FHZ → FACHHOCHSCHULE ZENTRALSCHWEIZ

HTA → HOCHSCHULE FÜR TECHNIK+ARCHITEKTUR LUZERN

Abteilung Informatik

Java

Thread Skalierung

SDD

Software Design Document

HTA Horw

HTA LUZERN
Technikumstrasse 21
CH—6048 Horw

T: 041-349-33-11
F: 041-349-39-60
W: www.hta.fhz.ch

Diplomarbeit 2006
Aregger Marcel
Meier Rainer

Änderungskontrolle

Version	Datum	Ausführende Stelle	Bemerkungen/Art der Änderung
1.1	2006-10-16	Rainer Meier	Initial Release
1.2	2006-11-06	Rainer Meier	Diverse Erweiterungen in allen Kapiteln. Erste freigegebene Draft-Version
1.3	2006-11-10	Rainer Meier	Fraktal ausgewählt., Source-Code dokumentiert Anpassungen Review Evaluation
1.4	2006-11-14	Marcel Aregger	Zuordnung Testumfang zu Projektzielsetzungen

Prüfung und Freigabe

Vorname/Name	Dokumentversion	Status	Datum	Visum
Rainer Meier	1.4	Final	2006-11-16	
Marcel Aregger	1.4	Final	2006-11-16	

1. Management Summary

Das Software Design Document bildet das Bindeglied zwischen der Basisanalyse und der darauf folgenden Testserie. Das Dokument ist folgendermassen strukturiert:

1. Evaluation Plattform
2. Abstecken des geplanten Testumfanges
3. Spezifikation der Testklasse
4. Auswahl der Testklasse
5. Implementierung der Testapplikation

Die Evaluation der Testplattform (siehe Kapitel 5) ist nötig um die aus der Basisanalyse (siehe [1]) resultierenden Einflussbereiche abzugrenzen. Daraus resultieren die Anforderungen an die Testplattform auf Ebene Hardware, Betriebssystem, Applikation und JVM:

Tabelle 1 Abgrenzung der Einflussbereiche

Hardware	Betriebssystem	Applikation	JVM
SMP	Designprinzip Thread, Win32 Thread, Scheduling (Prioritäten), Affinität	OpenMP (in Form von JOMP)	Java Threading, JOMP

Die Definition des geplanten Testumfanges (siehe Kapitel 6) ist notwendig um die Anforderungen an eine geeignete Testklasse definieren zu können da diese direkt vom geplanten Testumfang abhängig ist. Hier wurden 8 Testziele {T?} formuliert die in direktem Zusammenhang mit den im SPMP definierten Projektzielsetzungen {Z?} stehen.

Tabelle 2 Übersicht Testziele

SPMP	#	Zielsetzung Hardware, Betriebssystem und JVM	Kat.
{Z19}	{T1}	Feststellung Grad der Skalierung zwischen Single- und Multi-Prozessor-Architektur	Muss
{Z7}	{T2}	Nachweis Abbildung Java-Thread auf Win32-Thread	Muss
{Z7}	{T3}	Nachweis Abbildung Java-Thread-Priorität auf Win32-Thread-Priorität	Kann
{Z7}	{T4}	Analyse Systemverhalten bei Änderung Win32-Thread-Priorität	Kann
{Z7}	{T5}	Analyse Systemverhalten bei Festlegung einer Prozess-Affinität	Kann
{Z5}	{T6}	Analyse der Skalierung einer multithreaded Java-Applikation	Muss
{Z5}	{T7}	Analyse Einfluss der Thread-Synchronisation auf die Skalierung	Kann
{Z16}	{T8}	Analyse Anwendbarkeit und Effektivität von JOMP	Muss

Die aus den Testzielen abgeleitete Spezifikation der Testklasse (siehe Kapitel 7) beinhaltet einen Anforderungskatalog in 8 Punkten. Sie definieren die funktionalen- und logischen Anforderungen an die Testklasse.

Tabelle 3 Anforderungskatalog an eine geeignete Testklasse

#	Beschreibung	Typisierung
{R1.1}	Sequenzieller Code	Muss
{R1.2}	Parallelisierbar	Muss
{R1.3}	Messbar	Muss
{R1.4}	Portierbar auf verschiedene Konzepte und Plattformen	Muss
{R1.5}	Nachvollziehbar, einfach, übersichtlich	Muss
{R1.6}	Reproduzierbarkeit	Muss
{R2.1}	Konkurrierender Datenzugriff	Kann
{R2.2}	Visualisierbarkeit	Kann

Kapitel 8 beinhaltet die Evaluation einer geeigneten Testklasse anhand des Anforderungskataloges inklusive technischer Analyse des Algorithmus. Die Berechnung der Mandelbrot Menge stellte sich dabei als optimal heraus da sich alle Anforderungen mit diesem Algorithmus optimal abdecken lassen.

Basierend auf der ausgewählten Testklasse wurde in Kapitel 9 ein Anwendungs-Design erstellt und dokumentiert, welches eine bequeme Durchführung der Testreihen erlaubt. Das Design beinhaltet insbesondere die Lebenszyklus-Verwaltung der Threads sowie die Einbindung verschiedener Locking Techniken und der JOMP Technologie.

2. Inhaltsverzeichnis

3. Dokumentinformationen

3.1. Referenzierte Dokumente

Tabelle 4 Referenzierte Dokumente

Referenz	Beschreibung
[1]	Basisanalyse
[2]	Software Test Document (STD)
[3]	Software Project Management Plan (SPMP)

3.2. Definitionen und Abkürzungen

Tabelle 5 Abkürzungen

Abkürzung	Beschreibung
{R?}	Requirement einer Testklasse
{T?}	Zielsetzung geplanter Testumfang
{Z?}	Projektzielsetzung SPMP
API	Application Programming Interface
CAS	Compare-and-swap
CMP	Chip Multi Processing
CMT	Chip Multi Threading
CVS	Concurrent Versioning System
HW	Hardware
JOMP	Java OpenMP
JVM	Java Virtual Machine
MPI	Message Passing Interface
MVC	Model View Control
SDD	Software Design Document
SMP	Symmetric Multi Processing
STD	Software Test Document
TBB	Thread Building Blocks
UMA	Uniform Memory Architecture
UMA	Uniform Memory Access

3.3. Links

Tabelle 6 Links

Referenz	Beschreibung
[FRAKTAL]	Wikipedia, Fraktal: http://de.wikipedia.org/wiki/Fraktal
[JAVAMANDELBROT]	Java Mandelbrot Fraktal Renderer: http://www.aasted.org/fractal/
[JOMP]	EPCC, OpenMP-like directives for Java: http://www.epcc.ed.ac.uk/research/jomp/
[MANDELBROT]	Wikipedia, Mandelbrot-Menge: http://de.wikipedia.org/wiki/Mandelbrot-Menge
[OPENMP]	OpenMP, Homepage: http://www.openmp.org/

4. Einleitung

4.1. Zweck des Dokumentes

Übergeordnete Zielsetzung dieses Software Design Documents (SDD) ist die Definition einer geeigneten Testplattform. Auf dieser Plattform sollen mögliche Testfälle definiert werden, welche die verschiedenen Aspekte der Basisanalyse aufgreifen und konkret umsetzten. Ausgehend von dieser Plattform und den Testfällen definiert dieses Dokument Testklassen, die eine Umsetzung verschiedener Konzepte zulassen und in Bezug auf deren Skalierung getestet werden können.

Die Evaluation der (Test)Plattform bestehend aus HW, Technologien, Konzepten und Standards ist ebenfalls zentraler Bestandteil dieses Dokuments. Sie orientiert sich an den Einflussbereichen und Technologien aus der Basisanalyse und berücksichtigt dabei Faktoren wie die themenbezogene Fokussierung der Diplomarbeit oder die (potenzielle) Wirkung von Aspekten auf die Skalierung.

Mit der Absicht vor der eigentlichen Definition und Implementierung von Testklassen den „planned scope“ der ganzen Testphase abzugrenzen, werden in diesem Dokument auch konkrete Zielsetzung und Betrachtungsbereiche definiert. Sie beschreiben, was Gegenstand der Testserie sein soll bzw. welche Aspekte fokussiert werden sollen. Zielsetzungen und Betrachtungsbereiche im Testumfang können als „Guideline“ betrachtet werden für die Auswahl und Implementierung von Testklassen und Konzepten. Weiter sind sie Ausgangspunkt für die Erarbeitung der Testcases im STD ([2]).

Der SDD definiert mit Bezug auf die oben genannten Betrachtungsbereiche logisch- und technische Anforderung an eine Testklasse. Die Evaluation einer oder mehrerer geeigneter Klassen/Funktionen realisiert diese Requirements und bietet die Möglichkeit, die oben definierten Zielsetzungen über entsprechende Testcases abzudecken.

Eine kritische Auseinandersetzung mit den ausgewählten Testklassen zeigt Schlüsselstellen im Code in Bezug auf die Einflussbereiche der Skalierung. Sie soll das Verständnis der „Basisimplementierung“ fördern und zeigen wo Konzepte wie bspw. Java Threading, Locking oder CAS umgesetzt werden können.

Der SDD in der vorliegenden Form definiert mit Anforderungen an die Testplattform, geplanten Zielsetzungen/Betrachtungsbereichen und Spezifikation/Umsetzung von Testklassen das eigentliche „Design“ des Testings. Die effektive Umsetzung erfolgt nachfolgend im STD.

5. Evaluation Plattform

Die Plattform auf der die effektive Skalierung einer multithreaded Java-Applikation getestet bzw. nachgewiesen werden soll muss nach der Basisanalyse nun festgelegt werden. Die Auswahl und Definition von Technologien, Konzepten und Standards die diese Gesamtplattform charakterisieren, erfolgt wiederum auf den Layern Hardware, Betriebssystem, Applikation und JVM.

Ziel dieser Definition ist die Reproduzierbarkeit durchgeführter Testreihen. Die Nachvollziehbarkeit der Testresultate und Schlussfolgerungen wird durch Transparenz in der verwendeten Plattform ebenfalls sichergestellt. Die Evaluation wird aus der Basisanalyse abgeleitet und erfolgt in 3 Schritten:

Schritt 1; Technologien Basisanalyse

In Form einer Zusammenfassung definiert die Basisanalyse für jeden Layer Technologien, Konzepte oder Standards mit direktem oder indirektem Einfluss auf die Aufgabenstellung (siehe Kapitel „Auswirkung auf die Aufgabenstellung“). Sie bilden die themenbezogene Grundlage für die Festlegung der zukünftigen Testplattform.

Schritt 2; Einfluss auf Skalierung

Die Einflussbereiche pro Thema werden auf deren Wirksamkeit analysiert und bewertet. Der Grad der Beeinflussung auf die (mögliche) Skalierung einer Applikation wird abgeschätzt und entscheidet letztendlich darüber, ob ein Einflussfaktor für die Testplattform Relevanz hat oder nicht.

Schritt 3; Anforderung an Testplattform

Aus dem Subset der Faktoren die einen „starken“ Einfluss auf die Skalierung ausüben, werden jene ausgewählt, die im Rahmen der Arbeit umgesetzt werden können. Die Realisierbarkeit wird dabei beeinflusst durch Rahmenbedingungen wie Aufgabenstellung, Zeit, Aussagekraft, Verfügbarkeit, etc. Die ausgewählten Faktoren bilden die Anforderungen an die zu realisierende Testplattform.

Punktuell werden auch Faktoren berücksichtigt, die einen geringeren Einfluss auf die Skalierung ausüben oder nur indirekt angewendet werden können (beispielsweise Prioriät von Kernel-Level-Threads). Sofern der Verlauf der Arbeit eine Umsetzung zulässt, werden sie in einzelnen Testfällen mitberücksichtigt bzw. eingearbeitet.

5.1. Hardware

Tabelle 7 Abgrenzung Hardware

Technologien Basisanalyse	Einfluss auf Skalierung		Anforderung an Testplattform
	Stark	Schwach	
SMP	SMP		SMP
CMP	CMP		
CMT	CMT		
UMA		UMA	
NUMA		NUMA	
Skalar/Superskalar		Skalar/Superskalar	
Pipeline		Pipeline	

Begründung Skalierung

Die technologischen Ansätze können grob differenziert werden in Technologien die eine Verteilung von Prozessen/Instruktionen oder die Effizienz deren Verarbeitung fokussieren. Beide Bereiche er-

möglichen die Skalierung einer Anwendung indem die Instruktionen insgesamt schneller abgearbeitet werden. Die effektive Verteilung steht für diese Arbeit aber im Vordergrund.

Starker Einfluss

Symmetric Multi Processing (SMP) mit 2 oder mehreren Prozessoren und Chip Multi Processing (CMP) die Multi-Core Architektur mit physikalisch getrennten Kernen im gleichen Chip-Gehäuse ermöglichen die physische Verteilung von Prozessen und Threads. Mit mehreren zu Verfügung stehenden Recheneinheiten bieten sie die Grundlage für eine „echte“ Parallelisierung von multithreaded Applikationen. Die Chip Multi Threading-Technologie (CMT) unterstützt die parallele Abarbeitung in dem Sinne, dass pro Taktzyklus und Thread eine Instruktion gelesen werden kann.

Schwacher Einfluss

Uniform Memory Access (UMA) und Non-Uniform Memory Access (NUMA) Architekturen definieren die Art und Geschwindigkeit von Speicherzugriff für die jeweiligen CPUs. Ihre charakteristischen Eigenschaften haben in Bezug auf die Cache-Synchronisierung primär Einfluss auf die Verarbeitungsgeschwindigkeit von Prozessen/Instruktionen. Pipelines und Superskalare-Prozessoren fokussieren die Auslastung eines Prozessors bzw. die Optimierung des Durchsatzes. Sie sind für die Skalierbarkeit im Kontext der Aufgabestellung von geringerer Bedeutung.

Begründung Testumgebung

Die Verfügbarkeit von Hardware-Plattformen für diese Diplomarbeit ist limitiert. Ursprüngliche Zielsetzung war der Einsatz mehrerer Plattformen (SMP, CMP, CMT) um das Verhalten plattformübergreifend zu untersuchen. Für die zukünftige Testplattform steht aktuell eine SMP-Maschine zu Verfügung. Referenzplattform bildet eine Single-CPU-Maschine.

Tabelle 8 Hardwareplattform

SMP-Maschine	
Anzahl Prozessoren	2
Prozessor Typ	AMD Opteron 2GHz
Anzahl physische Cores	1 (pro Prozessor)
Hyperthreading	Nein
L1 Cache	128 kB
L2 Cache	1024 kB
L1 Data Cache	64 kB
L1 Instruction Cache	64 kB
AMD64 Architektur	Ja

Die Single-CPU Maschine kann durch die Angabe des `/numprocs=1` Parameters in `c:\boot.ini` simuliert werden. Dies bietet insbesondere den Vorteil, dass die Plattform (Hardware/Hintergrundprozesse) vergleichbar ist. Die Messwerte sind also direkt miteinander vergleichbar.

5.2. Betriebssystem

Tabelle 9 Abgrenzung Betriebssystem

Technologien Basisanalyse	Einfluss auf Skalierung		Anforderung an Testplattform
	Stark	Schwach	
Designprinzip Win32 Thread Scheduling Affinität	Designprinzip Win32 Thread	 Scheduling Affinität	Designprinzip Win32 Thread Scheduling (Prioritäten) Affinität

Begründung Skalierung

Der Fokus im Bereich Betriebssystem lag auf der Art und Weise wie Prozesse und Threads unter Windows XP verwaltet bzw. auf Systemressourcen verteilt werden. Die Skalierung auf Layer Betriebssystem umfasst daher die Themenbereiche Threads (als Designprinzip), Win32 Thread, Scheduling und die Affinität.

Starker Einfluss

Die Verwendung von Threads als Design-Prinzip ist gegeben um überhaupt eine Verteilung auf verschiedene Kerne zu ermöglichen. Da Windows XP mit dem Win32 Thread einen Kernel-Level-Thread implementiert, ist die Verteilung auf Level Betriebssystem realisierbar und nachvollziehbar. Diese 1:1-Abbildung eines Java-Threads auf einen Win32 Thread wird über die entsprechende JVM-Implementierung sichergestellt.

Schwacher Einfluss

Das „priority-driven"-Scheduling unter Windows XP erfolgt auf Level Threads und ist gesteuert über die Basis-Priorität dieser Threads. Der Festlegung von Prioritäten kommt in diesem Zusammenhang eine grosse Bedeutung zu. Sie kann über die Win32-API direkt, vom Java-Entwickler aber nur indirekt über die Priorität der Java-Threads beeinflusst werden. Weiter besteht die Möglichkeit mit Systemtools Prioritäten zur Laufzeit zu ändern um die Auswirkung auf die Skalierung zu untersuchen.

Die Affinität, eine explizite Zuordnung von Prozess und Prozessor kann unter Windows XP auf Level Prozess oder Threads erfolgen. Sie ist wiederum über die Win32-API oder entsprechende Systemtools steuerbar. Die indirekte Einflussnahme und die Tatsache, dass die Funktionen der Win32-API in dieser Arbeit nicht genutzt werden, führen zu dieser Klassifikation.

Begründung Testumgebung

Das Designprinzip Thread soll dahingehend umgesetzt werden, dass mit der Auswahl der „richtigen" JVM die 1:1-Abbildung (Java- auf Win32-Thread) sichergestellt ist. Die Wirkung von Thread-Prioritäten auf Level Java soll in Kombination mit manueller Änderung durch Systemtools ebenfalls untersucht werden. Es soll weiter gezeigt werden, wie die Affinität über Systemtools beeinflussbar ist und welche Wirkungen daraus resultieren.

5.3. Applikation

Tabelle 10 Abgrenzung Applikation

Technologien Basisanalyse	Einfluss auf Skalierung		Anforderung an Testplattform
	Stark	Schwach	

POSIX Threads OpenMP TBB MPI	POSIX Threads OpenMP TBB MPI		(OpenMP)

Begründung Skalierung

Techniken und Standards welche die Parallelität unterstützen oder aus diesem Themenbereich heraus entwickelt wurden gibt es einige. Die Basisanalyse hat mit POSIX Threads, OpenMP, TBB oder MPI aktuelle Themen aufgezeigt. Sie haben alle in Bezug auf die Skalierung eine sehr grosse Bedeutung, für die weiterführende Analyse sind sie allerdings weniger wichtig.

Begründung Testumgebung

Im Rahmen der Aufgabenstellung interessiert primär die Umsetzung eines Konzeptes auf Basis einer Java-Umgebung. Hiermit scheiden alle Verfahren ausser OpenMP aus, das in Form des JOMP Projekt für Java umgesetzt wurde.

5.4. JVM

Tabelle 11 Abgrenzung JVM

Technologien Basisanalyse	Einfluss auf Skalierung		Anforderung an Testplattform
	Stark	Schwach	
Java Threading JOMP JVM Optimierung	Java Threading JOMP	JVM Optimierung	Java Threading JOMP

Begründung Skalierung

Der Java-Entwickler besitzt mit der Java-API ein hilfreiches Interface für die parallele Programmierung. Threads sind integraler Bestandteil dieser API die implizit Funktionalität für die Verwaltung und Synchronisation von Threads bietet. Weiter besteht die Möglichkeit über Parameter das Verhalten der JVM zu beeinflussen. Letzteres ist aber eher als Feintuning zu verstehen.

Starker Einfluss

Der Umfang der Java-API in Bezug auf Threads und Synchronisierung von Threads soll für die Implementierung voll ausgeschöpft werden. Basis bilden hier die verfügbaren Packages aus Java-5.

JOMP ist die spezifische Umsetzung vom OpenMP-Standard auf Java und dient der semi-automatischen Parallelisierung von Java-Anwendungen. Es ist ein Werkzeug der Parallelisierung deren Wirksamkeit getestet werden soll.

Schwacher Einfluss

Die Optimierung der JVM in Bereichen wie JIT-Compiler oder Garbage Collection ist als Feintuning zu verstehen und wird die Skalierung nicht im Bereich von Faktoren beeinflussen. Sie wird darum für Implementierung und Test sekundären Charakter haben.

Begründung Testumgebung

Schwerpunkt und Zielsetzung der Arbeit ist u.a. die Implementierung in Java. Die Verwendung von Java-Threads aus der Java-API ist dadurch gegeben. Weiter bietet sich die JOMP Implementierung als (zukünftigen) Standard für diesen Themenbereich geradezu an. Die Anwendbarkeit soll im praktischen Test ebenfalls geprüft werden.

6. Geplanter Testumfang

Die Skalierung einer Applikation kann auf verschiedenen Ebenen wie beispielsweise Hardware, Betriebssystem oder JVM betrachtet bzw. beeinflusst werden. Mögliche (realisierbare) Einflussbereiche innerhalb dieser Ebenen im Zusammenhang mit der Fokussierung von Java und der vorliegenden Arbeit sind begrenzt und wurden im Kapitel 5 „Evaluation Plattform“ definiert. Die daraus resultierende „Testumgebung“ legt dabei die Plattform fest, mit der die nachfolgend beschriebenen Testbereiche untersucht werden sollen.

Der geplante Testumfang mit Zielsetzungen und Betrachtungsbereichen konkretisiert die unter „Evaluation Plattform“ getätigte Abgrenzung. Die Zielsetzungen definieren dabei, was in der jeweiligen Ebene untersucht werden soll um eine Aussage über die Skalierung machen zu können. Eine Zielsetzung fokussiert den Einfluss dieser Ebene oder ein Teilbereich aus dieser Ebene (Technologien, Konzepte, Standards) auf die Skalierung einer Applikation. Die Zielsetzungen {T?} sind jeweils einer Projektzielsetzung {Z?} aus dem SPMP (Kapitel 4.1; Ziele und Prioritäten) logisch zugeordnet.

Der oder die Betrachtungsbereiche einer Zielsetzung verfeinern diese weiter und geben Hinweise welche Bereiche fokussiert werden müssen um die Zielsetzungen entsprechend umzusetzen. Zielsetzungen und zugehörige Betrachtungsbereiche sind die Basis für die Auswahl und Implementierung von geeigneten Testklassen.

Für alle Betrachtungsbereiche und ausgewählten Testklassen werden im STD Testcases abgeleitet sowie Testparameter (Performance-Indikatoren) und Testtools definiert.

6.1. Hardware

Eine oder mehrere Testklassen werden auf einer Multi-Prozessor-Architektur (hier SMP) ausgeführt um die Verteilung von Threads bzw. die Skalierung zu untersuchen. Der Grad der Skalierung wird durch eine Referenzmessung auf einer Single-Prozessor-Architektur ermittelt.

Tabelle 12 Zielsetzung Hardware

SPMP	#	Zielsetzung Hardware	Kat.
{Z19}	{T1}	Feststellung Grad der Skalierung zwischen Single- und Multi-Prozessor-Architektur	Muss

Tabelle 13 Betrachtungsbereiche Hardware

#	Betrachtungsbereiche Hardware	Kat.
{T1.1}	Testklasse(n) auf Single Prozessor Maschine	
	Eine oder mehrere Testklassen (Single-Threaded; 1 Thread) soll(en) auf einer Single-Prozessor-Architektur ausgeführt werden um dabei Berechnungszeit und Ressourcenbedarf zu ermitteln	Muss
	Eine oder mehrere Testklassen (Multi-Threaded; 2 Threads) soll(en) auf einer Single-Prozessor-Architektur ausgeführt werden um dabei Berechnungszeit und Ressourcenbedarf zu ermitteln	Muss
{T1.2}	Testklasse(n) auf Multi Prozessor Maschine (SMP)	
	Eine oder mehrere Testklassen (Single-Threaded; 1 Thread) soll(en) auf einer Multi-Prozessor-Architektur ausgeführt werden um dabei Berechnungszeit und Ressourcenbedarf zu ermitteln	Muss
	Eine oder mehrere Testklassen (Multi-Threaded; 2 Threads) soll(en) auf einer Single-Prozessor-Architektur ausgeführt werden um dabei Berechnungszeit und	Muss

	Ressourcenbedarf zu ermitteln	
{T1.3}	Direkter Vergleich der Plattformen	
	Darstellung der Leistungsindikatoren beider Plattformen. Berechnung des Skalierungs-Faktors auf Basis Single-Threads	Muss
	Darstellung der Leistungsindikatoren beider Plattformen. Berechnung des Skalierungs-Faktors auf Basis Multi-Threads	Muss

6.2. Betriebssystem

Im Bereich Betriebssystem muss der Nachweis erbracht werden, wie Java-Threads auf Win32-Threads durch die JVM abgebildet werden. Dieser Nachweis schliesst die Priorität von Threads mit ein, weil Scheduling-Entscheidungen unter Windows durch diese Grösse beeinflusst werden. Das Systemverhalten in Bezug auf die Skalierung kann dann durch direkte oder indirekte Änderung von Prioritäten untersucht werden. Die manuelle Zuweisung eines Prozessors auf Level Prozess oder Thread (Affinität) und deren Auswirkung kann ebenfalls getestet werden.

Tabelle 14 Zielsetzungen Betriebssystem

SPMP	#	Zielsetzungen Betriebssystem	Kat.
{Z7}	{T2}	Nachweis Abbildung Java-Thread auf Win32-Thread	Muss
{Z7}	{T3}	Nachweis Abbildung Java-Thread-Priorität auf Win32-Thread-Priorität	Kann
{Z7}	{T4}	Analyse Systemverhalten bei Änderung Win32-Thread-Priorität	Kann
{Z7}	{T5}	Analyse Systemverhalten bei Festlegung einer Prozess-Affinität	Kann

Tabelle 15 Betrachtungsbereiche Betriebssystem

#	Betrachtungsbereiche Betriebssystem	Kat.
{T2.1}	JVM-Implementierung	
	Evaluation einer JVM mit Native-Thread-Unterstützung (gemäss Spezifikation)	Muss
{T2.2}	Testklasse und Nachweisverfahren	
	Definition geeignete(s) Testklasse und Verfahren um Abbildung von Java- auf Win32-Thread sichtbar/nachvollziehbar zu machen	Muss
{T2.3}	Nachweis Thread-Abbildung	
	Ausführen ein oder mehrerer Testklasse(n) um Thread-Abbildung durch die JVM im Betriebssystem sichtbar zu machen	Muss
{T3.1}	Default Priorität Java- und Win32-Thread	
	Java Default-Priorität und deren Abbildung auf die Win32-Thread-Priorität analysieren und dokumentieren	Kann
{T3.2}	Änderung Java (Default)Priorität	
	(Default)Priorität eines Java-Threads dynamisch ändern und deren Abbildung auf die Win32-Thread-Priorität analysiert und dokumentieren	Kann
{T3.3}	Prioritätsbereich Java	
	Abbildung Prioritätsbereich Java-Thread (1...5...10) auf den Prioritätsbereich	Kann

	eines Win32-Thread analysieren und dokumentieren	
{T4.1}	Systemverhalten mit direkter Änderung Priorität	
	Direkte Änderung der Win32-Thread-Priorität (Systemtools) und deren Auswirkung auf die Skalierung analysieren	Kann
{T4.2}	Systemverhalten mit indirekter Änderung Priorität	
	Indirekte Änderung der Win32-Thread-Priorität (Java-Thread) und deren Auswirkung auf die Skalierung analysieren	Kann
{T5.1}	Systemverhalten mit Festlegung Thread-Affinität	
	Direkte Festlegung einer Thread-Affinität (Systemtools) und deren Auswirkung auf die Skalierung analysieren	Kann
{T5.2}	Systemverhalten mit Festlegung Prozess-Affinität	
	Direkte Festlegung einer Prozess-Affinität (Systemtools) und deren Auswirkung auf die Skalierung analysieren	Kann

6.3. Applikation

Auf applikatorischer Ebene erfolgen direkt keine Implementierung und Tests. Der OpenMP-Standard als Werkzeug der parallelen Programmierung wird auf Ebene JVM über die Betrachtung von JOMP berücksichtigt.

6.4. JVM:

Im Bereich der Java Virtual Machine (JVM) soll die Java-API mit ihren Packages und Funktionen dazu benutzt werden um multithreaded Java-Applikationen zu schreiben und auszuführen. Dabei soll der mögliche Einfluss der Synchronisation mehrerer Threads genauer untersucht werden. Die Anwendbarkeit und Skalierung einer JOMP-Anwendung soll hier ebenfalls Teil der Analyse sein.

Tabelle 16 Zielsetzungen JVM

#	#	Zielsetzungen JVM	Kat.
{Z5}	{T6}	Analyse der Skalierung einer multithreaded Java-Applikation	Muss
{Z5}	{T7}	Analyse Einfluss der Thread-Synchronisation auf die Skalierung	Kann
{Z16}	{T8}	Analyse Anwendbarkeit und Effektivität von JOMP	Muss

Tabelle 17 Betrachtungsbereiche JVM

#	Betrachtungsbereiche JVM	Kat.
{6.1}	Testklasse(n) mit n Threads auf n-Prozessor-Architektur	
	Eine oder mehrere Testklassen mit n Threads sollen auf einer Multi-Prozessor-Architektur ausgeführt werden um dabei Laufzeit, Ressourcenbedarf und Verwaltungsaufwand zu ermitteln	Muss
	Gleiche Testklasse(n) mit n Threads sollen auf einer Single-Prozessor-Architektur ausgeführt werden um dabei Laufzeit, Ressourcenbedarf und Verwaltungsaufwand zu ermitteln	Muss

{6.2} Testklasse(n) mit m Threads auf n-Prozessor-Architektur (für m>>n)

Eine oder mehrere Testklassen mit m>>n Threads sollen auf einer Multi-Prozessor-Architektur ausgeführt werden um dabei Laufzeit, Ressourcenbedarf und Verwaltungsaufwand zu ermitteln	Muss
Gleiche Testklasse(n) mit m>>n Threads sollen auf einer Single-Prozessor-Architektur ausgeführt werden um dabei Laufzeit, Ressourcenbedarf und Verwaltungsaufwand zu ermitteln	Muss

{7.1} Einfluss der Methoden-Synchronisation auf Skalierung

Testklasse(n) mit n...m Threads, gemeinsamen Speicherbereich und Methoden-Synchronisation sollen unter Einfluss von klein bis grossem „lock contention“ auf einer Multi-Prozessor-Architektur ausgeführt werden. Dabei sollen Laufzeit, Ressourcenbedarf und Verwaltungsaufwand ermittelt werden.	Kann

{7.2} Einfluss der Objekt-Synchronisation auf Skalierung

Testklasse(n) mit n...m Threads, gemeinsamen Speicherbereich und Objekt-Synchronisation sollen unter Einfluss von klein bis grossem „lock contention“ auf einer Multi-Prozessor-Architektur ausgeführt werden. Dabei sollen Laufzeit, Ressourcenbedarf und Verwaltungsaufwand ermittelt werden.	Kann

{7.3} Einfluss der CAS-Methoden auf Skalierung

Testklasse(n) mit n...m Threads, gemeinsamen Speicherbereich und Lockfreien-Synchronisation (CAS) sollen auf einer Multi-Prozessor-Architektur ausgeführt werden. Dabei sollen Laufzeit, Ressourcenbedarf und Verwaltungsaufwand ermittelt werden.	Kann

{8.1} Parallelisierung durch JOMP

Wirkung der JOMP-Parallelisierung unter Verwendung verschiedener, variabler Thread-Konfigurationen (Attribut).	Muss

7. Spezifikation der Testklasse

Um unsere Skalierungs-Tests und Implementierungen machen zu können benötigen wir eine Testklasse oder eine Test-Anwendung. Dieses Kapitel definiert welche Anforderungen eine entsprechende Anwendung haben muss. Die Spezifikation dient später (siehe Kapitel 8) zur Bewertung möglicher Kandidaten.

7.1. Requirements

Die Requirements können grob in kann- und muss-Requirements aufgeteilt werden:

#	Beschreibung	Typisierung
{R1.1}	Sequenzieller Code	Muss
{R1.2}	Parallelisierbar	Muss
{R1.3}	Messbar	Muss
{R1.4}	Portierbar auf verschiedene Konzepte und Plattformen	Muss
{R1.5}	Nachvollziehbar, einfach, übersichtlich	Muss
{R1.6}	Reproduzierbarkeit	Muss
{R2.1}	Konkurrierender Datenzugriff	Kann
{R2.2}	Visualisierbarkeit	Kann

Nachfolgend werden die Requirements noch etwas genauer erläutert.

Requirement {R1.1}, Sequenzieller Code

Um eine Aussage über die Möglichkeiten der Parallelisierung machen zu können muss der Ausgangszustand ein sequenziell ablaufendes Programm sein. Dies erlaubt auch eine Aussage darüber, wie sich ein parallelisierter Code gegenüber einem sequenziellen Code verhält.

Requirement {R1.2}, Parallelisierbar

Der Code muss natürlich erlauben ihn vollumfänglich oder in Teilen parallel ablaufen zu lassen. Insbesondere Schleifen sind dazu sehr gut geeignet.

Requirement {R1.3}, Messbar

Der ausgewählte Code sollte eine Mindestlaufzeit aufweisen. Dies soll insbesondere die Messungenauigkeit relativieren. Die Laufzeit sollte zwischen 30 Sekunden und 5 Minuten betragen. Längere Laufzeiten sind auch denkbar aber für unsere Zwecke kaum von Interesse da wir uns erhoffen Laufzeitverbesserungen im Faktoren- und nicht im Prozentbereich zu erreichen.

Requirement {R1.4}, Portierbar auf verschiedene Konzepte und Plattformen

Der Ausgewählte Code muss auf die ausgewählten Techniken und auf die ausgewählten Plattformen portierbar sein. Das bedeutet, dass der Code sowohl mit Java Threads als auch mit JOMP implementierbar ist und auf allen zur Verfügung stehenden Testplattformen laufen muss.

Requirement {R1.5}, Nachvollziehbar, einfach, übersichtlich

Der ausgewählte Code muss möglichst einfach strukturiert sein um ihn schnell erklären und verstehen zu können. Dazu muss der Umfang des Kernalgorithmus auf weniger als 100 Zeilen Code implementiert werden können.

Requirement {R1.6}, Reproduzierbarkeit

Der Code muss reproduzierbare Ergebnisse liefern. Das heisst, dass die gemessene Laufzeit möglichst kleine Schwankungen aufweisen soll. Die Schwankungen müssen bei mehreren Durchläufen unter gleichen Testbedingungen im Bereich unter 10% liegen.

Requirement {R2.1}, Konkurrierender Datenzugriff

Hiermit ist die so genannte ‚lock contention' gemeint. Um den Einfluss von konkurrierenden, synchronisierten Speicherzugriffen zu simulieren muss der Code auf gemeinsame Datenfelder zugreifen. Da der Einfluss verschiedener Locking-Technologien nicht im Hauptfokus der Arbeit liegt ist dieses Requirement optional.

Requirement {R2.2}, Visualisierbarkeit

Nackte Zahlen und Laufzeiten sind zwar aussagekräftig, aber sehr trocken zu lesen. Im Optimalfall ist die Arbeitsgeschwindigkeit der Anwendung 1:1 verfolgbar.

8. Testklasse

Dieses Kapitel dient zur Dokumentation der ausgewählten Testklassen inklusive Quelltexte und weitere Fragmente. Die Testklassen werden nach den Spezifikationen aus Kapitel 7 ausgewählt. Die ausgewählte Klasse (bzw. die ausgewählten Klassen) wird dann in Kapitel 9 ausgearbeitet (parallelisiert).

8.1. Basisinformationen

Der Fokus liegt hier auf der parallelen Verarbeitung mit dem gewünschten Nebeneffekt der Visualisierung. Daher liegt es nahe sich im Bereich der Computergrafik umzusehen. Nach kurzer Recherche stellen sich Fraktale als besonders geeignet heraus. Insbesondere die Visualisierung ist damit sehr schön zu sehen; weshalb diese Grafiken auch unter dem Oberbegriff der Computer-Kunst zusammengefasst werden.

Fraktale haben ausserdem die für uns angenehme Eigenschaft, dass sie sich meist beliebig komplex berechnen lassen. Da zu diesem Zeitpunkt die zur Verfügung stehende Testplattform (insbesondere die Hardware) noch nicht eindeutig feststeht dürfte sich diese Eigenschaft als sehr nützlich erweisen. Eine Berechnung, die auf unseren Laptops Minuten dauert könnte ansonsten auf der Testplattform innert Sekunden erledigt sein was meistens mit einer grossen Messungenauigkeit einhergeht.

Weiterführende Informationen:

- Wikipedia, Fraktal: [FRAKTAL]

8.2. Die Mandelbrot Menge

Wie einleitend erwähnt eigenen sich Fraktale mit hoher Wahrscheinlichkeit am besten für unser Vorhaben. Hier bietet sich die klassische Mandelbrotmenge an (siehe auch [MANDELBROT]). Die Menge wird wegen ihrer Form auch gerne Apfelmännchen genannt:

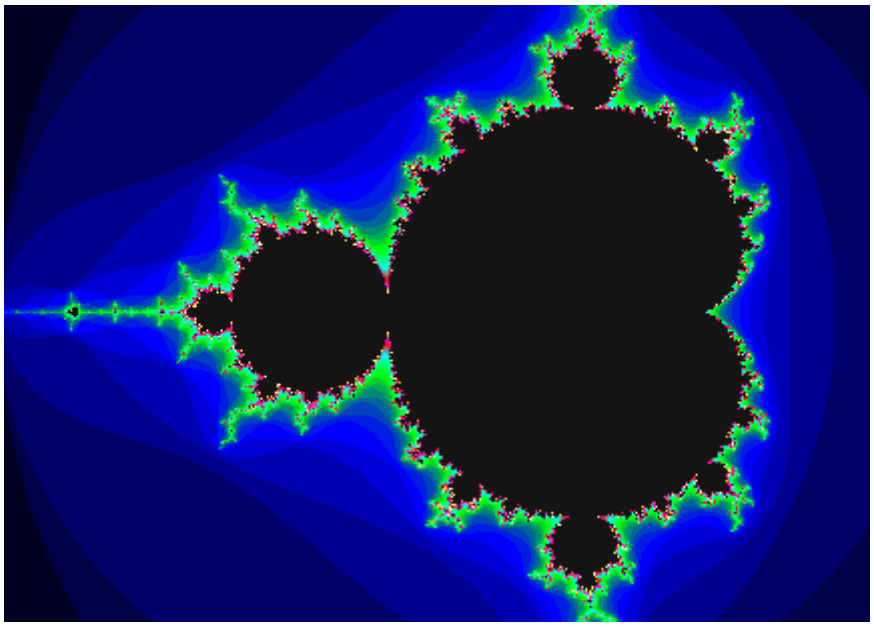

Abbildung 1 Mandelbrot-Menge (Apfelmännchen)

Die Farbgebung kann dabei frei an die persönlichen Vorlieben angepasst werden. Eine weitere wichtige Eigenschaft ist, dass die Menge rekursiv berechnet wird und sich Teile davon beliebig vergrössern lassen. Je nach gewählter Rekursionstiefe werden dabei mehr Details sichtbar:

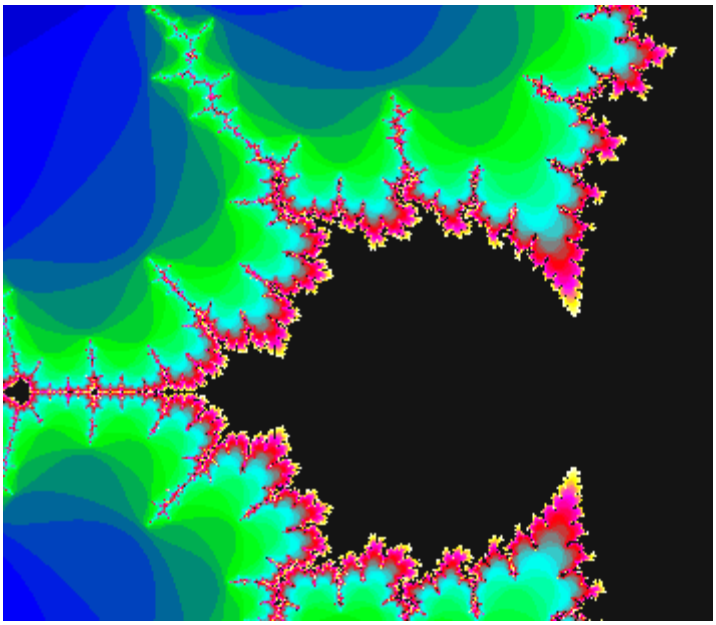

Abbildung 2 Vergrösserung der Mandelbrot Menge

Das Bild zeigt eine Vergrösserung des schwarzen Kreises auf der linken Seite.

Positiv zu erwähnen ist hier auch die Möglichkeit einzelne Bildteile komplett losgelösst voneinander zu berechnen. Diese Eigenschaft ist ideal für eine parallele Verarbeitung. Dadurch ist im Optimalfall überhaupt keine Synchronisierung der Prozesse/Threads nötig. Möchte man das Ergebnis aber visualisieren so muss zumindest in einen gemeinsamen Bildspeicher geschrieben werden.

Auf die mathematischen Grundlagen möchten wir innerhalb dieser Arbeit nicht näher eingehen. Auch der verwendete Code stammt aus Drittquellen und wird angepasst um eine parellele Verarbeitung zu ermöglichen.

In diesem Kapitel wird die ausgewählte Testklasse dokumentiert (in unveränderter Originalfassung).

8.3. Der Algorithmus

Der Algorithmus ist im Internet in verschiedensten Implementierungen zu finden. Eine davon haben wir uns rausgesucht und werden diese für unsere Arbeit verwenden. Der ursprüngliche Code stammt aus [JAVAMANDELBROT].

Die Berechnung kann mit folgendem Codefragment beschrieben werden:

```
   public void run() {
         double dx = width / sizex;
         double dy = height / sizey;

         double z = startx , zi = starty;

         done = false;

         System.out.println("Calculating...");
         for (int x = 0 ; x < sizex ; x++) {
               zi = starty;
               int it;
               for (int y = 0 ; y < sizey ; y++) {
                     if ((it =mandelbrotTest(z, zi)) != -1) {
                           // In the mandelbrot set.
                           raster.setPixel(x,y,(int[]) colors[it]);

                     } else {
                           // Not in the mandelbrot set
                           raster.setPixel(x,y,black);
                     }
                     zi += dy;
               }
               if ( (x%5) == 0) {
                     parent.repaint();

               }
               z += dx;
         }

         done = true;

         System.out.println("Done!");
   }

}
```

Listing 1 Mandelbrot Basisalgorithmus

Es wird also hauptsächlich für jeden Pixel die Methode `mandelbrotTest(z, zi)` ausgeführt. Der Rückgabewert dieser Methode entspricht der Anzahl Iterationen (oder `-1`, falls die maximale Anzahl überschritten wurde). Die errechnete Iterationszahl wird dann als Index für ein Array von Farben verwendet. Im Beispielcode wird dieses Array statisch initialisiert und bietet 200 Farben (was exakt der maximalen Iterationstiefe im Originalcode entspricht).

```
   public int mandelbrotTest(double a, double bi) {
         // System.out.println("Testing ("+ a + "," + bi + ")");

         double atmp, btmp;
         int number = 0;
         double z = 0,zi = 0;
```

```
        while ( (number != 200) && (comp_magnitude(z,zi) < 2)) {

              number++;
              atmp = comp_mult_real(z,zi,z,zi);
              btmp = comp_mult_imag(z,zi,z,zi);

              z = atmp;
              zi = btmp;

              z += a;
              zi += bi;
        }

        if (number == 200) {
              // System.out.println("Part of the Mandelbrot set!");

              return -1;
        } else {
              // System.out.print(" " + number);
              return number;
        }
  }
```

Listing 2 Mandelbrot Test Methode

Diese Funktion wiederum verwendet einzig die externen Methoden `comp_magnitude()`, `comp_mult_real()` und `comp_mult_imag()`.

Diese sind wie folgt definiert:

```
  public static double comp_mult_real(double a, double b,
                                       double c, double d) {
        return (a * c) - (b * d);
  }

  public static double comp_mult_imag(double a, double b,
                                       double c, double d) {
        return (a * d) + (b * c);
  }

  public static double comp_magnitude(double a, double b) {
        return Math.sqrt( a * a + b * b);
  }
```

Listing 3 Mandelbrot Hilfsmethoden

Die Initialisierung der Farben wird hier nicht abgebildet, da es sich um eine statische Liste (Hard-Coded) handelt. Für unsere Zwecke lässt sich der Code relativ einfach erweitern und flexibilisieren. Wir werden zu einem in Kapitel 8 auf die Modifikationen der eigenen Implementierung eingehen.

Den gesamten Code ist auf der Webseite unter [JAVAMANDELBROT] einsehbar und wird auch im CVS Repository abgelegt um die Nachvollziehbarkeit zu gewährleisten.

Weiterführende Informationen:

- Java Mandelbrot Fraktal Renderer: [JAVAMANDELBROT]

8.4. Analyse des Algorithmus

Die in der Hauptschleife (siehe Listing 1) verwendeten Methoden und Daten sind bereits alle lokal. Dies erleichtert die Parallelisierung sehr. Auf den ersten Blick könnte man meinen die Variabeln `z` und `zi` hängen jeweils vom vorherigen Schleifendurchgang ab. Dies ist aber nicht so, da diese nur jeweils um eine Einheit inkrementiert werden. Aus diesem Grund ist es deshalb ebenso möglich den Wert von `z` und zi einer ausgewählten Iteration direkt zu bestimmen (`z = startx + x * dx` sowie `zi = starty + y * dy`).

Weiter wird auf die gemeinsamen Objekte `raster` und `colors` zugegriffen. Das `colors` Objekt wird dabei nur lesend verwendet und besteht aus einem statischen Array, welches die Farben für die Iterationsstufen beinhaltet. Hier braucht also nicht synchronisiert zu werden. Das Objekt `raster` wird dagegen schreibend verwendet. Allerdings wird ein Pixel (der per x- und y-Koordinate definiert ist) nie zweimal beschrieben. Somit braucht hier auch nicht synchronisiert zu werden.

Im unglücklichsten Fall würde hier die Methode raster.setPixel() über ein einziges Objekt synchronisiert sein. In diesem Fall würde nach jeder Berechnung eines Pixels versucht den Lock zu bekommen. Dies würde sich vermutlich massiv auf die parallele Verarbeitung auswirken da hohe ‚lock contention' (siehe [1]) zu befürchten wäre.

Im Optimalfall würde das verwendete Bild-Objekt einen eigenen Lock pro Pixel verwenden. Dieser würde dann nur von einem einzigen Thread verwendet und würde die Konkrrenzierung (lock contention) der einzelnen Locks verringern.

Teilt man das Bild in Kacheln oder Streifen auf (lock coarsening/lock striping, siehe [1]) so wären auch Mischformen denkbar ohne gleich pro Pixel einen eigenen Lock zur Verfügung zu stellen. Man könnte das Bild in mehrere Blöcke aufteilen und für jeden Block einen eigenen Lock verwenden. Dies würde die Wahrscheinlichkeit für eine konkurrierende Lock-Anfrage senken. Im Optimalfall würden gleich viele Lock-Objekte wie Threads zur Verfügung stehen und diese exakt die Teile des Bildes „schützen", die von einem Thread bearbeitet werden.

Eine weitere Möglichkeit wäre die Verwendung von Compare-and-Swap (CAS) (siehe [1]) Algorithmen. Wäre jeder Pixel durch einen CAS-Algorithmus geschützt würde der Aufwand für das Locking entfallen. Da anzunehmen ist, dass jeder Pixel nur einmal beschrieben wird dürfte diese Implementierung sogar sehr effizient sein, da jeder Schreibvorgang erfolgreich wäre.

Wir werden beide Möglichkeiten und deren Auswirkungen in Kapitel 9 beleuchten.

8.5. Bewertung

Die ausgewählte Testklasse soll nun anhand der Requirements (siehe Kapitel 7.1) bewertet werden.

#	Beschreibung

{R1.1} Sequenzieller Code:

Die ausgewählte Testklasse ist nicht bereits parallelisiert und arbeitet in der Referenzimplementation mit nur einem Verarbeitungs-Thread.

{R1.2} Parallelisierbar:

Aufgrund der Analyse (siehe Kapitel 8.4) ist anzunehmen, dass sich der Algorithmus sehr gut parallel umsetzen lässt.

{R1.3} Messbar:

Die Berechnungsdauer eines kompletten Bildes bei fest definierten Parametern lässt sich messen. Die Dauer der Berechnung lässt sich durch variable Iterationstiefen fast beliebig wählen. Somit ist die Messbarkeit gegeben.

{R1.4} Portierbar auf verschiedene Konzepte und Plattformen:

Durch die Implementierung in Java ist der Code Plattformunabhängig. Sowohl auf Hardware- wie auch auf Software-Ebene. Der Algorithmus lässt sich sowohl mit Java Threads als auch mit anderen Techniken parallelisieren.

{R1.5} Nachvollziehbar, einfach, übersichtlich:

Der Basisalgorithmus besteht im Wesentlichen aus zwei verschachtelten `for`-Schliefen. Die Berechnung der Iterationen (`mandelbrotTest()`) ist einfach nachvollziehbar, kann aber für das Verständnis komplett abstrahiert betrachtet werden. Diese Eigenschaften sorgen für eine sehr einfach erklärbare Codebasis.

{R1.6} Reproduzierbarkeit:

In Tests hat sich gezeigt, dass die Berechnung bei mehreren Durchläufen praktisch immer gleich lange dauert. Um die geforderte Messungenauigkeit einzuhalten muss lediglich die Wahl einer geeigneten Iterationstiefe getroffen werden.

{R2.1} Konkurrierender Datenzugriff:

Wie in Kapitel 8.4 erwähnt existiert kaum ‚lock contention'. Diese kann aber „künstlich" erzeugt werden um die Auswirkungen zu eruieren. Somit kann auch diese Anforderung im Bedarfsfall abgedeckt werden.

{R2.2} Visualisierbarkeit:

Fraktale eignen sich sehr gut zur Visualisierung. Die berechneten Pixel lassen sich direkt am Monitor darstellen um den Fortschritt der Berechnung direkt mitverfolgen zu können.

Damit erfüllt der ausgesuchte Code alle Anforderungen. Einzig die Anforderung {R2.1} wird nicht direkt vom Referenzcode erfüllt. Diese lässt sich aber simulieren was sich auch als Vorteil herausstellen kann, da wir versuchen werden den Einfluss verschiedener Locking-Mechanismen einander gegenüberzustellen. Dann wäre es schlecht, wenn der Algorithmus bereits eine bestimmte Methode zwingend vorschreiben würde.

9. Implementierung

In diesem Kapitel werden die implementierten, parallelen Klassen dokumentiert. Hierbei werden die wichtigsten Code-Fragmente kurz erklärt und das Anwendungs-Design offengelegt.

9.1. Java Threads

Diese Implementierung basiert vollständig auf den Basisklassen der Java API (Version 1.5). Dabei werden die Klassen gemäss dem MVC (Model View Control) Konzept unterteilt:

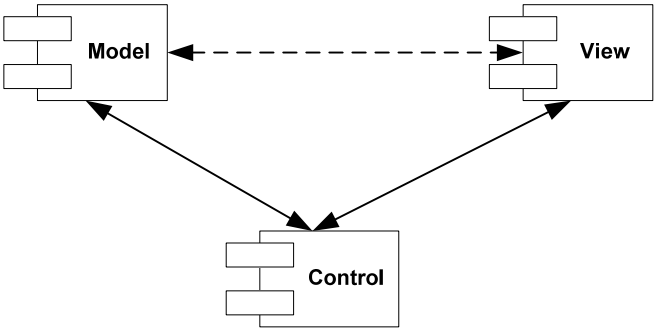

Abbildung 3 MVC Klassenstruktur

Die schematische Darstellung der internen Architektur sieht wie folgt aus:

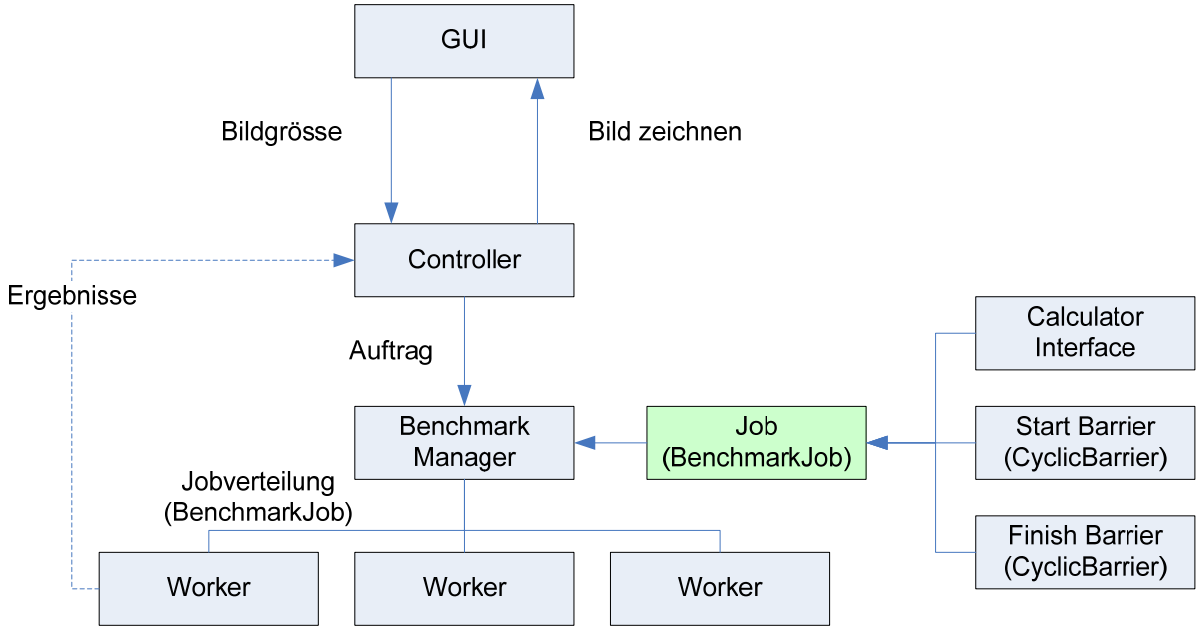

Abbildung 4 Architektur der Implementierung

Einzelne Komponenten werden nachfolgend etwas genauer erklärt. Der gesamte Source Code ist Teil der Arbeit und wird mit den Dokumenten zusammen abgegeben. Die technische Dokumentation auf Source Ebene wird mittels JavaDoc realisiert. Die entsprechenden Dokumentationen sind ebenfalls Teil dieser Arbeit und werden ebenfalls mit den Dokumenten zusammen abgegeben.

9.1.1. Model

Zum Modell gehört dabei die eigentliche Berechnung sowie die Thread Verwaltung. Wobei man sich bei Letzterer auch über die Zugehörigkeit zum Controller streiten könnte. Im weiteren Sinne gehören die abgeleiteten `BufferedImage` Klassen dazu. Die relevanten Klassen und Code-Fragmente werden nachfolgend kurz erklärt.

Benchmark Manager

Die Thread-Verwaltung übernimmt hier der so genannte Benchmark Manager. Dieser erzeugt und verwaltet die gewünschte Anzahl Worker Threads. Die Besonderheit hier ist, dass die Threads während der gesamten Laufzeit des Programmes bestehen bleiben. Das heisst, dass diese Worker sich nach beendeter Arbeit nicht beenden und neu erzeugt werden müssen. Dazu holen sich die Worker ihre Arbeiten aus einer blockierende Queue (genannt jobs).

Dieses Vorgehen ist notwendig um die Last auf den einzelnen Threads besser überwachen zu können. Würden sich die Threads nach der Berechnung beenden, so wären diese auch im Betriebssystem verschwunden. Wir möchten aber die Thread-Prioritäten für jeden einzelnen Thread konfigurieren können und diese Abbildung auch auf der Ebene des Betriebssystems nachvollziehen können. Dies geht nur, wenn die Threads über die gesamte Laufzeit des Programms existieren.

Hier eine Liste der wichtigsten Methoden der `BenchmarkManager` Klasse:

Tabelle 18 BenchmarkManager Methoden

Methode	Beschreibung
`setWorkerNumber()`	Setzt die neue Anzahl von Worker Threads. Falls mehr Worker bereits existieren, dann werden später erzeugte Threads beendet. Falls die neue Anzahl aber grösser ist, dann werden einfach weitere Threads erstellt. Bestehende Threads bleiben in jedem Fall erhalten.
`setJob()`	Stoppt alle Worker Threads und erstellt einen neuen Aurtrag. Dieser wird von den laufenden Threads abgeholt und bearbeitet.

Die Klasse `BenchmarkManager` beinhaltet ausserdem die privaten Klasse `BenchmarkJob`. Diese wird als „Container“ verwendet um eine Job-Definition in der Queue abzulegen.

Die zwei weiteren privaten Klassen `StartCallback` und `StopCallbacl` werden ebenfalls per `BenchmarkJob` in der queue an die Worker übergeben. Hier allerdings verpackt in eine `CyclicBarrier`. Die exakte Messung der Berechnungsdauer schliesst die Verteilung der Jobs und den damit verbundenen Aufwand nicht ein. Deshalb warten die Threads vor dem Start an einer Barriere (`CyclicBarrier`). Sind alle Threads an dieser Barriere angekommen wird die Barriere geöffnet und gleichzeitig eine „Callback“-Methode aufgerufen. Beim Start ist dies die `run()` Methode der `StartCallback` Klasse. Diese speichert lediglich die Startzeit mittels `System.nanoTime()`. Am Ende warten wieder alle Threads an der nächsten Barriere. Erst wenn alle Threads fertig sind wird diese durch den letzten eintreffenden Thread geöffnet. Dadurch wird die Callback-Methode aufgerufen und dadurch die Laufzeit ermittelt. Dadurch wird sichergestellt, dass die Laufzeit der gesamten Zeit vom Start aller Threads bis zur Terminierung des letzten Threads beinhaltet.

Die Threads beenden sich aber nach der Berechnung nicht sondern holen sich den nächsten Job von der Queue ab.

Worker Threads

Die grundsätzliche Arbeitsweise der Worker Threads ist im folgenden Bild schematisch dargestellt:

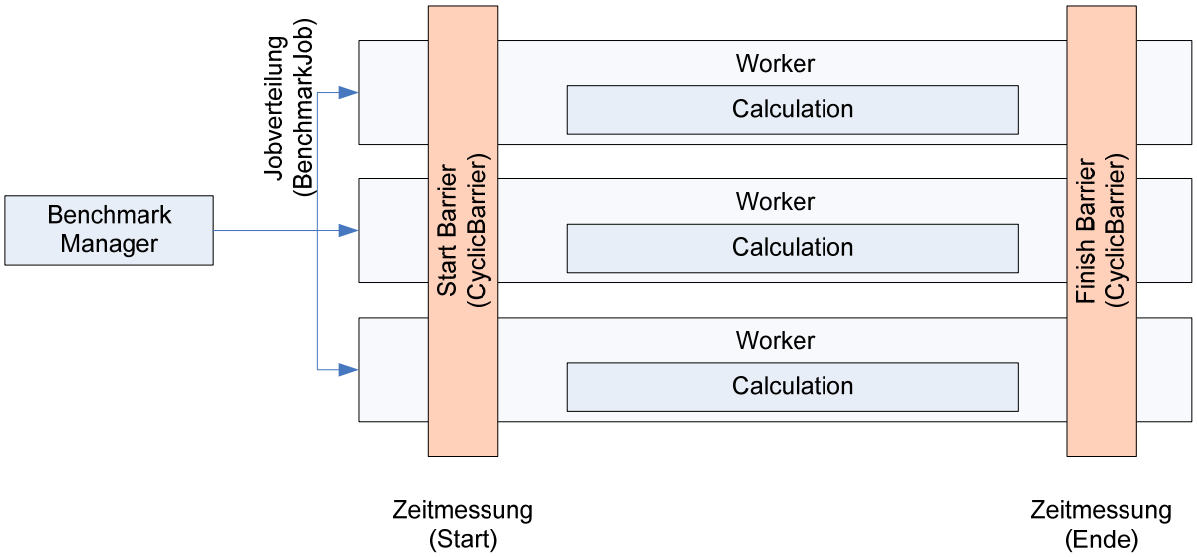

Abbildung 5 Worker Threads Schema

Die Threads bekommen als ihre Aufträge vom Benchmark Manager. Sie starten aber nicht direkt mit der Berechnung sondern warten an einer Barriere. Erst wenn alle Threads an der Barriere angekommen sind wird diese freigegeben und alle Workers können gleichzeitig mit der Arbeit beginnen. Gleichzeitig sorgt die Barriere für die Messung des Startzeitpunktes. Diese Architektur erlaubt die Isolation der Messung des Berechnungsaufwandes. Weder das Thread Setup noch andere Aufgaben haben somit einen Einfluss auf die Messung.

Nach Abschluss der Berechnungen warten die Threads wieder an einer Barriere. Erst wenn alle Threads ihre Arbeit beendet haben wird die Zeit gemessen und die Laufzeit berechnet. Danach kehren die Threads wieder zu ihrem Ursprungszustand zurück und warten auf weitere Aufträge.

Der wichtigste Teil der Worker-Threads ist die `run()` Methode. Diese wird hier kurz erklärt:

```
    @Override
    public void run() {
        while (!isInterrupted()) {
            try {
                // get a job from the queue
                BenchmarkJob job = jobQueue.take();
                synchronized (this) {
                    cancelProcessing.set(false);
                    this.startBarrier = job.getStartBarrier();
                    this.finishBarrier = job.getFinishBarrier();
                    this.currentJob = job.getCalculator();
                }

                // releasing the latch
                if (!cancelProcessing.get()) {
                    this.startBarrier.await();
                }

                // do calculation
                currentJob.run();

                // wait until all threads finished
                if (!cancelProcessing.get()) {
                    System.out.println("Thread " + this.getName()
                   + " done (Priority " + this.getPriority() + ")");
```

```
                    this.finishBarrier.await();
                }

            } catch (InterruptedException e) {
                // System.out.println("Interrupted while waiting");
                this.interrupt();
            } catch (BrokenBarrierException e) {
                // System.out.println("Warning: Barrier broken.");
                // do nothing, just start over
            }
        }
        System.out.println("Thread " + this.getName() + " done!");
    }
```

Listing 4 run() Methode der Worker Threads

Wie bereits in [1] erwähnt handelt es sich bei der `run()` Methode um die Methode, die beim Start des Threads ausgeführt wird. In diesem Fall besteht die Methode im Wesentlichen aus vier Teilen:

1. Abholung des Jobs von der Queue.
2. Warten an der Barriere (bis alle Threads bereit sind).
3. Ausführen der Berechnung.
4. Warten an einer weiteren Barriere bis alle Threads die Berechnung ausgeführt haben.

Dann beginnt die Schleife wieder von vorne. Die Schleife läuft so lange bis die `interrupt()` Methode des Threads aufgerufen wird und somit die Prüfmethode `isInterrupted()` den wert `true` zurückgibt.

Die Threads können also so lange für die Berechnung verwendet werden bis sie absichtlich abgebrochen werden.

Die verteilten Arbeitspakete verarbeiten Objekte vom Typ `CalculatorInterface`. Dies ermöglicht die Benutzung der Architektur auch für andere Berechnungen.

Fraktalberechnung

Auch die Klasse zur Berechnung des Fraktals wurde aktualisiert. Diese wurde erweitert um eine beliebige (konfigurierbare) Rekursionstiefe zu erlauben. Dazu musste die statische Definition der Farbtabellen-Berechnung einer dynamischen Methode weichen. Die Berechnung wurde durch ein Lock-Objekt synchronisiert um eine Parameter-Veränderung während Berechnung zu verhindern. Um die Berechnung (welche sehr lange dauern kann) abzubrechen wurden bei beiden, verschachtelten Schleifen Abbruchkriterien eingefügt. Somit lässt sich durch das Setzen eines `AtomicBoolean` Wertes die Berechnung zum jeweils nächsten Schleifendurchgang abbrechen.

Die dazu benötigte `run()` Methode sieht nun folgendermassen aus (leicht gekürzt):

```
    /**
     * Main Mandelbrot calculation routine. Calculates the image.
     */
    public void run() {
        // some parameters are not allowed to be changed during
        // calculation these are prtected by the calculationLock
        synchronized (calculationLock) {
            // calculate render area in Mandelbrot set
            double mandelbrotRenderX = this.mandelbrotCoordinates.x
                    + (this.mandelbrotCoordinates.width /
                  this.image.getWidth() * (this.imageRenderArea.x + 1));
            double mandelbrotRenderY = this.mandelbrotCoordinates.y
                    + (this.mandelbrotCoordinates.height
                            / this.image.getHeight() *
                      (this.imageRenderArea.y + 1));
            double mandelbrotRenderWidth =
                        this.mandelbrotCoordinates.width /
                        this.image.getWidth() *
                        this.imageRenderArea.width;
            double mandelbrotRenderHeight =
                        this.mandelbrotCoordinates.height /
                        this.image.getHeight() *
                        this.imageRenderArea.height;

            // calculate mandelbrot distances from pixel to pixel
            double distanceX = mandelbrotRenderWidth
                    / this.imageRenderArea.width;
            double distanceY = mandelbrotRenderHeight
                    / this.imageRenderArea.height;

            double z = mandelbrotRenderX, zi = mandelbrotRenderY;
            int iterations = 0;
            double colorspacing = (numColors - 1) /
                                  (double) maxIteration;

            for (int x = this.imageRenderArea.x; x <
                 this.imageRenderArea.x
                    + this.imageRenderArea.width
                    && !interruptCalculation.get(); x++) {
                zi = mandelbrotRenderY;
                for (int y = this.imageRenderArea.y; y <
                        this.imageRenderArea.y
                        + this.imageRenderArea.height
                        && !interruptCalculation.get(); y++) {
                    if ((iterations = mandelbrotTest(z, zi)) != -1) {
                        // part of the mandelbrot set
                        // get color index to use
                        int colorIndex = (int) (colorspacing *
                                          iterations);
                        // write pixel
```

```
                        image.setRGB(x, y, ((int[])
                                    colors[colorIndex])[0]);
                    } else {
                        // not part of the Mandelbrot set
                        // set RGB value - use this method because it's
                        // unsnychronized and does not use locking
                        image.setRGB(x, y, 1, 1, colorNotPart, 0, 1);
                    }
                    // notify controller
                    if (x % 50 == 0) {
                        Controller.getInstance().drawImage(image);
                    }
                    zi += distanceY;
                }
                z += distanceX;
            }
            // worker finished - draw final image
            Controller.getInstance().drawImage(image);
        }

    }
```

Listing 5 Angepasste Berechnungsmethode

Die verwendeten Farben werden mit folgendem Algorithmus berechnet:

```
    /**
     * Generates all colors and stores them in an array
     */
    static {
        byte redInc;
        byte greenInc;
        byte blueInc;
        int redValue = 0;
        int greenValue = 0;
        int blueValue = 0;
        for (byte i = 1; i < 8; i++) {
            // red value
            redInc = 0;
            if ((((i - 1) & 0x4) == 0x0) && ((i & 0x4) == 0x4)) {
                redInc = 1;
                redValue = -1;
            } else if ((((i - 1) & 0x4) == 0x4) && ((i & 0x4) == 0x0)) {
                redInc = -1;
                redValue = 256;
            }
            // green value
            greenInc = 0;
            if ((((i - 1) & 0x2) == 0x0) && ((i & 0x2) == 0x2)) {
                greenInc = 1;
                greenValue = -1;
            } else if ((((i - 1) & 0x2) == 0x2) && ((i & 0x2) == 0x0)) {
                greenInc = -1;
                greenValue=256;
            }
            // blue value
            blueInc = 0;
            if ((((i - 1) & 0x1) == 0x0) && ((i & 0x1) == 0x1)) {
                blueInc = 1;
                blueValue = -1;
            } else if ((((i - 1) & 0x1) == 0x1) && ((i & 0x1) == 0x0)) {
                blueInc = -1;
                blueValue=256;
```

```
            }

            // calculate transition states
            for (char j = 0; j < 256; j++) {
                int[] color = new int[1];
                color[0] = (redValue += redInc) << 16
                        | (greenValue += greenInc) << 8
                        | (blueValue += blueInc);
                colors[(i - 1) * 256 + j] = color;
            }

        }
```

Listing 6 Farb-Berechnung

9.1.2. View

Das GUI basiert auf den Java-Swing Klassen und verwendet keine zusätzlichen Hilfsmittel. Im Wesentlichen beschränkt es sich darauf dem Controller die Benutzereingaben weiterzureichen oder umgekehrt die vom Controller benötigten Bildschirmausgaben zu machen.

Die Klassen brauchen hier nicht weiter erklärt zu werden.

9.1.3. Control

Der Controller ist der zentrale Dreh- und Angelpunkt der Anwendung. Die `Controller` Klasse ist als Singleton implementiert. Von ihr existiert nur eine einzige Instanz. Diese Instanz kann durch die statische `getInstance()` Methode von jeder Komponente abgeholt werden. Dies macht es überflüssig jeder Klasse die Referenz auf den Controller zu übergeben.

Neben einer Reihe von get* Methoden bietet der Controller vile set* Methoden um den Status der Anwendung zu beeinflussen. Diese werden insbesondere vom GUI (View) verwendet um die vom Benutzer definierten Optionen zu aktivieren. Es folgt eine Liste der wichtigsten Methoden:

Tabelle 19 Controller Methoden

Methode	Beschreibung
`init()`	Hiermit wird der Controller erstmalig initialisiert. Diese Methode wird nur von der `main()` Methode aufgerufen.
`drawImage()`	Diese Methode zeichnet das als Parameter übergebene Bild im GUI. Diese Methode wird von den Model-Klassen aufgerufen wenn Daten zur Verfügung stehen.
`get*()`	Über diese Methoden kann der aktuelle Status der Anwendung abgefragt werden. Dies Umfasst beispielsweise die aktuellen Koordinaten im Mandelbrot-Bereich oder die Anzahl aktiver Threads.
`interruptCalculation()`	Mittels dieser Methode kann eine laufende Berechnung abgebrochen werden.
`quit()`	Threads Anwendung beenden.
`restartCalculation()`	Neu-Start der Berechnung mit den aktuellen Initialdaten.
`set*()`	Setzen verschiedenster Parameter (Anzahl Threads, Koordinaten, Bildgrösse…)

Die Implementation der einzelnen Methoden ist hier nicht besonders wichtig da der Controller die angefragten Aktionen hauptsächlich direkt an die betroffenen Objekte weiterleitet. So führt ein setNum-

Workers() zum Aufruf von `setWorkersNumber()` der Klasse `BenchmarkManager` (siehe Kapitel 0). Somit beinhaltet der Controller keine aufwändige Programmlogik oder Algorithmen. Der Gesamte Code ist natürlich im Umfang der Arbeit enthalten.

9.2. Locking

Hier werden die relevanten Code-Stellen beschrieben um das Locking-Verhalten zu beeinflussen. Wie in der Analyse (siehe Kapitel 8.4) erwähnt liegt der beste Ansatzpunkt dafür beim Bild. Jeder Thread schreibt die Pixel direkt per `setPixel()` Methode in das Bild. Diese Methode ist in der Referenzimplementation des in `BufferedImage` verwendeten Rasters zwar nicht „Thread safe“ (also auch nicht synchronisiert) aber da jeder Pixel nur von einem Thread geschrieben wird ist dies hier kein Problem. Wir werden an diesem Punkt ansetzen um verschiedene Locking-Verfahren zu testen. Der Einfachheit halber haben wir die `setPixel()` Methode übrigens durch den Aufruf der `setRGB()` Methode direkt aus der `BufferedImage` Klasse ersetzt.

Um verschiedene Locking-Mechanismen auf der Ebene der `BufferedImage` Klasse testen zu können haben wir diese abgeleitet und eine Klasse mit dem Namen `CountingImage` erstellt. Diese bietet zusätzlich noch eine Methode `getCount()` welche einen Zählerstand zurückgibt. Wir haben diesen Zähler eingefügt weil wir festgestellt haben, dass selbst bei einer Methodensynchronisation von `setRGB()` kein Lock gesetzt wird. Wir vermuten, dass die Sun HotSpot VM 1.5 hier optimiert und feststellt, dass ein Lock hier nicht nötig ist. Doch dazu mehr in Kapitel 0.

Die Klassenstruktur sieht nun schematisch wie folgt aus:

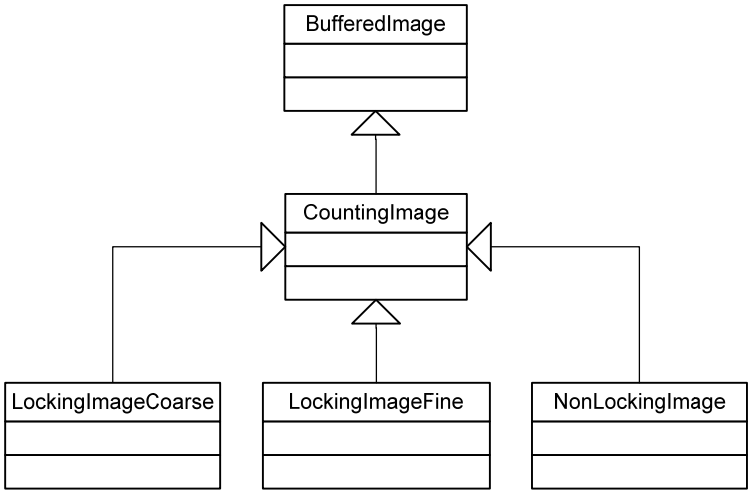

Abbildung 6 Schematische Klassenstruktur, CountingImage

Die Klasse CountingImage verwendet den folgenden Code:

```
public class CountingImage extends BufferedImage {
    public int count = 0;

    public CountingImage(int width, int height, int imageType) {
        super(width, height, imageType);
    }

    public synchronized int getCount() {
        return count;
    }
}
```

Listing 7 CountingImage Implementierung

Da der Controller zentral für die Erstellung der Bilder zuständig ist kann dieser frei zwischen den Implementierungen wählen und erlaubt somit flexibel die Wahl des Locking-Mechanismus bei jedem Testdurchgang.

Die Implemntierung der von `CountingImage` abgeleiteten Klassen wird nachfolgend kurz erklärt.

9.2.1. Kein Locking

Hier wird die Klasse `NonLockingImage` verwendet. Diese besteht quasi nur aus einer Wrapper-Klasse und wurder der Fairness wegen erstellt. Fairness deswegen, weil die Inkrementierung der `count` Variable selbst auch CPU-Zeit verbraucht und ein dieser Aufwand bei allen Implementierungen anfallen soll. Ansonsten wären die Ergebnisse möglicherweise nicht miteinander vergleichbar.

```
public class NonLockingImage extends CountingImage {
    public NonLockingImage(int width, int height, int imageType) {
        super(width, height, imageType);
    }

    public void setRGB(int startX, int startY, int w, int h,
                       int[] rgbArray, int offset, int scansize) {
        count++;
        super.setRGB(startX, startY, w, h, rgbArray, offset, scansize);
    }
}
```

Listing 8 NonLockingImage Implementierung

Ein wichtiger Hinweis sei hier aber noch angefügt: Diese Klasse ist NICHT Thread-Safe. Das bedeutet, dass der parallele Zugriff hier für inkonsistente Daten sorgen kann. Der Grund liegt nicht in der unsynchronisierten `super.setRGB()` Methode sondern im `count++` Ausdruck. Dieser könnte auch als `count = count + 1` geschrieben werden. Dies verdeutlicht, dass der Wert von `count` zuerst ausgelesen wird, dann inkrementiert und anschliessend wieder zurück geschrieben wird. Findet nun irgendwo zwischen auslesen und speichern ein Kontextwechsel statt wo ein anderer Thread den Inhalt von `count` verändert, dann wird beim Zurückschreiben der Wert überschrieben. Deshalb kann (und wird mit hoher wahrscheinlichkeit) `count` nach mehreren hunderttausend parallelen Zugriffen auf `setRGB()` (einmal pro Pixel) einen falschen Wert beinhalten.

Da wir diesen Wert aber nicht wirklich benötigen ist dieses Problem zu vernachlässigen.

9.2.2. Grobes Locking

Unter grobem Locking versteht man die Verwendung eines Locks für weite Code-Teile. Der Nachteil liegt darin, dass die entsprechenden Code-Teile für alle anderen Threads für eine entsprechend lange Zeit blockiert bleiben. In unserem Flll ist die `setRGB()` Methode sehr kurz. Dafür wird sie entsprechend häufig aufgerufen. Bei der Methodensynchronisation werden entsprechend häufig viele Lock-Wechsel und Blockierungen stattfinden.

Wir ersetzen die von `BufferedImage` zur Verfügung gestellte `setRGB()` Methode. Dazu leiten wir von der Klasse `CountingImage` ab (welche ihrerseits von `BufferedImage` abgeleitet ist) und überschreiben die Methode. Hinweis: `BufferedImage` bietet bereits eine synchronisierte `setRGB()` Methode. Diese zu verwenden wäre aber für unsere Einsatzzwecke zu unflexibel und würde ausserdem die Anpassung der Berechnungsklasse bedingen was wir mit einer erweiterten, eigenen Klasse elegant umgehen können.

Hier der Code für die `LockingImageCoarse` Klasse:

```
public class LockingImageCoarse extends CountingImage {
    public LockingImageCoarse(int width, int height, int imageType) {
        super(width, height, imageType);
    }

    public synchronized void setRGB(int startX, int startY, int w,
                    int h, int[] rgbArray, int offset, int scansize) {
        count++;
        super.setRGB(startX, startY, w, h, rgbArray, offset, scansize);
    }
}
```

Listing 9 LockingImageCoarse Implementierung

Hier liegt auch der Grund für unsere zusätzliche `count` Variable. Wir haben festgestellt, dass eine einfache Methodensynchronisation überhaupt keinen Effekt auf die Performance hatte. Dies war auf den ersten Blick sehr unlogisch. Wir vermuten, dass durch interne JVM Optimierungen dieser Lock einfach herausoptimiert wird da die JVM feststellen kann, dass in Wirklichkeit gar kein Lock nötig ist.

Um diese Optimierung zu verunmöglichen findet nun ein Zugriff auf die `count` Variable statt. Dieser muss in jedem Fall synchronisiert werden. Die Entfernung dieses Locks durch Optimierung wäre viel schwerer (wenn auch nicht unmöglich) und wird offenbar von der JVM nicht durchgeführt. Die Testresultate werden dies wohl untermauern können.

9.2.3. Feines Locking

Auch hier wird die `setRGB()` Methode überschrieben und durch eine mit Locking ersetzt:

```
public class LockingImageFine extends CountingImage {
    public LockingImageFine(int width, int height, int imageType) {
        super(width, height, imageType);
        // create locks
        locks = new Object[width][height];
        for (int row = 0; row < height; row++) {
            for (int column = 0; column < width; column++) {
                locks[column][row] = new Object();
            }
        }
    }

    public void setRGB(int startX, int startY, int w, int h,
                       int[] rgbArray, int offset, int scansize) {
        synchronized (this) {
            count++;
        }
        synchronized (locks[startX][startY]) {
            super.setRGB(startX, startY, w, h, rgbArray,
                         offset, scansize);
        }
    }
}
```

Listing 10 LockingImageFine Implementierung

Auch hier wird synchronisiert. Allerdings wird für jeden Pixel ein eigenes Lock-Objekt zur Verfügung gestellt. Lock-contention würde hier also nur auftreten wenn zwei Threads gleichzeitig auf einen identischen Pixel zugreifen würden. Da jeder Pixel nur einmal beschrieben wird dürfte dieser Fall nie eintreten. Den Aufwand zur Überprüfung des Locks muss aber trotzdem gemacht werden. Ob dieser von der JVM ebenfalls wegoptimiert werden kann oder wissen wir zum jetzigen Zeitpunkt noch nicht.

Dieser Code-Teil beinhaltet noch einen weiteren Lock auf die `this` Referenz. Dieser schützt den gemeinsamen Zugriff auf die `count` Variable. Die Besonderheit liegt hier darin, dass der synchronisierte Bereich extrem kurz ist und somit die Wahrscheinlichkeit für Lock-contention sinkt. Der Code zur Inkrementierung der Variable dürfte innerhalb weniger CPU-Befehle erledigt sein.

9.2.4. CAS (Lock-Free)

Um die Vergleichbarkeit der Resultate zu gewährleisten baut diese Locking-Klasse auf der für feines Locking (siehe Listing 10 LockingImageFine Implementierung) auf. Einzig der synchronisierte Block für die Inkrementierung des Zählers wird nicht mehr durch einen Objekt-Lock sondern durch eine CAS-Methode implementiert. Zu diesem Zweck wird die `AtomicInteger` Klasse verwendet. Diese

bietet eine Lock-freie `incrementAndGet()` Methode, die mittels CAS Funktion implementiert wurde. Der verwendete Code sieht folgendermassen aus:

```
public class LockingImageCAS extends CountingImage {
    /** Array of objects used to lock each pixel */
    private Object[][] locks;
    /**
     * Use an atomic integer as a counter. This one provides a CAS
     * method which is lock-free.
     */
    private AtomicInteger atomicCount = new AtomicInteger(0);

    /**
     * @see ch.skybeam.mandelbrot.model.lock.CountingImage#getCount()
     */
    @Override
    public synchronized int getCount() {
        return atomicCount.get();
    }

    public LockingImageCAS(int width, int height, int imageType) {
        super(width, height, imageType);
        // create locks
        locks = new Object[width][height];
        for (int row = 0; row < height; row++) {
            for (int column = 0; column < width; column++) {
                locks[column][row] = new Object();
            }
        }
    }

    @Override
    public void setRGB(int startX, int startY, int w, int h,
                       int[] rgbArray, int offset, int scansize) {
        // increment counter using a CAS method.
        atomicCount.incrementAndGet();
        synchronized (locks[startX][startY]) {
            super.setRGB(startX, startY, w, h, rgbArray,
                         offset, scansize);
        }
    }
}
```

Abbildung 7 LockingImageCAS Implementierung

Die Impementierung der CAS-Methode `incrementAndGet()` sieht folgendermassen aus:

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Abbildung 8 compareAndGet Implementierung

9.3. JOMP

Bei JOMP (siehe auch [JOMP]) handelt es sich um eine Implementierung der OpenMP direktiven für Java. Die Implementierung unterscheidet sich aber in der Umsetzung von der Referenz-Spezifikation. In Java übernimmt nicht der Java-Compiler die Umsetzung mittels Compiler-Direktiven sondern ein Pre-Compiler. Dieser übersetzt die JOMP-Klassen (Dateiendung .jomp) in Java Klassen (Dateiendun .java). Die generierten Java-Klassen können dann mit dem normalen Java Coompiler übersetzt werden. Auch in weiteren Details unterscheidet sich die Implementation. Einige Direktiven werden nicht unterstützt und gemäss Dokumentation werden die Threads nicht am Anfang generiert und existieren dann während des ganzen Programmablaufes sondern werden erst zur Laufzeit erzeugt und gleich wieder beendet.

Wir wollen hier untersuchen in wie fern sich die Mandelbrot-Berechnung mit JOMP parallelisieren lässt. Der Vorteil dieser Methode liegt insbesondere darin, dass die Architektur der Applikation meist nicht geändert werden muss um einige Teile/Schleifen parallel ablaufen zu lassen.

Ein weiter Nachteil der JOMP-Implementierung scheint die manuelle Konfiguration zu sein. Da über die Java-API die aktuelle CPU-Anzahl nicht abgefragt werden kann muss diese dem Programm mitgeteilt werden. Entweder zur Laufzeit oder per `-Djomp.threads=n` Parameter beim Start der Anwendung.

Weiterführende Informationen:

- OpenMP, Homepage: [OPENMP]
- EPCC, OpenMP-like directives for Java: [JOMP]

9.4. JOMP Architektur

Der Entwicklungsprozess bei der Entwicklung von JOMP-Programmen ist im folgenden Bild schematisch dargestellt:

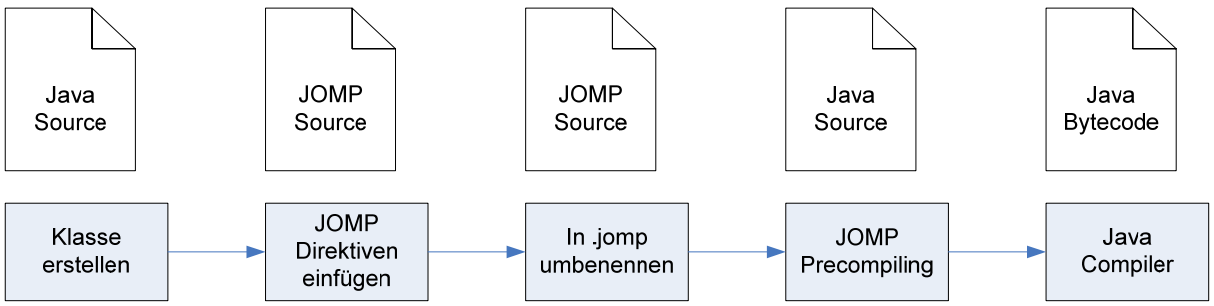

Abbildung 9 Schematische JOMP Entwicklung

Meistens wird zuerst der Java-Sourcecode erstellt. Häufig ist dieser aber schon in Form einer seriellen Verarbeitung vorhanden, der nun parallelisiert werden soll. Im bestehenden Java-Code werden jetzt JOMP-Direktiven (einfache Kommentare) eingefügt. Die daraus entstehende Datei könnte in der Regel immer noch mit dem Java-Compiler übersetzt werden. Dieser würde aber die JOMP Direktiven ignorieren und wie gewohnt ein Single-Threaded Programm erzeugen. Um diesen erweiterten Sourcecode als JOMP Programm zu kennzeichnen wird die Datei von `.java` in `.jomp` umbenannt. Diese Dateien können dann mit dem JOMP-Compiler in eine neue Java-Sourcecode-Datei konvertiert werden. Diese generierten Sourcen verwenden dann erweiterte Konstrukte um die Arbeit auf Threads aufzuteilen. Die generierten Sourcen lassen sich dann wieder mit einem beliebigen Java-Compiler in Bytecode übersetzen der auf einer normalen JVM läuft.

Da die JOMP Source-Dateien lediglich JOMP-Kommentare enthalten lassen sich diese bei Bedarf sogar ohne JOMP mit dem Java-Compiler übersetzen. Dabei werden die JOMP-Direktiven einfach ignoriert.

Die Übersetzung einer JOMP-Klasse geschieht mit folgendem Aufruf:

```
java -cp jomp1.0b.jar jomp.compiler.Jomp <Klasse>
```

Listing 11 Übersetzung einer JOMP-Klasse

Der Ausdruck `<Klasse>` braucht dabei natürlich durch die entsprechende JOMP-Klasse ersetzt zu werden. Die Endung `.jomp` wird nicht angegeben. Der `-cp` Parameter wurde nur der Vollständigkeit halber angeben, da das `jomp1.0b.jar` Java Archiv sowohl den Compiler als auch die Laufzeitklassen enthält. Dieselbe JAR Datei muss übrigens auch bei der Programmausführung im Classpath zu finden sein, da dort die von JOMP verwendeten Klassen liegen.

Beim Start der Applikation braucht dann noch per Parameter mitgeteilt zu werden, auf wie viele Threads die Aufgabe verteilt werden soll:

```
java -Djomp.threads=n <Klasse>
```

Listing 12 Start einer JOMP-Applikation

Hier unterscheidet sich JOMP einmal mehr von der C/C++/Fortran Implementierung. Dort werden standardmässig so viele Threads wie CPUs erzeugt. Bei Java ist dies nicht möglich, da einer Java-Anwendung die Anzahl physikalischer Prozessoren nicht bekannt ist. Deshalb muss dieser Parameter entweder beim Start oder zur Laufzeit gesetzt werden. JOMP bietet dafür aber auch eine dynamische Anpassung der Thread-Anzahl.

Bei Tests hat sich gezeigt, dass die Implementation noch einige Schwachstellen aufweisst. Insbesondere treten bei der parallelen Anwendung von JOMP und eigenen Java-Threads Probleme in Form von Exceptions auf. Weitere Probleme betreffen die Implementierung. Doch dazu mehr im Kapitel 9.4.1.

9.4.1. Mandelbrot-Berechnung mit JOMP

Um JOMP für die zentrale Berechnung der Mandelbrotmenge verwenden zu können haben wir die bestehende Klasse `MandelbrotCalculator` erweitert und die Berechnungs-Schleife (doppeltverschachtelte `for` Schleife) überschrieben.

Die überarbeitete Methode sieht nun folgendermassen aus (leicht gekürzt):

```
public void run() {
    double mandelbrotRenderX = this.mandelbrotCoordinates.x
            + (this.mandelbrotCoordinates.width / this.image.getWidth()
* (this.imageRenderArea.x + 1));
    double mandelbrotRenderY = this.mandelbrotCoordinates.y
            +           (this.mandelbrotCoordinates.height          /
this.image.getHeight() * (this.imageRenderArea.y + 1));
    double mandelbrotRenderWidth = this.mandelbrotCoordinates.width
            / this.image.getWidth() * this.imageRenderArea.width;
    double mandelbrotRenderHeight = this.mandelbrotCoordinates.height
            / this.image.getHeight() * this.imageRenderArea.height;

    // calculate mandelbrot distances from pixel to pixel
    double distanceX = mandelbrotRenderWidth /
                       this.imageRenderArea.width;
    double distanceY = mandelbrotRenderHeight /
                       this.imageRenderArea.height;

    double z = mandelbrotRenderX, zi = mandelbrotRenderY;
    int iterations = 0;
    double colorspacing = (numColors - 1) / (double) maxIteration;

    // omp parallel for shared(distanceX, distanceY, mandelbrotRenderX,
       mandelbrotRenderY) private(z, zi, iterations)
    for (int x = imageRenderArea.x; x < imageRenderArea.x
            + imageRenderArea.width; x++) {
        z = mandelbrotRenderX + (distanceX * ((x - imageRenderArea.x) +
                                                                1));
```

```
        zi = mandelbrotRenderY;
        for (int y = imageRenderArea.y; y < imageRenderArea.y
                + imageRenderArea.height; y++) {
            zi = mandelbrotRenderY + ((y - imageRenderArea.y)) *
                 distanceY;
            if ((iterations = mandelbrotTest(z, zi)) != -1) {
                // part of the mandelbrot set
                // get color index to use
                int colorIndex = (int) (colorspacing * iterations);
                // write pixel
                image
                        .setRGB(x, y, 1, 1,
                                ((int[]) colors[colorIndex]), 0, 1);
            } else {
                // not part of the Mandelbrot set
                // set RGB value - use this method because it's
                // unsnychronized and does not use locking
                image.setRGB(x, y, 1, 1, colorNotPart, 0, 1);
            }
            // notify controller
            if (x % 50 == 0) {
                Controller.getInstance().drawImage(image);
            }
        }
    }
    Controller.getInstance().drawImage(image);
}
```

Listing 13 JOMP Implementierung der MandelbrotCalculator Klasse

Hier wurde sehr wenig geändert. Lediglich die `omp parallel for` Direktive vor der ersten Schleife eingefügt. Ausserdem wurde die inkrementelle Berechnung von `z` und `zi` durch eine absolute ersetzt. Damit braucht diese nicht synchronisiert zu werden was den Programmfluss bzw. die parallele Verarbeitung einschränken würde.

Eine JOMP Besonderheit war, dass alle Parameter des Konstruktors in gleichnamige Datenfelder (Klassenvariabeln) geschrieben werden müssen. Ansonsten wiesen die generierten Java-Quelldateien (nach der Generierung durch den JOMP-Compiler) Fehler auf. Deshalb musste in unserem Beispiel noch der Konstruktor überschrieben werden:

```
public class MandelbrotCalculatorJOMP extends MandelbrotCalculator {
    private double mandelbrotX = 0;
    private double mandelbrotY = 0;
    private double mandelbrotWidth = 0;
    private Rectangle renderArea;
    public MandelbrotCalculatorJOMP(BufferedImage image, Rectangle
           renderArea, double mandelbrotX, double mandelbrotY,
           double mandelbrotWidth, int maxIteration) {
        super(image, renderArea, mandelbrotX, mandelbrotY,
              mandelbrotWidth, maxIteration);
        this.mandelbrotX = mandelbrotX;
        this.mandelbrotY = mandelbrotY;
        this.mandelbrotWidth = mandelbrotWidth;
        this.renderArea = renderArea;
    }
[…]
```

Listing 14 Überschriebener Konstruktor

10. Glossar

Tabelle 20 Glossar

Begriff	Beschreibung
Affinität	Bezeichnet die Zuordnung eines Prozesses/Threads zu physikalischen Recheneinheiten. Durch die Definition einer Affinitätsmaske kann gesteuert werden auf welchen Recheneinheiten die Anwendung ausgeführt werden kann. Siehe Kapitel 5.2.
CAS	Compare and Swap bzw. Compare and Set bezeichnet eine atomare (meist hardwareunterstützte) Operation in der ein gespeicherter Wert mit dem vermuteten Wert verglichen wird. Stimmt dieser überein, so wird ein neuer Wert gesetzt. Ansonsten wird nichts getan. CAS Funktionen erlauben Lock-freie Algorithmen.
CMP	Chip Multi Processing (CMP) bezeichnet einen Chip, der in der Lage ist mehrere Prozesse gleichzeitig abzuarbeiten. Dies passiert aber auf einem Chip und nicht auf mehreren Prozessoren. Siehe Kapitel 5.1.
CMT	Chip Multi Threading (CMT) ist eine Technologie bei der ein Prozessor bei jedem Taktzyklus n Instruktionen (je eine pro n-Threads) einlesen kann. Siehe Kapitel 5.1.
CPU	Abkürzung für Central Processing Unit. Wird synonym für die deutsche Bezeichnung Hauptrpozessor bzw. Prozessor verwendet.
CVS	Concurrent Versioning System; Ein System zur Versionierung von Dateien (vorzugsweise Source-Code). CVS erlaubt die konkurrierende Arbeit an Quelltexten ohne diese für den exklusiven Zugriff zu sperren.
JVM	Die Java Virtual Machine ist ein Interpreter für Java Bytecode. Die JVM ist dabei das Bindeglied zwischen Betriebssystem und den plattformunabhängigen Java Anwendungen.
MPI	Das Message Passing Interface (MPI) wird zum Nachrichtenaustausch (Inter-Process-Communication, IPC) verwendet. Dabei kann MPI transparent sowohl auf einem lokalen Rechner als auch verteilt im Netzwerk verwendet werden. Siehe Kapitel 5.3.
NUMA	Non-Uniform Memory Access (NUMA) bezeichnet eine Architektur in der jede Verarbeitungseinheit lokalen Speicher besitzt und durch Kommunikation mit den anderen Verarbeitungseinheiten auch deren Speicher ansprechen kann. Siehe Kapitel 5.1.
OpenMP	Eine Spezifikation der API zur Parallelisierung von Programmen. OpenMP definiert Compiler-Direktiven damit ein Compiler den bestehenden Code parallelisieren kann. Siehe Kapitel 5.3.
Pipelining	Bezeichnet die Abarbeitung einer Instruktion in vereinfachten Teilschritten. Dadurch kann die folgende Instruktion bereits eingelesen werden sobald die vorhergehende die nächste Stufe erreicht hat. Siehe Kapitel 5.1.
Scheduling	Bezeichnet die Tätigkeit des Betriebssystems beim Preemptiven Multitasking die Prozessorzeit nach einem bestimmten Algorithmus den einzelnen Ausführungseinheiten zuzuweisen (auf Ebene Thread oder Prozess).

Begriff	Beschreibung
Skalar	Ein Prozessor in Skalarem Design verarbeitet immer nur eine Instruktion gleichzeitig. Siehe Kapitel 5.1.
SMP	Symmetric Multi Processing (SMP) bezeichnet die Verarbeitung mit parallel arbeitenden Einheiten wobei jede Einheit gleichberechtigt behandelt wird. Siehe Kapitel 5.1.
Superskalar	Ein Prozssor in superskalarem Design versucht mittels Dispatcher alle Recheneinheiten gleichzeitig auszulasten. Siehe Kapitel 5.1.
TBB	Intel Thread Building Blocks. Eine C++ Bibliothek die Methoden zur parallelen Verarbeitung bereitstellt (Schleifenparalleisierung). Siehe Kapitel 5.3.
Thread	Ein leichtgewichtiger Prozess. Ein Thread teilt den Adressraum mit dem Prozess zu dem er gehört. Dadurch werden einerseits die Kommunikation und andererseits der Kontextwechsel beschleunigt.
UMA	Uniform Memory Access (UMA) bezeichnet eine Architektur in der alle Verarbeitungseinheiten über ein gemeinsames Bussystem auf den Speicher zugreifen. Siehe Kapitel 5.1.

11. Verzeichnisse

11.1. Tabellenverzeichnis

11.2. Abbildungsverzeichnis

11.3. Code Listings

11.4. Index