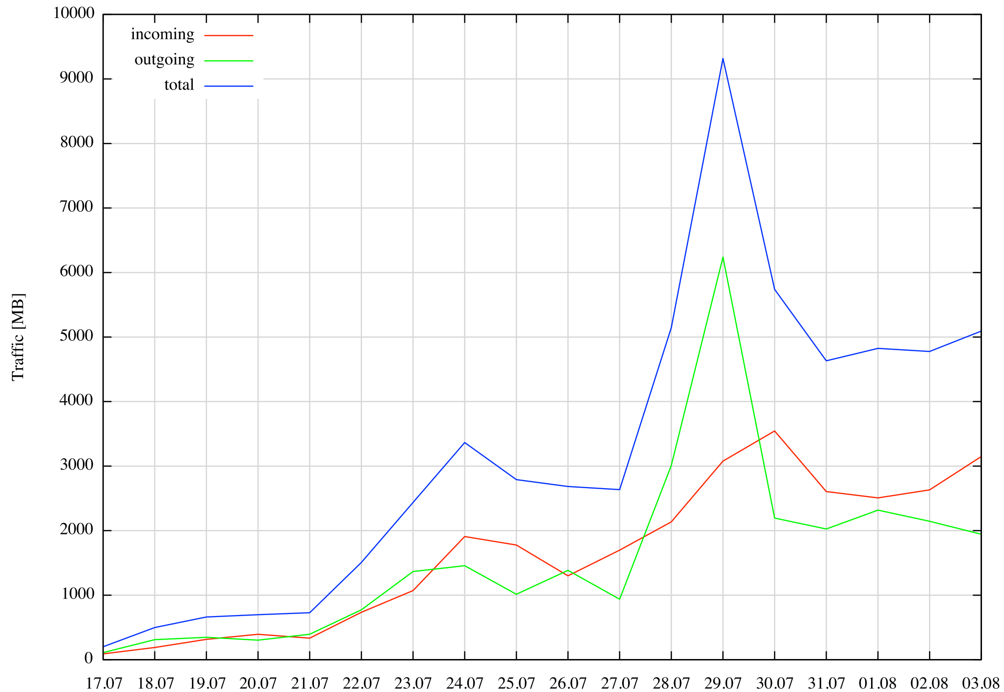

Abbildung 5.17.: Täglicher Datentraffic des BOINC-Servers

5.6. Umfrage

Um ein Profil der am primaboinca-Projekt teilnehmenden Nutzer zu erstellen, wurde eine Online-Umfrage durchgeführt. Diese Umfrage wurde unter Verwendung der SurveyMonkey™-Anwendung erstellt. Insgesamt wurden vier Mehrfachauswahlfragen und eine textuelle Frage formuliert. Die Umfrage war etwa sieben Tage online und 36 Projektteilnehmer (etwa 18% der damaligen gesamten Primaboinca-Teilnehmerzahl) nahmen daran teil. Der Verweis auf diese Umfrage erfolgte zum einem durch die Primaboinca-

Projektseite und zum anderen durch Einträge im Projekt-Forum. Die Teilnahme an dieser Umfrage erfolgte anonym und freiwillig.

1. How many others BOINC projects do you support? Das Ergebnis dieser Frage (siehe Abbildung 5.18) führte zu der Erkenntnis, dass mehr als ein Drittel der Befragten über 20 andere BOINC-Projekte unterstützen. Außerdem wird von keinem der Befragten ausschließlich das primaboinca-Projekt unterstützt.

		Beantwortung [%]	Anzahl
0		0 %	0
1 - 5		8,3 %	3
6 - 10		8,3 %	3
11 - 20		5,6 %	2
> 20		**77,8 %**	**28**
			36

Abbildung 5.18.: How many others BOINC projects do you support?

2. How many of these projects are math based? Mindestens ein weiteres mathematisches Projekt wird von jedem Umfrageteilnehmer unterstützt. Mehr als 8% der Teilnehmer stellen ihre Rechenleistung mehr als 20 mathematischen Projekten zur Verfügung.

		Beantwortung [%]	Anzahl
0		0 %	0
1 - 5		**44,4 %**	**16**
6 - 10		30,6 %	11
11 - 20		16,7 %	6
> 20		8,3 %	3
			36

Abbildung 5.19.: How many of these projects are math based?

3. How would you classify your mathematical skills? Obwohl die Beteiligung an mathematischen Projekten, wie in Abbildung 5.20 ersichtlich, recht groß ist, stufen sich $22,2\%$ der Teilnehmer als Anfänger ein.

		Beantwortung [%]	Anzahl
expert		0 %	0
high qualified		16,7 %	6
intermediatte		**44,4 %**	**16**
pre-intermediatte		16,7 %	6
beginner		22,2 %	8
			36

Abbildung 5.20.: How would you classify your mathematical skills?

4. Do you know the AKS-Algorithm? Obwohl mehr als 40% der Umfrageteilnehmer die eigenen mathematischen Kenntnisse als mittelmäßig einstufen, kennen nur $16,7\%$ den AKS-Algorithmus.

		Beantwortung [%]	Anzahl
Yes		16,7 %	6
No		**83,3 %**	**30**
			36

Abbildung 5.21.: Do you know the AKS-Algorithm?

5. I'm supporting this project, because ... Folgende Aussagen zeigen eine repräsentative Auswahl der Antworten:

- *" It's a BOINC project with a seemingly worthy goal. I especially like to help new projects and see them succeed. "*
- *" I will learn more math (when I eventually read and understand the 2 conjectures). "*
- *" Actually I support every boinc project. But a prime based project is always a nice addition. "*
- *" I want a million credits on all projects. "*

KAPITEL 6

FAZIT

Die in dieser Arbeit vorgenommenen theoretischen Untersuchungen zum Algorithmus ermöglichten, den Worst Case zu identifizieren. Messungen belegten die deutlich niedrigere Komplexität der untersuchten Vermutungen im Vergleich zum AKS-Algorithmus. Mit Hilfe der eingesetzten Methoden zur Performanceoptimierung konnte die Laufzeit des implementierten Algorithmus um 40% reduziert werden. Die eindeutige Identifizierung der am häufigsten aufgerufenen rechenintensiven Stellen im Programmcode ermöglichte, durch gezielte Optimierungen die insgesamt notwendige Rechenzeit deutlich zu senken. Dennoch wurde die benötigte Rechenzeit zur Überprüfung eines bestimmten Zahlenbereichs in dieser Thesis unterschätzt.

In dieser Thesis wurde bewiesen, dass sich die Volunteer-Computing-Architektur für ein gemeinnütziges, wissenschaftliches Projekt hervorragend eignet. Die bereits bestehende große und aktive BOINC-Community ermöglicht, innerhalb kürzester Zeit auf eine hohe Rechenleistung zuzugreifen. Die geringen Anforderungen an Ressourcen des eingesetzten BOINC-Servers stellten sich ebenfalls als Vorteil heraus. Der administrative Zeitaufwand eines BOINC-Projekts und somit des BOINC-Servers wurde in dieser Arbeit allerdings unterschätzt. Leider erweist sich das BOINC-Framework bei näherer Betrachtung als etwas undurchdacht, hier zeigt sich schnell, dass es sich um ein gewachsenes und nicht um ein theoretisch strukturiert entworfenes Projekt handelt. Eine in BOINC nicht vorgesehene Funktionalität, die sich in dieser Thesis als nützlich gezeigt hat, ist die Verwaltung paralleler Projekte. Nur durch eine Anpassung des Algorithmus in der projektspezifischen Client-Applikation ist es möglich, zusätzlich zur Überprüfung der Hypothesen einen bestimmten Teil der vorhandenen Ressourcen für die Überprüfung von Lenstras Heuristik einzusetzen. Wünschenswert wäre hier die Möglichkeit, zwei unterschiedliche Applika-

tionen bereitzustellen, die jeweils von einem bestimmten Anteil der Clients bearbeitet werden.

Die Cell BE Architektur erwies sich als sehr geeignet für das Projekt, sie eignet sich besonders für mathematische Berechnungen, die eine partielle Abarbeitung zulassen. Im Einsatz als Client für das BOINC Projekt zeigte sich diese Architektur als sehr performant.

Der anfangs ebenfalls in Betracht gezogene Grid-Computing-Ansatz stellte sich dagegen als ungeeignet heraus. Der zeitlicher Aufwand für die Einarbeitung in die Globus-Toolkit-Architektur war sehr hoch. Die lose Kopplung dieser Architektur macht die Installation und Konfiguration dieser Middleware sehr aufwändig. Der Grid-Computing Ansatz erschien nicht attraktiv, da der Zugang zu den nötigen Ressourcen mit zu großem Aufwand verbunden ist.

Um die weitere Suche nach einem Gegenbeispiel zu beschleunigen, kämen verschiedene Maßnahmen in Frage. Die Einrichtung eines zweiten BOINC-Projekts, welches die Implementierung von Lenstras Heuristik nutzt, würde die Erfolgswahrscheinlichkeit der Suche deutlich erhöhen. Weitere Optimierungsmaßnahmen könnten die Performance weiter verbessern. Die Implementierung eines Clients, welcher die CUDA-Technologie unterstützt, würde die Rechenleistung der Grafikkarten nutzen und somit mehr Rechenleistung erbringen.

Grundsätzlich gilt, dass Aussagen über die Existenz eines Gegenbeispiels im bisher überprüften Zahlenbereich nur bedingt gelten, da die Korrektheit der vorhandenen Algorithmen nicht bewiesen wurde. Die algorithmische Zahlentheorie wird unabhängig von den zur Verfügung stehenden Ressourcen einen mathematischen Beweis nie ersetzen können, sie kann immer nur als Hilfsmittel dienen.

David Anderson, Hendrik Lenstra und Roman Popovych zeigten sich in der Kommunikation per E-Mail interessiert und hilfsbereit.

Literaturverzeichnis

[ACA06] ANDERSON, David P. ; CHRISTENSEN, Carl ; ALLEN, Bruce: Designing a runtime system for volunteer computing. In: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA : ACM, 2006. – ISBN 0–7695–2700–0, S. 126

[ACK+02] ANDERSON, David P. ; COBB, Jeff ; KORPELA, Eric ; LEBOFSKY, Matt ; WERTHIMER, Dan: SETI@home: an experiment in public-resource computing. In: *Commun. ACM* 45 (2002), Nr. 11, S. 56–61. `http://dx.doi.org/http://doi.acm.org/10.1145/581571.581573`. – DOI http://doi.acm.org/10.1145/581571.581573. – ISSN 0001–0782

[AH92] ADLEMAN, Leonard M. ; HUANG, Ming-Deh A.: *Lecture Notes in Mathematics*. Bd. 1512: *Primality testing and abelian varieties over finite fields*. Berlin : Springer-Verlag, 1992. – ISBN 3–540–55308–8

[AKS02] AGRAWAL, M. ; KAYAL, N. ; SAXENA, N.: PRIMES in P. In: *Ann. of Math. (2)* 160 (2002), Nr. 2, S. 781–793

[And02] ANDERSON, P.: *Opinion - Volunteer computing: grid or not grid?* Version: 2002. `http://www.isgtw.org/?pid=1000527`, Abruf: 05.08.2010

[And04] ANDERSON, David P.: BOINC: A System for Public-Resource Computing and Storage. In: *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2256–4, S. 4–10

[APR83] ADLEMAN, L. M. ; POMERANCE, C. ; RUMELY, R. S.: On distinguishing prime numbers from composite numbers. In: *AM* 117 (1983), S. 173–206

[BP01a] BHATTACHARJEE, R. ; PANDEY, P.: Primality testing, IIT Kanpur. In: *Available at http://www.cse.iitk.ac.in/research/btp2001/primality.html* (2001)

[BP01b] BHATTACHARJEE, R. ; PANDEY, P.: *Primality testing, IIT Kanpur, 2001*. Version: 2001. http://www.cse.iitk.ac.in/research/btp2001/primality.html, Abruf: 25.07.2010

[Cam10a] CAMPOS, Fabio: *Mailkommunikation*. Direkte Mailkommunikation mit H. Lenstra, 2010

[Cam10b] CAMPOS, Fabio: *Mailkommunikation*. Direkte Mailkommunikation mit R. Popovych, 2010

[CF05] COHEN, Henri (Hrsg.) ; FREY, Gerhard (Hrsg.): *Handbook of elliptic and hyperelliptic curve cryptography*. CRC Press, 2005. – ISBN 1–58488–518–1

[Cli10] CLIMAPREDICTION.NET: *Climaprediction.net Homepage*. Version: 2010. http://www.climateprediction.net/, Abruf: 23.07.2010

[Col10] COLLABORATION, LIGO S.: *Einstein@home Homepage*. Version: 2010. http://einstein.phys.uwm.edu/, Abruf: 23.07.2010

[CP03] CRANDALL, R. ; PAPADOPOULOS, J.: On the implementation of AKS-class primality tests. (2003)

[CP08] CAMPOS, Fabio ; PALER, Alexandru: *Primes on a PS3. AKS-Algorithm on the Cell Broadband Engine*. LV (8161) Parallele und verteile Algorithmen. http://primaboinca.com/aks.pdf. Version: 2008

[CPCP05] CRANDALL, Richard ; POMERANCE, Carl ; CRANDALL, Richard ; POMERANCE, Carl: *Prime numbers: a computational perspective. Second Edition*. Springer, 2005

[CS09] COSTIGAN, Neil ; SCHWABE, Peter: Fast Elliptic-Curve Cryptography on the Cell Broadband Engine. In: *AFRICACRYPT '09: Proceedings of the*

2nd International Conference on Cryptology in Africa. Berlin, Heidelberg : Springer-Verlag, 2009. – ISBN 978–3–642–02383–5, S. 368–385

[Erd60] ERDŐS, Paul: Remarks on number theory III. On addition chains. In: *Acta Arith.* (1960), S. 77–81

[GK86] GOLDWASSER, S. ; KILIAN, J.: Almost all primes can be quickly certified. In: *STOC'86, Proceedings of the 18th Annual ACM Symposium on the Theory of Computing (Berkeley, CA, 1986)*, ACM, 1986, S. 316–329

[Gor98] GORDON, Daniel M.: A Survey of Fast Exponentiation Methods. In: *Journal of Algorithms* 27 (1998), S. 129–146

[Gra05] GRANVILLE: It is Easy to Determine Whether a Given Integer is Prime. In: *BAMS: Bulletin of the American Mathematical Society* 42 (2005)

[KDH+05] KAHLE, J. A. ; DAY, M. N. ; HOFSTEE, H. P. ; JOHNS, C. R. ; MAEURER, T. R. ; SHIPPY, D.: Introduction to the cell multiprocessor. In: *IBM J. Res. Dev.* 49 (2005), Nr. 4/5, S. 589–604. – ISSN 0018–8646

[Knu97] KNUTH, Donald E.: *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997 `http://portal.acm.org/citation.cfm?id=270146`. – ISBN 0201896842

[LP02] LENSTRA, H ; POMERANCE, C.: *Remarks on Agrawal's Conjecture*. Version: 2002. `http://www.aimath.org/WWN/primesinp/articles/html/50a/`, Abruf: 23.07.2010

[Mil76] MILLER, G. L.: Riemann's hypothesis and tests for primality. In: *Computer and System Sci.* 13 (1976), S. 300–317

[Moo65] MOORE, Gordon E.: Cramming More Components onto Integrated Circuits. In: *Electronics* 38 (1965), Nr. 8, 114–117. `http://www.intel.com/technology/mooreslaw/index.htm`

[Oev96] OEVEL, Walter: *Einfuhrung in die Numerische Mathematik*. Spektrum Akademischer Verlag, 1996

[Pop09] POPOVYCH, Roman: *A note on Agrawal conjecture*. Cryptology ePrint Archive, Report 2009/008, 2009. – http://eprint.iacr.org/

[Rab80] RABIN, M. O.: Probabilistic algorithms for testing primality. In: *Number Theory* 12 (1980), S. 128–138

[Rei09] REINDERS, James: *Intel director explains why multicore processors offer better performance on less power*. ZDNet?s Parallelism Breakthrough series, 2009. – http://blogs.techrepublic.com.com/itdojo/?p=605

[SG06] SYSTEMS, IBM ; GROUP, Technology: *Cell Broadband Engine Programming Tutorial*. International Business Machines Corporation, 2006

[SG07] SYSTEMS, IBM ; GROUP, Technology: *Cell Broadband Engine Programming Handbook*. International Business Machines Corporation, 2007

[SS77] SOLOVAY, R. ; STRASSEN, V.: A fast Monte-Carlo test for primality. In: *SIAM Journal on Computing* 6 (1977), S. 84–86

[Sti07] STILLER, Andres: *Cell-kultur*. CT Special - Playstation 3, 2007

ANHANG B

INHALT DER CD

CD 1

Dateien

- *thesis.pdf* (PDF-Datei dieser Thesis)
- *expose.pdf* (Expose der Thesis)
- *Kurzfassung.pdf* (Kurzfassung der Thesis)
- *Umfrage.pdf* (Screenshot der gestellten Umfrage)

Verzeichnisse

- *Figures* (Digitale Kopie der verwendeten Bilder)
- *gnuplot* (Implementierte GNUPlot-Skripte und Daten zur Grafikerzeugung)
- *Windows* (Visual Studio-Projekt der BOINC-Anwendung)
- *Linux* (Linux-Version der Anwendung und Quellcode)
- *Cell-BE* (Cell-BE-Version der Anwendung und Quellcode)
- *Work-Generator* (Projektspezifischer Generator und Quellcode)
- *Mathematica* (Verwendete Mathematica-Skripte)
- *Results* (Gepackte beispielhafte Ergebnisdateien)