Technische Universität Dresden

Zentrum für Informationsdienste
und Hochleistungsrechnen
Prof. Dr. Wolfgang E. Nagel

Hauptseminar

Maximilian Knespel 3803449
Moritz Häussler 4096912
Korcan Kirikici 4049055

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Nico Hoffmann

Dresden,

Inhaltsverzeichnis

1 Einführung

Im Rahmen dieser Belegarbeit soll ein Ansatz entwickelt werden, um in Java oder Scala auf Clustern auf Grafikkarten zu berechnen. Es wurde sich für eine Kombination von Spark für die Kommunikation im Cluster und Rootbeer für die Grafikkartenprogrammierung entschieden.
Zuerst wird in den Kapiteln 2-5 die benutzten Algorithmen und Bibliotheken vorgestellt, in Kapitel 6 wird die eigene Implementierung dokumentiert und in Kapitel 7 werden Benchmarks dieser Implementierung vorgestellt.

2 Rechenintensiver Testalgorithmus

2.1 Monte-Carlo Algorithmen

Monte-Carlo-Algorithmen bezeichnet Algorithmen, die mit Hilfe von (Pseudo-)Zufallszahlen versuchen das gesuchte Ergebnis statistisch zu approximieren. Dafür werden Stichproben aus statistischen Verteilungen durch z.B. physkalisch begründete Abbildungen transformiert und jene Ergebnisse statistisch ausgewertet. Diese Art von Verfahren eignet sich z.B. zur Berechnung von Integralen über zig Koordinaten, die mit üblichen Newton-Cotes-Formeln aufgrund der hohen Dimensionalität nicht praktikabel wären. Eine andere Anwendung ist die Analyse von durch kosmischer Strahlung ausgelösten Teilchenschauern mit Hilfe von Markov-Ketten[12].
Monte-Carlo-Algorithmen sind als statistische Stichprobenverfahren schon länger bekannt, wurden aber erst mit dem Aufkommen der ersten Computer, z.B. dem ENIAC um 1947-1949, praktikabel[11]. Der Name, nach der Spielbank "Monte-Carlo", wurde von N.Metropolis vorgeschlagen und hielt sich seitdem. Der Vorschlag zu dieser Art von Algorithmus kam von John von Neumann auf, als man mit dem ENIAC thermonukleare Reaktionen simulieren wollte. Aber Fermi wird nachgesagt schon Jahre zuvor statistische Stichprobenverfahren in schaflosen Nächten händisch angewandt zu haben und mit den überraschend genauen Resultaten seine Kollegen in Staunen zu versetzen.
Monte-Carlo-Verfahren sind inhärent leicht zu parallelisieren, da eine Operation, die Simulation, mehrere Tausend oder Milliarden Mal ausgeführt wird. Eine Schwierigkeit besteht jedoch darin den Pseudozufallszahlengenerator (pseudorandom number generator - PRNG) korrekt zu parallelisieren. Das heißt vor allem muss man unabhängige Startwerte finden und an die parallelen Prozesse verteilen. - Zeitangaben sind hierbei nicht sinnvoll. Das betrifft alle möglichen Zeitgeber in Rechnern wie z.B. .

2.1.1 Berechnung von Pi

Um Pi zu berechnen wird Pi als Integral dargestellt, da sich beschränkte Integrale durch Monte-Carlo-Verfahren approximieren lassen.

$$\pi = \int \begin{cases} 1 & |x^2 + y^2| \leq 1 \\ 0 & \text{sonst} \end{cases} \mathrm{d}x\mathrm{d}y \tag{2.1}$$

Das heißt wir integrieren die Fläche eines Einheitskreises. Durch die Ungleichung wissen wir auch, dass nur für $x, y \in [-1, 1]$ der Integrand ungleich 0 ist.
Da es programmatisch trivialer ist Zufallszahlen aus dem Intervall $[0, 1]$ anstatt $[-1, 1]$ zu ziehen,

wird das Integral über den Einheitskreis in ein Integral über einen Viertelkreis geändert:

$$\pi = 4 \int_0^\infty \mathrm{d}x \int_0^\infty \mathrm{d}y \begin{cases} 1 & |x^2 + y^2| \leq 1 \\ 0 & \text{sonst} \end{cases} \tag{2.2}$$

Das Vorgehen ist nun wie folgt

1. Setze die Zählvariable `Summe` auf 0
2. Ziehe für x und y je eine gleichverteilte Zufallszahl aus dem Intervall $[0, 1]$
3. Falls $x^2 + y^2 < 1$, dann erhöhe `Summe` um 1
4. Gehe zu 2.

Mathematisch ausgedrückt also:

$$\mu_N = \langle f(\vec{x}_i) \rangle := \frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i), \ \vec{x}_i \text{ uniform zufallsverteilt aus } \Omega := [0, 1] \times [0, 1] \tag{2.3}$$

Im allgemein ist f eine beliebige Funktion, aber für die Berechnung von Pi ist es die Einheitskugel in 2D, vgl.Gl.2.2.
In Python kann man dies, wenn man sich auf Einkernprozessoren einschränkt, mit NumPy[1] in nur wenigen zeilen niederschreiben:

```
1 from numpy import *
2 N=10000000
3 x=random.rand(N)
4 y=random.rand(N)
5 pi = 4.0 * sum( x*x + y*y < 1 ) / N
```

Der Vollständigkeit halber seien kurz ein paar Worte zu den Rändern verloren, das betrifft die Zufallszahlen die entweder aus einem rechtsoffenem oder geschlossenen Intervall $[0, 1]$ stammen können und den Vergleich, der die Gleichheit mit einschließen kann oder nicht.
Aus der Integraltheorie ist klar, dass die Ränder ein Nullmaß haben und damit keine Rolle spielen. Aber für diskrete Verfahren könnte dies zu einer zusätzlichen systematischen Fehlerquelle führen, der Fehlerskalierverhalten möglicherweise beeinträchtigt.
Am Beispiel von nur vier Zuständen für Zufallszahlen für den rechtsoffenen Fall, also $x, y \in \{0, 0.25, 0.5, 0.75\}$, sei dies einmal durchgedacht. Damit ergibt sich

$$x^2 + y^2 = \{0, 0.0625, 0.125, 0.25, 0.3125, 0.5, 0.5625, 0.625, 0.8125, 1.125\} \tag{2.4}$$

Hier macht es aufgrund der begrenzten Anzahl an Zuständen, unter denen die 1.0 ohnehin nicht auftritt, keinen Unterschied ob man $<$ oder $\leq$ vergleicht, man erhielte Pi zu 3.6. Hinzu kommt aber, dass Zustände auf den Grenzen $x = 0$ und $y = 0$ liegen, sodass die Grenzen vierfach gezählt werden da wir nur den Viertelkreis berechnen und mit vier multiplizieren.
Man hat also ohnehin immer einen Diskretisierungsfehler von $O\Delta x)$ wobei Δx die Diskretisierungslänge zwischen zwei Zuständen ist. Angemerkt sei, dass dies für Gleitkommazahlen komplizierter gestaltet.

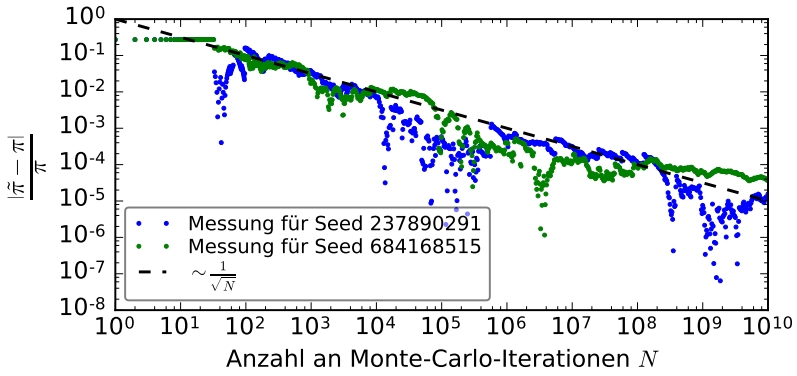

Abbildung 2.1: captiontext

Abschließend sei angemerkt, dass Monte-Carlo-Methoden dafür gedacht sind einen praktisch unerschöpflichen Raum Stichprobenartig auszutesten, sodass Diskretisierungs- und Randfehler ohnehin als vernachlässigbar angenommen werden. Wenn man merkt, dass es zu Diskretisierungsfehler wie obig an den Rändern kommt, oder man gar die Anzahl aller möglichen Zustände an Zufallszahlen erschöpft hat und sich die Approximation damit nicht mehr verbessern kann, sollte man über ein anderes Verfahren nachdenken oder den Zufallsgenerator anpassen und z.B. mit 128-Bit statt 32-Bit betreiben. Auch die maximale Periodenlänge von Pseudozufallsgeneratoren spielt hier eine Rolle!
Da die Monte-Carlo-Pi-Integration einer Mittelwertbildung entspricht, vgl. Gl.2.3, ist die statistische Unsicherheit gegeben durch die Standardabweichung des Mittelwerts σ_{μ_N}, welche gegeben ist als

$$\sigma_{\mu_N} \frac{\sigma}{\sqrt{N}} \tag{2.5}$$

wobei σ die Standardabweichung der Stichprobe ist, vgl. Anhang A. Wenn f_i in einem beschränkten Intervall liegt, dann ist auch die Standardabweichung der Stichproben f_i beschränkt, sodass die Standardabweichung auf den Mittelwert $\propto \frac{1}{\sqrt{N}}$ abnimmt.

2.1.2 Pseudozufallszahlgeneratoren

2.2 Vergleich - Alternativen

3 Rootbeer

Rootbeer[16] ist ein von Philip C. Pratt-Szeliga entwickeltes Programm und Bibliothek welches das Schreiben von CUDA-Kerneln in Java erleichtert. Zum aktuellen Zeitpunkt Mai 2016 hat Rootbeer leider noch Beta-Status und wurde seit ca. einem Jahr nicht weiterentwickelt[13].
Mit Rootbeer lassen sich CUDA-Kernel direkt in Java schreiben anstatt in C/C++. Dafür muss zuerst vom Nutzer die `org.trifort.rootbeer.runtime.Kernel`-Klasse implementiert und zu einer Java class-Datei kompiliert werden. Wenn das komplette zu schreibende Programm zu einer jar-Datei zusammengefügt wurde, dann muss diese noch einmal an den Rootber-Compiler übergeben werden. Rootbeer nutzt Soot[8, 9], um den Bytecode in Jimple zu übersetzen. Jimple ist eine vereinfachte Zwischendarstellung von Java-Bytecode, welcher ca. 200 verschiedene Befehle besitzt, in Drei-Address-Code mit nur 15 Befehlen. Der Jimple-Code wird dann analysiert und in CUDA übersetzt welcher dann mit einem installierten NVIDIA-Compiler übersetzt wird. All das geschieht automatisch, aber die Zwischenschritte kann man zur Fehlersuche unter Linux in `$HOME/.rootbeer/` einsehen. Die erstellte cubin-Datei wird zusammen mit `Rootbeer.jar` der jar-Datei des selbstgeschriebenen Programms hinzugefügt.
Die zweite große Vereinfachung, die Rootbeer zur Verfügung stellt, ist die Automatisierung des Datentransfers zwischen GPU und CPU. Das besondere hierbei ist, dass Rootbeer die Nutzung von beliebigen, also insbesonderen auch nicht-primitiven Datentypen erlaubt. Diese Datentypen serialisiert Rootbeer automatisch und unter Nutzung aller CPU-Kerne und transferiert sie danach auf die Grafikkarte.
Diese zwei Vereinfachungen obig machen die erste Nutzung von Rootbeer verglichen zu anderen Lösungen sehr einfach, sodass Rootbeer insbesondere für das Erstellen von Prototypen günstig ist. In Kontrast dazu ist es jedoch auch mögliche wiederum sehr nah an der Grafikkarte zu programmieren. Dafür kann man mit Rootbeer auch manuell die Kernel-Konfiguration angeben, mehrere GPUs ansprechen und auch shared memory nutzen.

3.1 Nutzung

Für die Nutzung von Rootbeer unter Debian-Derivaten ist das `openjdk-7-jdk`-Pakt und das `nvidia-cuda-toolkit`-Paket notwendig. Leider funktioniert Rootbeer nicht mit JDK 8. JDK 7 funktioniert vollends in den hier durchgeführten Beispielen, aber volle Unterstützung ist bisher nur für JDK 6 offiziell gegeben[14].
Ein Minimalbeispiel für einen Kernel, dessen Threads nur ihre ID in einen Array schreiben sieht wie folgt aus:

```
1 import org.trifort.rootbeer.runtime.Kernel;
2 import org.trifort.rootbeer.runtime.RootbeerGpu;
3
4 public class ThreadIDsKernel implements Kernel
```

```
{
    private int miLinearThreadId;
    private long[] mResults;
    /* Constructor which stores thread arguments: seed, diceRolls */
    public ThreadIDsKernel( int riLinearThreadId, long[] rResults )
    {
        miLinearThreadId = riLinearThreadId;
        mResults         = rResults;
    }
    public void gpuMethod()
    {
        mResults[ miLinearThreadId ] = RootbeerGpu.getThreadId();
    }
}
```

minimal/ThreadIDsKernel.java

Der Aufruf der Kernels geschieht über eine Liste von Kernel-Objekten, die per Konstruktor mit Parametern initialisiert wurden. Diese Liste wird an `rootbeer.run` übergeben, der den Kernel dann mit einer passenden Konfiguration startet.

```
import java.io.*;
import java.util.List;
import java.util.ArrayList;
import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.Rootbeer;

public class ThreadIDs
{
    /* prepares Rootbeer kernels and starts them */
    public static void main( String[] args )
    {
        final int nKernels = 3000;
        long[] results = new long[nKernels];
        /* List of kernels / threads we want to run in this Level */
        List<Kernel> tasks = new ArrayList<Kernel>();
        for ( int i = 0; i < nKernels; ++i )
        {
            results[i] = 0;
            tasks.add( new ThreadIDsKernel( i, results ) );
        }
        Rootbeer rootbeer = new Rootbeer();
        rootbeer.run(tasks);
        System.out.println( results[0] );
    }
}
```

minimal/ThreadIDs.java

Zuerst müssen diese beiden Dateien mit ‘javac‘ kompiliert werden und dann zusammen mit einer ‘manifest.txt‘-Datei, die die Einsprungsklasse anzeigt, zu einem Java-Archiv gepackt werden, welches im letzten Schritt mit Rootbeer kompiliert und dann ausgeführt wird.

```
javac ThreadIDsKernel.java -classpath "$ROOTBEER_ROOT/Rootbeer.jar:." &&
```

```
javac ThreadIDs.java       -classpath "$ROOTBEER_ROOT/Rootbeer.jar:." &&
cat > manifest.txt <<EOF
Main-Class: ThreadIDs
Class-Path: .
EOF
jar -cvfm preRootbeer.tmp.jar manifest.txt ThreadIDsKernel.class ThreadIDs.class
    &&
java -jar "$ROOTBEER_ROOT/Rootbeer.jar" preRootbeer.tmp.jar ThreadIDs.jar -64bit
    &&
java -jar ThreadIDs.jar
```

minimal/compile.sh

Bei der Kompilierung mit `Rootbeer.jar` muss beachtet werden, dass alle benutzten Klassen mit in der jar-Datei enthalten sind, sonst quittiert Soot mit folgender Fehlermeldung:

```
java.lang.RuntimeException: cannot get resident body for phantom class
```

Bei Nutzung von scala heißt das insbesondere, dass `scala.jar` mit in das Java-Archiv gepackt werden muss.
Weiterhin ist zu beachten, dass die an Rootbeer/Soot übergebene Datei entgegen der Linux-Ideologie mit `.jar` enden muss, insbesondere führen Dateinamen wie `gpu.jar.tmp` zu der Fehlermeldung

```
There are no kernel classes. Please implement the following interface to use
    rootbeer:
org.trifort.runtime.Kernel
```

3.2 Multithreaded Rootbeer

Bei der Benutzung von Rootbeer, kam es leider zu einigen Bugs, die teilweise in einem geforkten Repository auf Github behoben wurden, siehe [15]. Das wichtigste Problem sei hier kurz vorgestellt.
Auf Taurus ist das Heimverzeichnis über alle Knoten zugreifbar. Dies führt zu einen Problem, wenn man Rootbeer auf verschiedenen Nodes oder von verschiedenen Threads aus nutzen möchte, da bei einem Kernel-Aufruf `~/.rootbeer/rootbeer_cuda_x64.so.1` aus der jar-Datei extrahiert wird. Wenn also ein Thread meint die Datei fertig entpackt zu haben, während ein anderer Thread die Datei nochmal entpackt, aber noch nicht fertig ist, dann kam es in ersten Versuchen zu einem `java.lang.UnsatisfiedLinkError`. Dies wurde behoben, indem ein Pfad aus Hostname, Prozess-ID und Datum genutzt wird:

```
m_rootbeerhome = home + File.separator + ".rootbeer" + File.separator
               + getHostname() + File.separator
               + getProcessId("pid") + "-" + System.nanoTime()
               + File.separator;
```

Dies resultiert z.B. in diesen Pfad: `~/.rootbeer/taurusi2093/7227-311934180383710/`.

4 Scala

4.1 Kurzbeschreibung

4.2 Funktionsweise

4.3 Anwendung

4.4 Probleme

Counting Variable ..

5 Spark

Spark ist ein Programmierframework für Datenanalyse auf Clustern, was vor allem zusammen mit dem Stichwort Big-Data und maschinellem Lernen an Beliebtheit gewonnen hat. Es vereinigt hierbei ausfallgehärtet Funktionalitäten von Batchverarbeitungssystem bzw. Cluster-Management wie Slurm, paralleler Kommunikation zwischen Prozessen wie OpenMPI und OpenMP und Programmierbibliotheken. Die von Spark zur Verfügung gestellten primitiven sind inhärent hochparallel und ausfallgehärtet ausführbar und aus Java, Scala und Python heraus ansprechbar. Aus letzteren beiden auch interaktiv, was das Experimentieren und schnelle Erstellen von Prototypen vereinfacht. Es gibt erweiternde Bibliotheken für maschinelles Lernen und Graphen. Der große Funktionsumfang macht es für Anfänger schwer einzuordnen, was Spark ist, aber ermöglich das schnelle interaktive und nichtinteraktive Auswerten von Big Data auf Clustern.

5.1 Konfiguration von Spark auf Taurus

Um Spark nutzen zu können müssen zuerst Master- und Slave-Knoten gestartet werden. Hier soll es nur einen Master-Knoten und n Slave-Knoten geben. Damit alle gleichzeitig gestartet werden, kann Slurms `--multi-prog` Option genutzt werden, welche als Argument einen Pfad zu einer Konfigurationsdatei erwartet, in der für jeden Rank ein auszuführendes Programm angegeben werden muss.
Alternativ kann man auch anhand von der Umgebungsvariable `SLURM_PROCID` im Skript entweder einen Master-Knoten oder einen Slave-Knoten starten. Letzeres wurde aufgrund der Übersichtlichkeit, d.h. alle Funktionalitäten in einem Skript zu haben, gewählt, siehe Listing B.1.
Wenn Spark gestartet ist, kann sich z.B. mit einer aktiven Eingabeaufforderung an den Master verbunden werden:

```
1 spark-shell --master=$MASTER_ADDRESS
```

6 Eigene Implementierung

6.1 Kompilierung

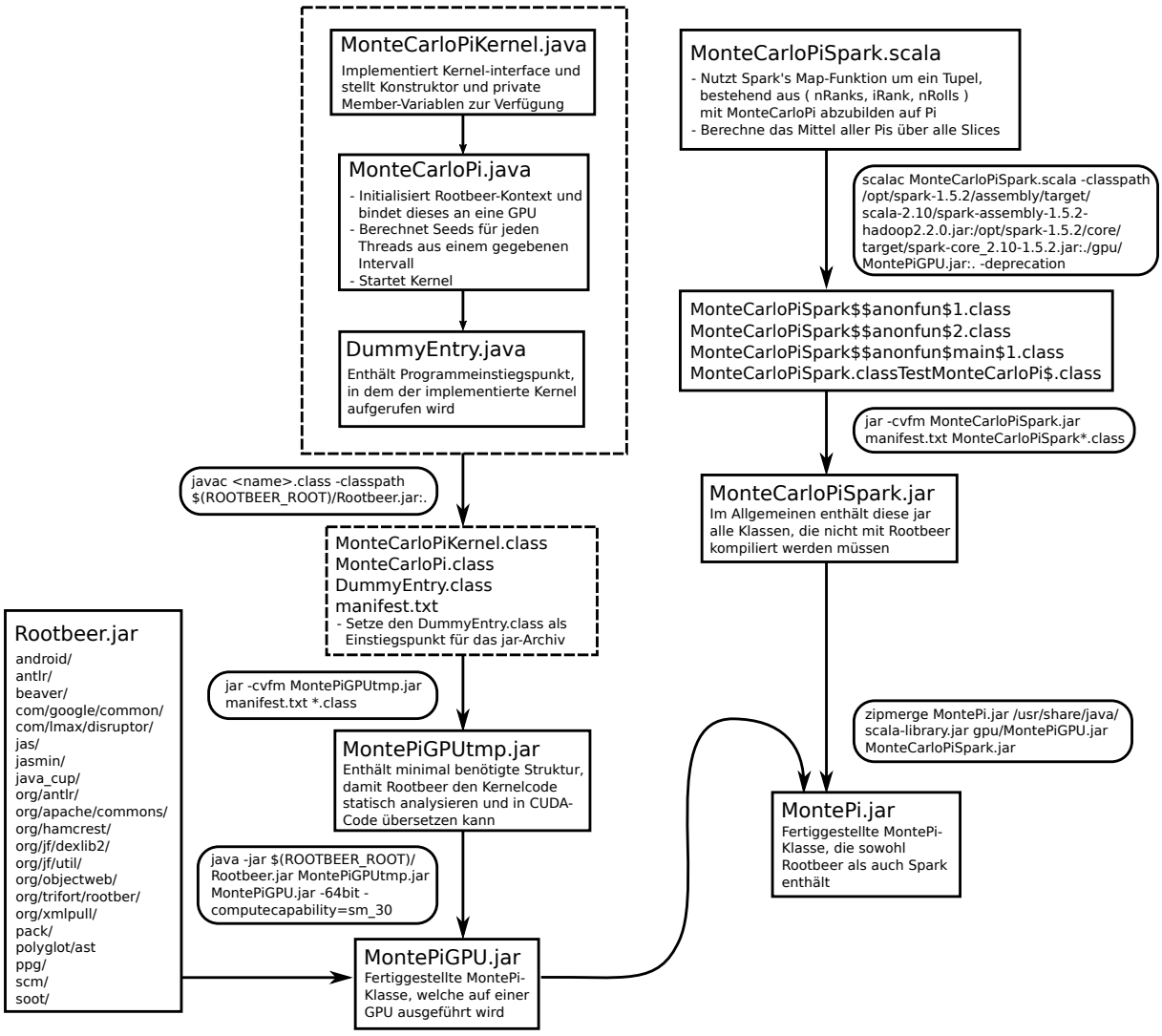

Abbildung 6.1: Kompilationsschema mit Kommadozeilenbefehlen und Zwischenstati.

!!! Problem: Hab mehrere Fehler gefunden deren Kenntnis möglicherweise eine Vereinfachung des Schemas bedeutet. Da bin ich noch am rumspielen, daher ist das halbfertig.

6.2 Probleme

Implementierung: Was ist bei GPUs zu beachten (Seeds, 64-Bit) Was ist bei Rootbeer zu beachten? - private Variablen werden wirklich immer per memcpy hin und her transportiert. -

muss nicht auf ungerade Kernel-Zahl achten, werden automatisch aussortiert

7 Benchmarks

7.1 Testsysteme

7.1.1 System 1: Heimsystem

Aufgrund der leichten Verfügbarkeit und als Beispiel für Grafikkartenbeschleuniger im nicht-professionellen Verbrauchersegment, wurden einige Tests auf einem herkömmlichen Arbeitsplatzrechner ausgeführt:

Prozessor	Intel(R) Core(TM) i3-3220 (Ivy-Bridge), 3.30 GHz, 2 Kerne (4 durch SMT), AVX[2]
Arbeitsspeicher	4×4 GiB DDR3 1600 MHz CL9[6]
Grafikkarte	GigaByte GTX 760 WindForce 3X Overclocked, Codename: GK104-225-A2 (Kepler), 1152 CUDA-Kerne, 1085 MHz (1150 MHz Boost), 2 GiB GDDR5-VRAM mit 1502 MHz PCIe-3.0-x16-Schnittstelle[7, 4, 5]

Tabelle 7.1: Heimsystemkonfiguration

Die Maximalleistung in der Berechnung von Fließkommazahlen einfacher Genauigkeit (SPFLO) im Verhältnis einer Multiplikation zu einer Addition beträgt

$$3.30\,\text{GHz} \cdot 2\,\text{Kerne}\left(1\,\frac{\text{AVX ADD Einheit}}{\text{Kern}} + 1\,\frac{\text{AVX MUL Einheit}}{\text{Kern}}\right) \cdot 8\,\frac{\text{SPFLO}}{\text{AVX Einheit}} \tag{7.1}$$

$$= 105.6\,\text{GSPFLOPS} \tag{7.2}$$

für den Prozessor. Informationen zur Architektur, wie die Anzahl an AVX-Einheiten wurde aus Ref.[20] entnommen.
Die Maximalleistung der Grafikkarte beträgt:

$$1.085\,\text{GHz} \cdot 1152\text{CUDA-Kerne} \cdot 1\frac{\text{FMA-Einheit}}{\text{CUDA-Kern}} \cdot 2\frac{\text{SPFLO}}{\text{FMA-Einheit}} = 2500\,\text{GSPFLOPS} \tag{7.3}$$

Zugegeben, es wurde beim Prozessor gespart und bei der Grafikkarte nicht, aber der Geschwindigkeitsunterschied von 24x begründet dennoch das Interesse daran Grafikkarten zu nutzen, auch wenn es bei Grafikkarten mehr zu beachten gibt, um diese Maximalleistung erhalten zu können.

7.1.2 System 2: Taurus

Für Skalierungstests wurde einer der Hochleistungsrechner der TU-Dresden, ein Bull HPC-Cluster mit dem Namen Taurus, benutzt. Der Bau der ersten Phase von Taurus war 2013 abgeschlossen[22]. Zum Zeitpunkt der Nutzung (2015/2016) waren alle Knoten von Phase 1 schon in die 2015 fertiggestellt[18] Phase 2 integriert wurden[21] und werden nun beide unter dem Namen Taurus zusammengefasst.

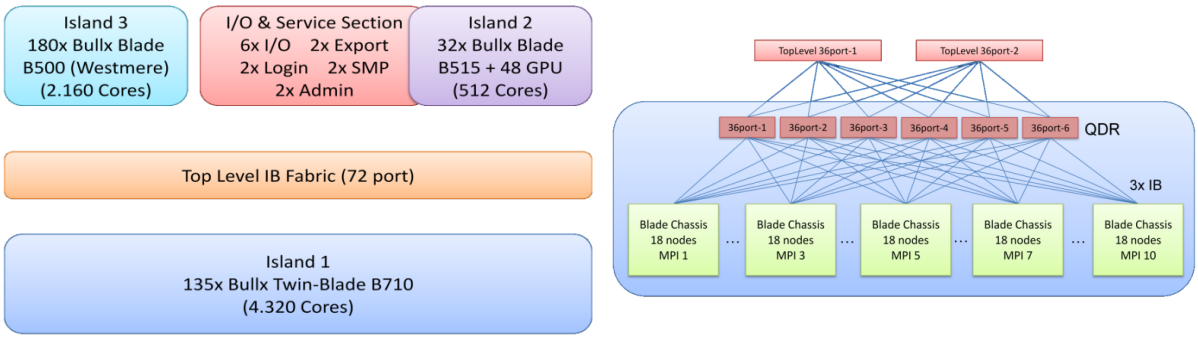

Abbildung 7.1: Links: Übersicht Taurus Phase 1. **Rechts:** Schema der Topologie von Insel 2, auf der ausschließlich gerechnet wurde. Die Bilder wurden übernommen aus Ref.[22]

Gerechnet wurde auf Insel 2 von Taurus, vgl. Tabelle 7.2. Wenn nicht anders erwähnt, dann beziehen sich Benchmarks auf die Tesla K20x Knoten.

	Phase 1	**Phase 2**
Knoten	44	64
Hostnamen	taurusi2[001-044]	taurusi2[045-108]
Prozessor	2x Intel Xeon CPU E5-2450 (8 Kerne) @ 2.10GHz, MultiThreading deaktiviert, AVX, 2x 268.8 GSPFLOPS	2x Intel(R) Xeon(R) CPU E5-2680 v3 (12 Kerne) @ 2.50GHz, MultiThreading deaktiviertt, AVX2 insbesondere FMA3[3], 2x 537.6 GSPFLOPS
GPU	2x NVIDIA Tesla K20x	4x NVIDIA Tesla K80
Arbeitsspeicher	32 GiB	64 GiB
Festplatte	128 GiB SSD	128 GiB SSD

Tabelle 7.2: Zusammensetzung Insel 2 von Taurus[19]

	K20x	**K80**
Chip	GK110	GK210
Takt	0.732 GHz	0.560 GHz
CUDA-Kerne	2688	4992
Speicher	6 GiB, GDDR5 384 Bit Busbreite	2×12 GiB, GDDR5 384 Bit Busbreite
Bandbreite	250 GB s^{-1}	2×240 GB s^{-1}
Theoretische	3935 GSPFLOPS	5591 GSPFLOPS
Spitzenleistung	1312 GDPFLOPS	1864 GDPFLOPS

Tabelle 7.3: Spezifikationen der Kepler-Grafikkarten von Taurus[5, 17]

Zu den Spitzenleistungen in Tabelle 7.3 sei angemerkt, dass jeder CUDA-Kern einfache Fließkommagenaugigkeit berechnet und auf drei CUDA-Kerne eine Doppelpräzisionseinheit kommt, wodurch sich die DFLOPS berechnen. Die K80 hat außerdem einen Boost-Modus mit 0.875 GHz, also einer Leistungssteigerung von 1.56.

7.2 Monte-Carlo-Simulation verschiedener Implementationen

In Abb.7.2 wurde die Ausführungszeit von Monte-Carlo-Simulationen in verschiedenen Programmiersprachen über die Anzahl an Monte-Carlo-Iterationen gemessen. Gemessen wurde die Zeit mit dem Linux `time`-Befehl und zwar die `real`-Zeit. Es fällt auf, dass alle Versionen eine Initialisierungszeit haben. Bei der C++-Version beträgt diese jedoch nur knapp 30 ms, während die reine Java-Version schon ca. 70 ms benötigt. Die Nutzung von Scala erhöht dies schon auf 200 ms und die Nutzung von Rootbeer führt eine weitere Initialisierungszeit von 430 ms ein, sodass für wenig Iterationen die Rootberversion bis zu 20x langsamer sind. Erst für 10 Milliarden Iterationen beginnt die Initialisierungszeit im Vergleich zur Rechenzeit vernachlässigbar zu werden, sodass aber da die Lastenskalierung ein lineares Verhalten annimmt. Bei der C++-Version ist dies schon bei ca. 100 Millionen Iterationen der Fall.

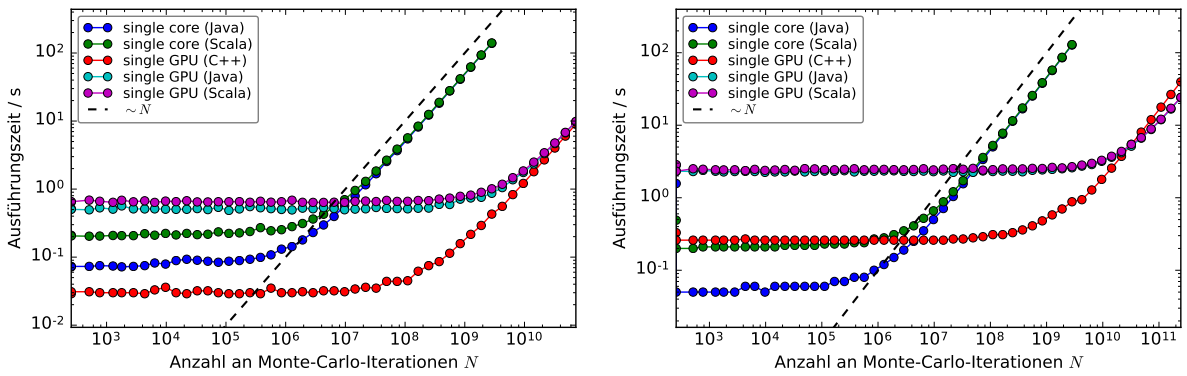

Abbildung 7.2: Benötigte Ausführungszeit der Monte-Carlo Pi-Berechnung in Abhängigkeit von der Anzahl an Iterationen. Getestet auf **links:** System 1 und **rechts:** System 2 (Taurus), siehe Kapitel 7.1.1

Im Programmausdruck 7.1 ist die Hauptschleife, die hauptsächlich die Arbeitslast generiert,

zu sehen. Es handelt sich also für jede der zwei Zufallszahlen um eine Multiplikations und zwei Divisionen und dann nochmals zwei Multiplikationen für die Berechnung des Quadrat des Radius, also zusammen acht Operationen pro Iteration und neun Operationen für Iterationen die im Kreis liegen. Dies tritt für $\frac{\pi}{4} = 0.785\%$ der Fälle auf, das heißt die Rechenlast, definiert als die Anzahl an arithmetischen Operationen N_{Op} sollte sich wie folgt aus der Anzahl an Iterationen N berechnen:

$$N_{\mathrm{Op}} = \left[\frac{\pi}{4} \cdot 9 + \left(1 - \frac{\pi}{4}\right) \cdot 8\right] N = 8.8 \cdot N \tag{7.4}$$

Zuletzt ist aus dem Plot abzulesen, dass für große Lasten wie z.B. für drei Milliarden Iterationen die Versionen, die von Grafikarten gebrauch machen, um einen Faktor 140 (Scala) bis 320 (C++) schneller sind als die Java Version, die auf einem Prozessor-Kern ausgeführt wird. Dieser große Unterschied zwischen CPU und GPU lässt darauf schließen, dass Java weder AVX noch mehrere Kerne gleichzeitig nutzt. Das heißt für die CPU-Versionen wäre für das Testsystem 1 noch ein Geschwindigkeitsgewinn von $8 \frac{\mathrm{Op}}{\text{AVX-Einheit}} \cdot 2$ Kerne erreichbar. Dies würde den Geschwindigkeitsunterschied von 320 auf 20 reduzieren, was in Übereinstimmung mit dem Verhältnis der Peakflops aus Kapitel 7.1.1 wäre.

```
1 for ( int i = 0; i < dnDiceRolls; ++i )
2 {
3     dRandomSeed = (int)( (randMagic*dRandomSeed) % randMax );
4     float x = (float) dRandomSeed / randMax;
5     dRandomSeed = (int)( (randMagic*dRandomSeed) % randMax );
6     float y = (float) dRandomSeed / randMax;
7     if ( x*x + y*y < 1.0 )
8         nHits += 1;
9 }
```

Listing 7.1: Hauptschleife der Monte-Carlo Pi-Berechnung

Dieses Branching verändert also nicht das Skalierverhalten linear mit N, sondern führt nur zu einem veränderten Faktor. Daher ist in Abb.7.2 lineares Verhalten zu beobachten.

7.3 Monte-Carlo-Simulation mit Spark + Rootbeer

In Abbildung 7.3 ist die Laufzeit über die Anzahl an Kernen bzw. Grafikkarten dargestellt. Aus gründen der Rechenzeit wurde für eine Anzahl von $N = 1, 2, 4, 8, 16, 24, 32$ Knoten die Leistungsanalyse für $4(N-1)$ bis $4N$ Kerne/Grafikkarten durchgeführt. Dies ist vor allem in der Abbildung rechts zu sehen, wo die Messpunkte immer in fünfer-Gruppen auftreten.
Zwar nicht in der Abbildung dargestellt, wurde auch für $4N + 1$ und $4N + 2$ gemessen. Das heißt Spark hat zwei mehr Slices zu verarbeiten als es Kerne gibt. Dies führt dazu, dass zwei Kerne doppelt so viel Arbeit wie der Rest haben, wodurch es zu einem sprunghaften Anstieg der Laufzeit kommt. Aus diesem Grund ist es normalerweise besser, wenn die Anzahl an Slices viel größer als die Anzahl an Kernen bzw. Grafikkarten wäre. Dies würde aber die ohnehin schon recht hohe Mindestproblemgröße noch einmal um ein, zwei weitere Größenordnungen erhöhen.

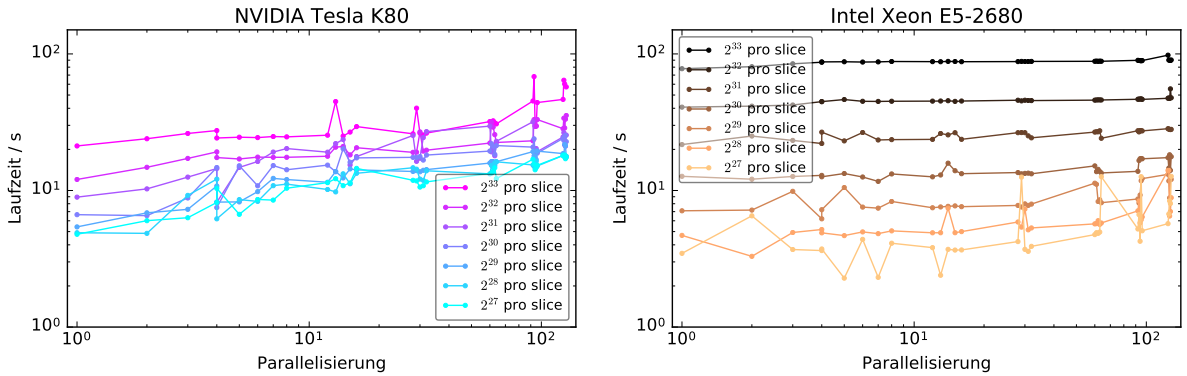

Abbildung 7.3: Benötigte Ausführungszeit der Monte-Carlo Pi-Berechnung in Abhängigkeit von der Anzahl an **links:** Kernen und **rechts:** Grafikkarten. Getestet auf System 2, siehe Kapitel 7.1.2

Weiterhin auffällig sind zufällig auftretende Spitzen in sowohl im CPU als auch im GPU-Benchmark. Z.b. für eine Arbeitslast von 2^33 Iterationen pro Slice auf 24 Knoten also 92 bis 96 Grafikkarten schwankt die Ausführungszeit zwischen 30 s, 45 s und 68 s. Möglicherweise liegt dies an einer ungünstigen Verteilugn der Slices auf die Knoten. Die vorliegende Version nimmt eine lineare Verteilung an, sodass Slice 0 auf Grafikkarte 0 von Knoten 0 rechnet, während Slice 1 auf Grafikkarte 1 von Knoten 0 und Slice 4 auf Grafikkarte 0 von Knoten 1 rechnet. Die Zeit die eine Grafikkarte für diese Last benötigt ist 22 s. Es ist also wahrscheinlich, dass zumindest im Fall der 68 s drei verschiedene Prozesse auf einem Knoten diesselbe Grafikkarte anfordern. Diese Vermutung wurde getestet, indem jeder Slice seinen Hostnamen ausgeben soll. Mit dem Skript aus Listing B.1 ist dies schnell interaktiv getestet:

```
1 startSpark --time=04:00:00 --nodes=5 --partition=west --gres= --cpus-per-task=12
2 spark-shell --master=$MASTER_ADDRESS
3 scala> import java.net.InetAddress
4 scala> sc.parallelize( 1 to 5*12, 5*12 ).map( (x) => { Thread.sleep(10); x+" : "
      +InetAddress.getLocalHost().getHostName() } ).collect().foreach( println )
```

Es ist also wie vermutet: die Verteilung ist nicht linear, sondern eher verzahnt, aber eigentlich zufällig.
Eine Lösung dieses Problems ist schwierig, da die Verteilung der Slices von Spark auf die Worker-Knoten opak erfolgt und es auch schwierig ist mit CUDA und vor allem mit Rootbeer herauszufinden welche der verfügbaren Grafikkarten in Benutzung ist. Ein ändern des Compute-Modus in einen Thread- oder Prozess-exklusiven Modus mittels

```
1 nvidia-smi --compute-mode=EXCLUSIVE_PROCESS
```

ist auf Grund fehlender Berechtigungen im Cluster nicht möglich. In diesem Modus würde der Versuch eine schon in Benutzung seiende Grafikkarte anzusprechen in einer ”GPU device not available”-Fehlermeldung enden. Womöglich ist es gar nicht möglich dies über Rootbeer aus abzufangen.
Die Spitzen im CPU-Benchmark lassen sich dadurch jedoch nicht erklären, da sie für sehr Hohe Arbeitslasten kleiner verschwindet klein werden. Es handelt sich also wahrscheinlich eher um

zufällige Initialisierungsoffsets oder Kommunikationslatenzen. Sie sind ungefähr 3 s groß, womit Kommunikationslatenz sehr unwahrscheinlich sind, da `ping taurusi2063` als Beispiel Latenzen im Bereich von 200 µs misst. Es sei hier angemerkt, dass es in den 343 Testläufen für jeweils CPU und GPU zu zwei Fällen kam, in denen ein Job über das Spark-Web-Interface manuell beendet werden musste, da sie schon mehrere Minuten ohne Fortschritt liefen. Möglicherweise war dies aber auch ein Sympton der GPU-Konflikte pro Knoten.
In Abbildung 7.3 ist für große Arbeitslasten wie zu erwarten ein nahezu konstantes Verhalten über erhöhte Parallelisierung abzulesen. Für kleine Arbeitslasten ist eine schwache monotone Abhängigkeit zu beobachten. Möglicherweise ist dies die Zeit, die eine Reduktion über 32 Knoten länger braucht als z.B. über vier Knoten.
Die Benchmarks wurden für eine Grafikkarte pro Knoten wiederholt. Außerdem wurde leider festgestellt, dass die Benchmarks mit nur 384 Threads pro Grafikkarte ausgeführt wurden. Dies wurde geändert auf eine automatische Bestimmung, die zu ungefähr 20000 Threads führen sollte, womit Pipelining genutzt werden kann, sodass ein Faktor von 30 und mehr an Speedup zu erwarten ist.

8 Zusammenfassung

- Ausblick(Nutzbarkeit, Anwendungsfälle, Deep Learning) - deep learning - Ähnlichkeit (assembling)
- mehr cores

Literaturverzeichnis

[1] Numpy. http://www.numpy.org/ [Online; accessed 2016-05-04],

[2] Corporation, Intel: Intel Core i3-3220 Prozessor Spezifikationen. http://ark.intel.com/de/products/65693/Intel-Core-i3-3220-Processor-3M-Cache-3_30-GHz [Online; accessed 2016-05-07],

[3] Corporation, Intel: Intel Xeon E5-2680v3 Prozessor Spezifikationen. http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz?wapkw=e5-2680v3 [Online; accessed 2016-05-08],

[4] Corporation, NVIDIA: NVIDIA GeForce GTX 760 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-760/specifications [Online; accessed 2016-05-07],

[5] Corporation, NVIDIA: Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf [Online; accessed 2016-05-07], 2014

[6] CORSAIR: Vengeance® - 4GB Single Module DDR3 Memory Kit (CMZ4GX3M1A1600C9) - Tech Specs. http://www.corsair.com/en/vengeance-4gb-single-module-ddr3-memory-kit-cmz4gx3m1a1600c9 [Online; accessed 2016-05-07],

[7] GIGA-BYTE Technology Co., Ltd.: NVIDIA GeForce GTX 760 Specifications. http://www.gigabyte.com/products/product-page.aspx?pid=4663#sp [Online; accessed 2016-05-08],

[8] Group, Sable R. ; Group, Secure Software E.: Soot - A framework for analyzing and transforming Java and Android Applications. https://sable.github.io/soot/ [Online; accessed 2016-05-09],

[9] Lam, Patrick ; Bodden, Eric ; Lhoták, Ondrej ; Hendren, Laurie: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 2011

[10] Marti, Othmar: Mittlerer Fehler des Mittelwertes - Vorlesungsskript. wwwex.physik.uni-ulm.de/lehre/fehlerrechnung/node15.html [Online; accessed 2016-05-07],

[11] Metropolis, Nicholas: The beginning of the Monte Carlo Method. In: Los Alamos Science 15 (1987), Nr. 584, 125–130. http://jackman.stanford.edu/mcmc/metropolis1.pdf

[12] Metropolis, Nicholas ; Ulam, Stanislaw: The Monte Carlo Method. In: Journal of the American statistical association 44 (1949), Nr. 247, S. 335–341

[13] Pratt-Szeliga: Rootbeer GPU Compiler - Java GPU Programming. https://github.com/pcpratts/rootbeer1 [Online; accessed 2016-05-09],

[14] Pratt-Szeliga: Issues with JRE 1.8. https://github.com/pcpratts/rootbeer1/issues/175#issuecomment-61431951 [Online; accessed 2016-05-08], 2014

[15] Pratt-Szeliga, Maximilian K.: Rootbeer GPU Compiler - Java GPU Programming. https://github.com/pcpratts/rootbeer1 [Online; accessed 2016-05-09],

[16] Pratt-Szeliga, Philip C. ; Fawcett, James W. ; Welch, Roy D.: Rootbeer: Seamlessly using gpus from java. In: High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on IEEE, 2012, S. 375–380

[17] Smith, Ryan: NVIDIA Launches Tesla K20 & K20X: GK110 Arrives At Last. http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last [Online; accessed 2016-05-08], 11 2012

[18] Stiller, Andreas: Neuer Supercomputer an der TU-Dresden wird am Mittwoch eingeweiht. http://www.heise.de/newsticker/meldung/Neuer-Supercomputer-an-der-TU-Dresden-wird-am-Mittwoch-eingeweiht-2639880.html [Online; accessed 2016-05-08], 05 2015

[19] Ulf Markwardt, Guido J. u. a.: TU Dresden Internetauftritt - Hochleistungsrechner. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus [Online; accessed 2016-05-08], 2013-2016

[20] Valentine, Bob: Introducing Sandy Bridge. https://www.cesga.es/pt/paginas/descargaDocumento/id/135 [Online; accessed 2014-10-21],

[21] Web-Team, HPC: TU Dresden Internetauftritt - Zentrale Komponenten. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus [Online; accessed 2016-05-08], 2013-2016

[22] Wender, Jan: HRSK-II Nutzerschulung. https://doc.zih.tu-dresden.de/hpc-wiki/pub/Compendium/SystemTaurus/HRSK-II-Nutzerschulung.pdf [Online; accessed 2016-05-08], 5 2014

A Standardabweichung des Mittelwertes

Dieses Kapitel richtet sich leicht nach Ref.[10]. Sei $f \equiv (f_i)$ eine Folge von N Stichproben aus einer Zufallsverteilung mit einem Mittelwert μ und μ_N der empirische Mittelwert dieser Folge. Die Differenz $\Delta_N := \mu - \mu_N$ wird nun abgeschätzt mit der Standardabweichung einer Folge von empirischen Mittelwerten $(\mu_{N,k})$ die alle mit (sehr wahrscheinlich) verschiedenen Folgen bzw. Vektoren (f_i) gebildet seien.
Sei nun $\langle\cdot\rangle_N : \mathbb{S} \times \mathbb{R}^N \to \mathbb{R}^N, \langle (f_i)_k \rangle_N := \frac{1}{N} \sum_{i=0}^{N} f_{ik}$ der empirische Mittelwert und $E(\cdot) : \mathbb{S} \to \mathbb{R}, E\left(x_k\right) := \lim_{N\to\infty} \frac{1}{N} \sum_{k=0}^{N} f_k$ der Erwartungswert über eine unendlich Folge aus einem beliebigen statistischen Werten für begrenzte Folgen (f_i). Hierbei ist $\mathbb{S}$ der Raum der Folgen. Aus der Definition der beiden Mittelwerte wird klar, dass man die Summen und damit die Bildung der Mittelwerte vertauschen kann, sofern ein Grenzwert existiert. Dies wird in Gleichung A.3 angewandt.

$$\sigma_{\mu_N}^2 := E\left(\left(\mu_N - \mu\right)^2\right) := E\left(\left(\langle f_{ik}\rangle_N - \mu\right)^2\right) = E\left(\langle f_{ik} - \mu\rangle_N^2\right) \tag{A.1}$$

$$= E\left(\left(\frac{1}{N}\sum_{i=0}^{N}\left(f_{ik}-\mu\right)\right)\left(\frac{1}{N}\sum_{i=0}^{N}\left(f_{ik}-\mu\right)\right)\right) = E\left(\frac{1}{N^2}\sum_{i=0}^{N}\sum_{j=0}^{N}\left(f_{ik}-\mu\right)\left(f_{jk}-\mu\right)\right) \tag{A.2}$$

$$= \frac{1}{N}E\left(\frac{1}{N}\sum_{i=0}^{N}\left(f_{ik}-\mu\right)^2\right) + E\left(\frac{1}{N}\sum_{i=0}^{N}\left(f_{ik}-\mu\right)\frac{1}{N}\sum_{j=0,j\neq i}^{N}\left(f_{jk}-\mu\right)\right) \tag{A.3}$$

$$= \frac{1}{N}E\left(\sigma_k\right) + \frac{1}{N}\sum_{i=0}^{N}\frac{1}{N}\sum_{j=0,j\neq i}^{N}\left(\underbrace{E\left(f_{ik}\right)}_{=\mu}-\mu\right)\left(\underbrace{E\left(f_{jk}\right)}_{=\mu}-\mu\right) = \underline{\underline{\frac{\sigma}{N}}} \tag{A.4}$$

Man beachte, dass der Schritt in Gl.A.3-A.4 nur möglich ist, wenn f_i unabhängig von f_j ist, was hier der Fall ist, da der einzige abhängige Fall für $i = j$ aus der SUmme rausgezogen würde, sodass $E(ab) = E(a)E(b)$ anwendbar ist.
Man beachte, dass der zweite Summand nur durch die Mittelung über mehrere komplett verschiedene Versuchsreihen Null wird. Betrachtet man jedoch nur eine Versuchsreihe, dann hat der zweite Summand auch ein Skalierverhalten in Abhängigkeit zu N. Da aber das Vorzeichen wechseln kann, muss man den Betrag betrachten:

$$\frac{1}{N}\sum_{i=0}^{N}\left(f_i-\mu\right)\frac{1}{N}\sum_{j=0,j\neq i}^{N}\left(f_j-\mu\right) = \frac{1}{N}\sum_{i=0}^{N}\frac{1}{N}\sum_{j=0,j\neq i}^{N}\left(f_i f_j - \mu\left(f_i+f_j\right)+\mu^2\right) \tag{A.5}$$

$$\approx \mu_N^2 - 2\mu\mu_N + \mu^2 \overset{\mu_N\approx\mu-\sigma_{\mu_N}}{\approx} \mu^2 - 2\sigma_{\mu_N}\mu + \sigma_{\mu_N}^2 - 2\mu^2 + 2\mu\sigma_{\mu_N} + \mu^2 = \sigma_{\mu_N}^2 = \frac{\sigma}{N} \tag{A.6}$$

Schritt A.6 ist stark skizzenhaft und nicht mathematisch korrekt ausgeführt, wird aber gestützt durch empirische Auswertungen, vgl. Abb.A.1. In der Abbildung sieht man, dass sowohl der

erste Summand als auch der zweite invers proportional zu N skaliert. Ein wichtiger Unterschied ist jedoch, dass der erste Summand immer positiv ist, während das Vorzeichen des zweiten Summanden oszilliert, wodurch er über die Mittelung mit $E(\cdot)$ gegen Null geht.
Interessant zu bemerken ist auch, dass der Graph der Standardvarianz des Mittelwertes $\sigma^2_{\mu_N}$ aufgetragen über die Anzahl an einbezogener Stichproben einer Zufallsbewegung ähnelt, anstatt stochastisch zu streuen. Dies wäre nicht der Fall, würde man für alle N komplett neue Stichproben ziehen.
Weiterhin fällt auf, dass beide Summanden einer sehr glatten Geraden mit wenig Streuung folgen, während dies für $\sigma^2_{\mu_N}$ nicht der Fall ist. Dies zeigt, dass es durchaus zu einer Fehlerauslöschung durch den wegdiskutierten zweiten Summanden kommt. Dies beeinträchtigt jedoch nicht die Fehlerskalierung mit $\mathcal{O}\left(\frac{1}{N}\right)$.

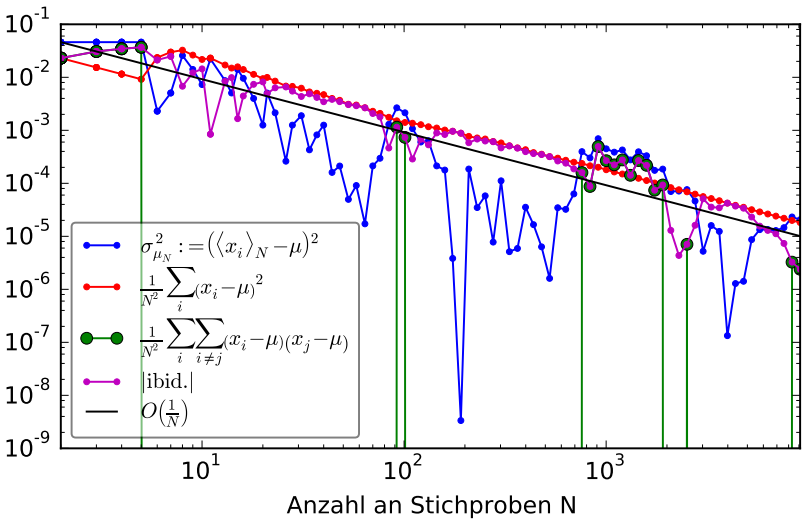

Abbildung A.1: Darstellung der Standardvarianz des Mittelwertes $\sigma^2_{\mu_N}$ und der beiden in der Herleitung A.3 auftretenden Summanden über die Anzahl einbezogener Stichproben. Hierbei ist zu beachten, dass beim Vergleich von den Werten für die Stichprobenanzahl von N_1 und N_2 die ersten $\min(N_1, N_2)$ Stichproben identisch sind.

B Listings

```
#!/bin/bash

# These settings can be overwritten setting by setting by specifying
# 'fresH' command line parameters to sbatch
#SBATCH --account=p_zih-hops
#SBATCH --partition=gpu1
#SBATCH --nodes=3
# ntasks per node MUST be one, because multiple slaves per work does not work with slurm + spark in this script
#SBATCH --ntasks-per-node=1
# CPUs per Task must be equal to gres:gpu or else too few or too much GPUs will
# be used. Partition 'gpu1' on taurus has 2 K20x GPUs per node and 'gpu2' has
# 4 K80 GPUs per node
#SBATCH --cpus-per-task=2
#SBATCH --gres=gpu:2
#SBATCH --mem-per-cpu=1000
# Beware! $HOME will not be expanded and invalid output-URIs will result
# Slurm jobs hanging indefinitely.
#SBATCH --output="/home/s3495379/spark/logs/%j.out"
#SBATCH --error="/home/s3495379/spark/logs/%j.err"
#SBATCH --time=01:00:00

# E.g. use it like this to run MontePi.jar with 8 slices (slice count is
# specified in the program itself, but it was written to accept arguments):
#   startSpark
#   sparkSubmit ~/scaromare/MontePi/multiNode/multiCore/MontePi.jar 1234567890 8 2>/dev/null
# Output could be:
#   Rolling the dice 1234567890 times resulted in pi ~ 3.1416070527180278 and took 7.370468868 seconds
# and when running with only 1 slice( ontePi.jar 1234567890 1 ):
#   Rolling the dice 1234567890 times resulted in pi ~ 3.141646073428979 and took 27.902197481 seconds

realpath() { echo "$(cd "$(dirname "$1")" && pwd)/$(basename "$1")"; }

# This section will be run when started by sbatch
if [ "$1" != 'sran:D' ]; then
{
    # Get path of this script
    this=$0
    if [ ! -f "$this" ]; then
    {
```

```
40          echo "[Note] Can't find calling argument path '$this', trying another
    method"
41          this=$(scontrol show jobid $SLURM_JOBID | grep -i command)
42          if [ -z "$this" ]; then
43              echo "[Warning] Couldn't get path from slurm job info!"
44          elif [ ${this:0:1} != '/' ]; then
45              this=$SLURM_SUBMIT_DIR/$this
46          fi
47          if [ ! -f "$this" ]; then
48              echo "[Note] Can't find SLURM job argument path '$this', trying
    another method"
49              this=$SLURM_SUBMIT_DIR/$(basename "$0")
50              if [ ! -f "$this" ]; then
51                  echo "[Error] Couldn't find path of this script. All methods
    exhausted"
52                  exit 1
53              fi
54          fi
55      }
56      fi
57      # I experienced random problems with the second thread not finding the
    script:
58      #   slurmstepd: execve(): /var/spool/slurm/job6924681/slurm_script: No such
    file or directory
59      #   srun: error: taurusi2029: task 1: Exited with exit code 2
60      if [ ! -d "/scratch/$USER/" ]; then
61          echo "[ERROR] Couldn't find shared directory '/scratch/$USER/'"
62          exit 1
63      fi
64      script=/scratch/$USER/${SLURM_JOBID}_$(basename "$0")
65      cp "$this" "$script"
66      echo "[Father] Working directory  : '$(pwd)'"
67      echo "[Father] Path of this script: '$this'"
68
69      export sparkLogs=$HOME/spark/logs
70      export sparkTmp=$HOME/spark/tmp
71      mkdir -p "$sparkLogs" "$sparkTmp"
72
73      # these variables must be set!
74      export SPARK_ROOT=$HOME/spark-1.5.2-bin-hadoop2.6/
75      export SPARK_JAVA_OPTS+="-Dspark.local.dir=$sparkTmp -XX:+UseParallelGC -XX:
    MaxPermSize=5G"
76      export SPARK_DAEMON_MEMORY=$(( $SLURM_MEM_PER_CPU * $SLURM_CPUS_PER_TASK / 2
     ))m
77      export SPARK_MEM=$SPARK_DAEMON_MEMORY
78      export SPARK_WORKER_DIR=$sparkLogs
79      export SPARK_LOCAL_DIRS=$sparkLogs
80      export SPARK_MASTER_PORT=7077
81      export SPARK_MASTER_WEBUI_PORT=8080
82      export SPARK_WORKER_CORES=$SLURM_CPUS_PER_TASK
83
84      echo "[Father] srun $script 'sran:D' $@"
```

```
85      srun "$script" 'sran:D' "$@"
86      echo "[Father] srun finished, exiting now"
87      exit 0
88  }
89  # If run by srun, then decide by $SLURM_PROCID whether we are master or worker
90  else
91      #echo "SLURM_PROCID = $SLURM_PROCID"
92
93      module load scala/2.10.4 java/jdk1.7.0_25 cuda/7.0.28
94      nvidia-smi
95
96      if [ -z "$SLURM_PROCID" ]; then
97          echo "[Process $SLURM_PROCID] [Error] $SLURM_PROCID is not set, maybe
        srun failed somehow?"
98          exit 1
99      elif [ ! "$SLURM_PROCID" -eq "$SLURM_PROCID" ] 2>/dev/null; then
100         echo "[Process $SLURM_PROCID] [Error] SLURM_PROCID=$SLURM_PROCID is not
        a number!"
101         exit 1
102     elif [ $SLURM_PROCID -eq 0 ]; then
103     {
104         # This does similar things as vanilla $SPARK_ROOT/sbin/start-master.sh
105         # but slurm compatible, e.g. not in daemon-mode
106
107         . "$SPARK_ROOT/sbin/spark-config.sh"
108         . "$SPARK_ROOT/bin/load-spark-env.sh"
109
110         export SPARK_MASTER_IP=$(hostname)
111         MASTER_NODE=$(scontrol show hostname $SLURM_NODELIST | head -n 1)
112         if [ "$MASTER_NODE" != "$SPARK_MASTER_IP" ]; then
113             echo "[Process $SLURM_PROCID] [Error] The method to get the master
        hostname won't work for the worker nodes! (This process is the master and is
        on $(hostname), but method will find '$MASTER_NODE' to be the master.)"
114             exit 1
115         fi
116
117         # This can be used for debugging purposed and/or to find out the WebUI
        address
118         # Furthermore this is necessary to submit jobs to the spark instance!
119         echo "spark://$SPARK_MASTER_IP:$SPARK_MASTER_PORT" > "$sparkLogs/${
        SLURM_JOBID}_spark_master"
120
121         echo "[Process $SLURM_PROCID] Starting Master at spark://
        $SPARK_MASTER_IP:$SPARK_MASTER_PORT (WebUI: $SPARK_MASTER_WEBUI_PORT)"
122         "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.master.Master \
123             --ip $SPARK_MASTER_IP                                          \
124             --port $SPARK_MASTER_PORT                                      \
125             --webui-port $SPARK_MASTER_WEBUI_PORT
126         echo "[Process $SLURM_PROCID] spark master finished, exiting now!"
127     }
128     else
129     {
```

```
130         # This does similar things as vanilla $SPARK_ROOT/sbin/start-slave.sh
    but slurm compatible
131         # scontrol show hostname is used to convert host20[39-40] to host2039
132         MASTER_NODE=spark://$(scontrol show hostname $SLURM_NODELIST | head -n
    1):7077
133         echo "[Process $SLURM_PROCID] Process $SLURM_PROCID starting slave at $(
    hostname) linked to $MASTER_NODE"
134         "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.worker.Worker
    $MASTER_NODE
135         echo "[Process $SLURM_PROCID] spark slave finished, exiting now!"
136     }
137     fi
138 fi
```

Listing B.1: `start_spark_slurm.sh` enthÃ¤lt Parameter fÃ¼r `sbatch`, berechnet nÃ¶tige Parameter fÃ¼r Spark aus denen von Slurm und startet dann einen Master und mehrere Worker-Prozesse startet

```
1  #!/bin/bash
2
3  function startSpark() {
4      export SPARK_LOGS=$HOME/spark/logs
5      mkdir -p "$SPARK_LOGS"
6      if [ ! -d "$SPARK_LOGS" ]; then return 1; fi
7      jobid=$(sbatch "$@" --output="$SPARK_LOGS/%j.out" --error="$SPARK_LOGS/%j.
     err" $HOME/scaromare/start_spark_slurm.sh)
8      jobid=${jobid##Submitted batch job }
9      echo "Job ID : $jobid"
10     # looks like: 16/05/13 20:44:59 INFO MasterWebUI: Started MasterWebUI at
     http://172.24.36.19:8080
11     echo -n "Waiting for Job to run and Spark to start.."
12     MASTER_WEBUI=''
13     while [ -z "$MASTER_WEBUI" ]; do
14         echo -n "."
15         sleep 1s
16         if [ -f $HOME/spark/logs/$jobid.err ]; then
17             MASTER_WEBUI=$(sed -nE 's|.*Started MasterWebUI at (http://[0-9.:]*)
     |\1|p' $HOME/spark/logs/$jobid.err)
18         fi
19     done
20     echo "OK"
21     export MASTER_WEBUI
22     export MASTER_ADDRESS=$(cat ~/spark/logs/${jobid}_spark_master)
23     function sparkSubmit() {
24         ~/spark-1.5.2-bin-hadoop2.6/bin/spark-submit --master $MASTER_ADDRESS $@
25     }
26     cat "$SPARK_LOGS"/$jobid.*
27     echo "MASTER_WEBUI    : $MASTER_WEBUI"
28     echo "MASTER_ADDRESS : $MASTER_ADDRESS"
29 }
```

Listing B.2: `startSpark.sh` welche eine Funktion definiert die einen den Sparkslurmjob aus Listing B.1 startet und Master-IP aus der Logdatei extrahiert