Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg

Studiengang Geoinformatik und Geodäsie

Untersuchungen zur Parallelisierung praktischer Problemstellungen für Mehrkern-Prozessoren

Masterthesis

vorgelegt von: *Daniel Rohde*

Zum Erlangen des akademischen Grades

„Master of Engineering“ (M.Eng.)

Erstprüfer: Prof. Dr.-Ing. Andreas Wehrenpfennig
Zweitprüfer: Prof. Dr. Welf Löwe
Abgabetermin: 02. 05. 2011

urn:nbn:de:gbv:519-thesis2011-0064-1

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Masterthesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Berlin, den 01.05.2011

Daniel Rohde

Kurzfassung

Die hier vorliegende Masterthesis, des Fachbereichs LGGB der Hochschule Neubrandenburg, befasst sich mit der Parallelisierung von Algorithmen auf Computersystemen mit Mehrkern-Prozessoren.
Die Arbeit gibt einen Überblick, über die Grundlagen der Parallelisierung. Dabei werden verschieden Möglichkeiten der Parallelisierung mittels Hardware, theoretischen Entwurfsmustern und softwaretechnischen Spracherweiterungen und Bibliotheken für die Programmiersprache C++ aufgezeigt.
Während der Thesis wurden zwei Algorithmen aus der praktischen Photogrammetrie auf verschiedene Weisen sequenziell und parallel implementiert. Bei der Parallelisierung der Algorithmen wurden unterschiedliche Parallelisierungsbibliotheken und Entwurfsmuster eingesetzt, wie zum Beispiel POSIX-Threads, MFC[1], TBB[2] und das Muster der Pipeline. Nach den Implementierungen durchgeführte Laufzeitmessungen geben Aufschluss über die Effizienz der vorgenommenen Parallelisierungen.

Abstract

This master thesis was written for the University of Neubrandenburg at the department of LGGB. The topic is the parallelizing of algorithms for computer systems with multicore processor architecture. The master thesis is giving an overview about the basics of parallelizing software. Thereby are different possibilities explained like hardware parallelizing, design pattern and software parallelizing under C++. During the master thesis two algorithms from the scope of photogrammetry were implemented in a sequential and multiple parallel ways. During the parallelization different paralleling libraries and design patterns were used, like POSIX-Threads, MFC, TBB and the pattern of the pipeline. The performed runtime measurements gave information about the efficiency of the parallelization.

[1] MFC – Microsoft Foundation Classes
[2] TBB – Threading Building Blocks

Inhalt

Inhalt i

Abbildungsverzeichnis iii

Tabellenverzeichnis v

Formelverzeichnis v

1 Einleitung 1

1.1 Abgrenzungen 1

2 Beispielalgorithmen 2

2.1 Datengrundlage 2

2.2 Schnitte-Algorithmus 3

2.3 Photo-2-Scan 5

3 Grundlagen der Parallelisierung 7

3.1 Hardware 7

3.1.1 Mehrkern-Prozessoren 7

3.1.2 Cluster 9

3.1.3 Grafikkarten 10

3.2 Entwurfsmuster 12

3.2.1 Allgemeine Entwurfsmuster 12

3.2.2 Parallele Entwurfsmuster 14

3.3 C++: Spracherweiterungen & Bibliotheken 19

3.3.1 MPI 19

3.3.2 Thread-Bibliotheken 20

3.3.3 Berechnungen auf Grafikkarten 22

3.3.4 Threading Building Blocks 23

3.3.5 Weitere Möglichkeiten 24

4 Vorbetrachtungen 25

5 Implementierung 29

5.1 Computerdetails 29

5.2 Laufzeitbestimmung 31

5.3 Schnitte-Algorithmus 32

5.3.1 Beschreibung der Teilaufgaben 33

5.3.2 Parallelisierung 39

5.4 Photo2Scan 47

5.4.1 Beschreibung der Teilaufgaben 50

5.4.2 Parallelisierung 60
6 Ergebnisse der Zeitmessungen 68
6.1 Schnitte-Algorithmus 68
6.1.1 Testcomputer TC1 68
6.1.2 Testcomputer TC2 70
6.1.3 Abschlussbetrachtung 73
6.2 Photo2Scan 74
6.2.1 Testcomputer TC1 74
6.2.2 Testcomputer TC2 77
6.2.3 Abschlussbetrachtung 80
7 Schlussfolgerung 82
7.1 Bewertung 84
7.2 Aussichten 85
Literatur v
Anhang v
Messwerte Schnitte-Algorithmus TC1 – sequenziell v
Messwerte Schnitte-Algorithmus TC1 – MFC-Threads – kleiner Scan vii
Messwerte Schnitte-Algorithmus TC1 – MFC-Threads – großer Scan ix
Messwerte Schnitte-Algorithmus TC1 – POSIX-Threads – kleiner Scan xi
Messwerte Schnitte-Algorithmus TC1 – POSIX-Threads – großer Scan xiv
Messwerte Photo2Scan TC1 – sequenziell – kleiner Scan xvii
Messwerte Photo2Scan TC1 – sequenziell – großer Scan xviii
Messwerte Photo2Scan TC1 – sequenzielle Pipeline – kleiner Scan xix
Messwerte Photo2Scan TC1 – sequenzielle Pipeline – großer Scan xx
Messwerte Photo2Scan TC1 – parallele Pipeline – kleiner Scan xxii
Messwerte Photo2Scan TC1 – parallele Pipeline – großer Scan xxiii
Messwerte Schnitte-Algorithmus TC2 – sequenziell xxiv
Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – kleiner Scan – Thread 1&2 xxvi
Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – kleiner Scan – Thread 3&4 xxviii
Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – großer Scan – Thread 1&2 xxx
Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – großer Scan – Thread 3&4 xxxii
Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – kleiner Scan – Thread 1&2 xxxiv
Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – kleiner Scan – Thread 3&4 xxxvii
Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – großer Scan – Thread 1&2 xxxix
Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – großer Scan – Thread 3&4 xli

Messwerte Photo2Scan TC2 – sequenziell – kleiner Scan xliv
Messwerte Photo2Scan TC2 – sequenziell – großer Scan xlv
Messwerte Photo2Scan TC2 – sequenziell Pipeline – kleiner Scan xlvi
Messwerte Photo2Scan TC2 – sequenzielle Pipeline – großer Scan xlvii
Messwerte Photo2Scan TC2 – parallele Pipeline – kleiner Scan xlviii
Messwerte Photo2Scan TC2 – parallele Pipeline – großer Scan l

Abbildungsverzeichnis

Abbildung 2-1: Schnitte - Ablauf 5
Abbildung 2-2: Innere Orientierung 6
Abbildung 2-3: Äußere Orientierung 6
Abbildung 2-4: Photo2Scan - Ablauf 6
Abbildung 3-1: Aufbau einer CPU 8
Abbildung 3-2: Mehrkern-Prozessor Architekturen 9
Abbildung 3-3: Aufbau eines kleinen Clusters 10
Abbildung 3-4: Leistungsentwicklung von GPUs und CPUs 11
Abbildung 3-5: ALUs auf einer CPU und einer GPU 11
Abbildung 3-6: Datenorganisation Replizierungsmuster 15
Abbildung 3-7: Aufbau Replizierungsmuster 15
Abbildung 3-8: Divide-and-Conquer Ablauf 16
Abbildung 3-9: Divide-and-Conquer Aufbau 16
Abbildung 3-10: Master/Worker-Muster 17
Abbildung 3-11: Struktur einer Zuständigkeitskette 18
Abbildung 3-12: CUDA zugriff auf die GPU 23
Abbildung 4-1: Grobe Zeitorganisation 25
Abbildung 4-2: Amdahlsches Gesetz 27
Abbildung 5-1: Cache-Hierarchie 30
Abbildung 5-2: Implementation einer Zeitmessung 32
Abbildung 5-3: Format der Ergebnisdateien 32
Abbildung 5-4: Schnitte: Ablauf und Arbeitsbereiche 33
Abbildung 5-5: Definition einer Datenstruktur 34
Abbildung 5-6: Zugriff auf eine DLL-Funktion 37
Abbildung 5-7: Lupos3D Funktion -> Lesen einer Zeile 38
Abbildung 5-8: Ergebnisse in Datei schreiben 39
Abbildung 5-9: Ermittelung der Prozessoranzahl 40
Abbildung 5-10: Beispielrechnung für den Arbeitsbereich 41
Abbildung 5-11: Definition neuer Datenstrukturen 43
Abbildung 5-12: Konfiguration der Thread-Parameter 43
Abbildung 5-13: Anlegen eines Mutex 45
Abbildung 5-14: Anlegen und Starten von Windows-Threads 46
Abbildung 5-15: Anlegen und Starten von POSIX-Threads 47
Abbildung 5-16: Photo2Scan: Ablauf und Arbeitsbereiche 48

Abbildung 5-17: 2D-Ansicht eines Laserscans 49
Abbildung 5-18: Code zum Lesen aus einer Textdatei 51
Abbildung 5-19: grafische Darstellung des Standardfalls der Senkrechtaufnahme 52
Abbildung 5-20: Implementierung einer Ergebnisstruktur und Anwendung der Sinus/Kosinus-Funktionen 52
Abbildung 5-21: Code zum Auslesen von Bildinformationen 54
Abbildung 5-22: Bildung 2 dimensionaler Felder in C++ 54
Abbildung 5-23: Adressierung eines Elementes 54
Abbildung 5-24: grafische Darstellung der Ermittlung der Pixelkoordinate 57
Abbildung 5-25: Beispielergebnis des Berechnungsteils 58
Abbildung 5-26: Schreiben einer Scandatei mit Lupos3D-Funktionen 59
Abbildung 5-27: aktuelle Pipelineversion 61
Abbildung 5-28: Einfügen einer neuen Aufgabe in die Pipeline 62
Abbildung 5-29: Erstellen, Konfigurieren und Starten der sequenziellen Pipeline 63
Abbildung 5-30: Beispiel der Implementation einer Pipelineaufgabe 64
Abbildung 5-31: Struktur der parallelen Pipeline 65
Abbildung 5-32: Beispiel der Implementation einer Pipelineaufgabe mit TBB 66
Abbildung 5-33: Erstellen, Konfigurieren und Starten der parallelen Pipeline 67
Abbildung 6-1: Messergebnisse des Schnitte-Algorithmus (kl. / TC1) 69
Abbildung 6-2: Messergebnisse des Schnitte-Algorithmus (gr. / TC1) 70
Abbildung 6-3: Messergebnisse des Schnitte-Algorithmus (kl. / TC2) 72
Abbildung 6-4: Messergebnisse Schnitte-Algorithmus (gr. / TC2) 73
Abbildung 6-5: Auslastung der parallelen Pipeline 75
Abbildung 6-6: Messergebnisse Photo2Scan-Algorithmus (kl. / TC1) 76
Abbildung 6-7: Messergebnisse Photo2Scan-Algorithmus (gr. / TC1) 77
Abbildung 6-8: Messergebnisse Photo2Scan-Algorithmus (kl. / TC2) 78
Abbildung 6-9: Messergebnisse Photo2Scan-Algorithmus (gr. / TC2) 79

Tabellenverzeichnis

Tabelle 2-1: Vergleich von Dateiformaten 2
Tabelle 3-1: Liste der Entwurfsmuster 13
Tabelle 5-1: Zugriffszeit auf den Cache 30
Tabelle 5-2: Details Testcomputer 1 (Notebook) 30
Tabelle 5-3: Details Testcomputer 2 31
Tabelle 6-1: SpeedUp des Schnitte-Algorithmus (kleiner Datensatz / TC1) 69
Tabelle 6-2: SpeedUp des Schnitte-Algorithmus (großer Datensatz / TC1) 70
Tabelle 6-3: SpeedUp des Schnitte-Algorithmus (kleiner Datensatz / TC2) 72
Tabelle 6-4: SpeedUp des Schnitte-Algorithmus (kleiner Datensatz / TC2) 73
Tabelle 6-5: Theoretischer SpeedUp des Schnitte-Algorithmus 74
Tabelle 6-6: Anteile der Operationen am Photo2Scan-Algorithmus (TC1) 76
Tabelle 6-7: Photo2Scan - SpeedUp (kl. / TC1) 77
Tabelle 6-8: Photo2Scan - SpeedUp (gr. / TC1) 77
Tabelle 6-9: Anteile der Operationen am Photo2Scan-Algorithmus (TC2) 78
Tabelle 6-10: Photo2Scan - SpeedUp (kl. / TC2) 79
Tabelle 6-11: Photo2Scan - SpeedUp (gr. / TC2) 79
Tabelle 6-12: Theoretischer SpeedUp des Photo2Scan-Algorithmus 80

Formelverzeichnis

Formel 1: hessische Normalform 4
Formel 2: Bestimmung des normierten Normalenvektors 4
Formel 3: Abstand der Ebene von Koordinatenursprung 4
Formel 4: Abstand eines Punktes von einer Ebene 4
Formel 5: Bestimmung der Gesamtlaufzeit 25
Formel 6: Zeit sequenzielle Operationen 26
Formel 7: Bestimmung des Anteils einer Operation 26
Formel 8: Zusammensetzung der Gesamtlaufzeit 26
Formel 9: Theoretischer SpeedUp 26
Formel 10: Realer SpeedUp 28
Formel 11: Subtraktion von Vektoren 34
Formel 12: Skalarprodukt zweier Vektoren 34
Formel 13: Kreuzprodukt zweier Vektoren 35
Formel 14: Transformation in das globale Koordinatensystem 35
Formel 15: Berechnung des Arbeitsbereiches 40
Formel 16: Berechnung des Standardfalls der Senkrechtaufnahme 51
Formel 17: Index eines Elementes im 2D Feld 54
Formel 18: Kollinearitätsgleichung 56
Formel 19: Verhältnis zwischen Bildkoordinatensystem und Indexposition 57
Formel 20: Berechnung der Indexposition 57
Formel 21: Prüfbedingungen für die Indexpositionen 58
Formel 22: Benötigter Cache-Speicher für die parallele Verarbeitung 71
Formel 23: Bestimmung des Anteils von Festplattenoperationen 75

1 Einleitung

In nahezu jedem heutzutage neu gekauften Computer steckt Leistung, die nicht gut genutzt wird. Die Software, die im Heim- und auch Wissenschaftsbereich, entwickelt wurde und wird, nutzt oft nicht alle dem Computer zur Verfügung stehenden Ressourcen. Oft spielt die Unwissenheit der Entwickler oder auch der vermeidlich hohe Entwicklungsaufwand eine Rolle bei dem Verzicht auf eine Parallelisierung von Algorithmen.

Diese an der Hochschule Neubrandenburg, Fachbereich Landschaftsarchitektur, Geoinformatik, Geodäsie und Bauingenieurwesen, verfasste Masterthesis soll aufzeigen, wie Parallelisierungen von Algorithmen vorgenommen werden können.

Während dieser Thesis werden verschiedene Möglichkeiten Algorithmen und Software zu parallelisieren vorgestellt und beschrieben. Ausgewählte Möglichkeiten werden im späteren Verlauf der Thesis implementiert und mittels Laufzeitvergleichen untersucht. Dadurch lassen sich nicht nur Aussagen über die direkte Leistungssteigerung im Vergleich zu einer sequenziellen Implementation ziehen, sondern auch die verschiedenen Parallelisierungen lassen sich auch noch untereinander vergleichen und erlauben somit einen Rückschluss auf die Effizienz der jeweiligen Techniken.

Die Basis für die Parallelisierungen bilden dabei zwei Algorithmen aus der praktischen Photogrammetrie. Die Ideen bzw. zum Teil auch die sequenziellen Implementationen stammen von der Firma Lupos3D.

Der Kontakt zu der Firma entstand während der 15. Intergeo, einem Kongress und einer Fachmesse für Geodäsie, Geoinformationen und Landmanagement, in Karlsruhe. Lupos3D stellte in einer Vortragsreihe ihre Software (LuposScan) zur Auswertung von Laserscandaten vor. Während eines Gespräches mit dem Geschäftsführer, Michael Pospiš, stellte sich heraus, dass das von der Firma geschaffene Programm komplett sequenziell entwickelt wurde. Es bestand jedoch auch das Interesse an einer Leistungssteigerung der Algorithmen des Programms. Während weiterer Treffen, in Berlin und Neubrandenburg, wurden die Möglichkeiten der Leistungssteigerung erläutert. Das Resultat war die Idee für zwei parallelisierbare Algorithmen und auch die Übergabe von Beispieldatensätzen.

1.1 Abgrenzungen

Schon zu Beginn dieser Arbeit steht fest, dass eine Vielzahl von parallelen Programmiertechniken existiert und nicht alle während dieser Arbeit umgesetzt werden können. Die Thesis bezieht sich daher auf die Techniken, die dem normalen Nutzer hardwaretechnisch zur Verfügung stehen. Einige Techniken, bei denen die Verfügbarkeit der Grundvoraussetzungen bei einem Nutzer nicht sichergestellt werden kann, werden nicht implementiert. Dazu zählen Cluster-Architekturen und das Rechnen mit Grafikkarten. Bei den beiden angesprochenen Techniken steht ganz besonders die Verfügbarkeit der benötigten Hardwareressourcen im Vordergrund. Denn nicht jede Grafikkarte unterstützt die Parallelisierung von Algorithmen mittels der entsprechenden Techniken (z. B. Cuda von Nvidia). Bei den Cluster-Architekturen steht immer die Einrichtung und Wartung solcher Netzwerke im Mittelpunkt. Diese erfordern ein entsprechendes Wissen des Nutzers beziehungsweise einen entsprechenden Support.

Weiterhin spielt der finanzielle Aspekt bei der Umsetzung der Algorithmen eine Rolle, deswegen können auch nur frei zugängige parallele Programmiertechniken umgesetzt werden.

Die Thesis bezieht sich, aufgrund der zuvor angeführten Aspekte, bei der Implementation der Beispielalgorithmen nur auf die Techniken der Parallelisierung, die bei Systemen mit Mehrkern-Prozessoren möglich sind. Daher liegt der Schwerpunkt der Thesis bei der Threadprogrammierung.

2 Beispielalgorithmen

In den folgenden Abschnitten werden die ausgewählten Algorithmen aus der Praxis beschrieben. Wie schon im Kapitel 1 angedeutet, handelte es sich bei den Beispielalgorithmen um Ideen, die schon in der Implementierung des Programmes LuposScan von Lupos3D umgesetzt wurden. Dort sind die Algorithmen jedoch ausnahmslos ohne parallele Programmieransätze implementiert. Damit der Zweck der verwendeten Algorithmen deutlich wird, wird die Theorie hinter dem jeweiligen Algorithmus genauer betrachtet und auch der Ablauf wird genauer aufgezeigt.

Das Programm LuposScan arbeitet mit einem ganz speziellen Datenformat. Damit die Vergleiche zwischen den sequenziellen Implementierungen von Lupos3D und den während dieser Thesis entstandenen Programmen nicht schon am Datenformat scheitern, wird als Erstes das Format genau betrachtet und beschrieben.

2.1 Datengrundlage

Für die Programme aus Kapitel 2.2 und 2.3 werden Daten benötigt, die die Grundlage für die Algorithmen bilden. Die Daten liegen in einer Datei im „ptb“-Format vor. Beim „ptb“-Format handelt es sich um eine aus dem „ptx“-Format abgeleitete Struktur.

Beim „ptx“-Format handelt es sich um ein gängiges Austauschformat für Punktwolken unterschiedlichster Hersteller von Laserscannern. Bei diesem Format werden die Daten mittels dem „ASCII“-Zeichensatz[3] gespeichert.

Im Gegensatz zum „ptx“-Format liegen die Daten im „ptb“-Format in binärer Form vor das bedeutet, die Daten werden nicht im „ASCII“-Zeichensatz dargestellt und sind somit auch nicht mit einem Texteditor lesbar. Das „ptb“-Format wurde von Lupos3D entworfen, um die bei Laserscannern anfallenden Datenmengen möglichst effizient zu verwalten. Die binäre Darstellung der Daten spart im Gegensatz zur nicht binären Darstellung sehr viel Speicherplatz. Der Aufbau der Datendatei gliedert sich in einen Header- und einen Datenbereich. Die Konvertierung in das „ptb“-Format ermöglicht es, auch Nutzern von Computern mit geringem Hauptspeicher und geringer Prozessorleistung die Daten effizient zu verarbeiten. Die Tabelle 2-1 zeigt einen Vergleich der Formate „ptx“ und „ptb“. Der Vergleich stammt aus einer Präsentation[4] von Lupos3D an der Hochschule Bochum. Verglichen werden dabei sowohl die Dateigröße, als auch die Einlesegeschwindigkeit. Bei dem Datensatz handelt es sich um eine Punktwolke mit insgesamt 11.000.440 Messwerten.

	Dateigröße	Einlesegeschwindigkeit
„ptx“-Format	514 MB	115 Sekunden
„ptb“-Format	167 MB	22 Sekunden

Tabelle 2-1: Vergleich von Dateiformaten

Wie schon erwähnt, gliedert sich das „ptb“-Format in zwei Bereiche, dem Header- und dem Datenbereich. Die im Header gespeicherten Informationen geben Aufschluss über die Dimensionen der Daten, wie zum Beispiel die Anzahl von aufgenommen Punkten in einer Zeile und die Anzahl der Zeilen. Zusätzlich sind auch Aussagen über die Dimension eines einzelnen Punktes, wie Anzahl der Elemente pro Punkt, verfügbar. Ein Punkt kann laut der Definition des „ptb“-Formats aus einer unterschiedlichen Menge von Elementen bestehen, nach der Anzahl der Elemente wird eine Typisierung vorgenommen. Jedes Element in einem Typ ist ein „float“-Wert, es besteht somit aus vier Byte. Insgesamt sind vier Typen möglich, der Typ0 stellt lediglich die Position des Punktes mit einer

[3] ASCII – **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
[4] Vgl. (33)

„XYZ“-Koordinate, im lokalen Scannersystem, dar. Die angegebene Koordinate eines Punktes besteht aus drei Elementen. Bei jedem der drei Elemente handelt es sich um einen Wert „float“-Wert. Bei den folgenden Typen handelt es sich um Erweiterungen des Typ0. Beim Typ1 ist zusätzlich zum Typ0 ein Intensitätswert vorhanden. Dieser Wert beschreibt in der Bildverarbeitung den Grauwert und ist somit auch mit dem Helligkeitswert gleichzusetzen. Es handelt sich dabei um einen einzelnen „float“-Wert. Der Typ2 setzt sich zusammen aus dem Typ0 und den „RGB“-Farbinformationen[5]. Die zusätzlichen Informationen bestehen aus drei Elementen, eines für Rot (R), eines für Grün (G) und eines für Blau (B). Jedes der Elemente ist wiederum ein „float“-Wert. Der Typ3 ist eine Zusammensetzung aller drei vorherigen Typen. Der Typ besteht sowohl aus einer „XYZ“-Koordinate als auch aus einem Intensitätswert und den „RGB“-Farbinformationen.
Die zu jedem Punkt gespeicherte „XYZ“-Koordinate liegt im lokalen Scannersystem vor. Das bedeutet, die Koordinate bezieht sich mit ihrem Ursprung auf den Standpunkt des Laserscanners. Dadurch muss sie in ein globales/übergeordnetes Koordinatensystem transformiert werden bevor Berechnungen durchgeführt werden können. Die Parameter für diese Transformation sind in der Orientierungsmatrix im Header gespeichert. Bei der Orientierungsmatrix handelt es sich um eine „4x4“-Matrix mit neun Rotationsparametern, drei Translationsparametern und einer Maßstabsangabe. Alle 16 Werte der Orientierungsmatrix liegen als Double-Werte vor.

Der Datenbereich gliedert sich in Zeilen. In jeder Zeile sind so viele Punkte, mit den dazugehörigen Informationen gespeichert, wie es der Headerbereich vorsieht. Die Anzahl der Zeilen ist fest definiert und somit ist auch die Gesamtanzahl der Punkte bestimmt. Die Anordnung der Zeilen in der Datendatei spielt eine wichtige Rolle. Bei der Darstellung einer Datei, mit der Lupos3D Software „Viewer“, existieren zwei Arten. Zum einen eine 3D-Ansicht, bei dieser spielt die Reihenfolge der Zeilen in der Datendatei jedoch keine Rolle. Zum anderen eine 2D-Ansicht, bei der die Reihenfolge der Zeilen eine sehr wichtige Rolle spielt. Denn in der 2D-Ansicht werden die Intensitätswerte der Punkte als Grauwerte dargestellt, so wie sie aus der Datei kommen. In korrekter Reihenfolge ergibt sich für das Auge ein „sinnvolles“ Bild, in falscher Reihenfolge gegebenenfalls gar kein „logisches“ Bild.

2.2 Schnitte-Algorithmus

Der Algorithmus mit dem Namen „Schnitte“, ist in seiner Funktionsweise eine recht simple Funktion. Sie eignet sich daher gut für das Testen auf Parallelisierbarkeit und den Umgang mit großen Datenmengen. Der Algorithmus fällt bei der Parallelisierung in die Kategorie „peinlich parallel“ (engl.: embarrassing parallel). Das bedeutet, die Überführung des sequenziellen Algorithmus in einen parallelen ist nicht besonders kompliziert. Eine detaillierte Beschreibung der Kategorie „peinlich parallel" kann im Kapitel 5.3 nachgelesen werden. Der Algorithmus wird mit dem Ziel ausgeführt in dem Eingangslaserscan alle Punkte zu finden, die in einer gewissen Toleranz auf einer von Nutzer definierten Ebene liegen. Eine Ebene definiert sich durch drei Punkte, welche jeweils aus den Koordinaten X, Y und Z bestehen. Diese Koordinaten dürfen dabei nicht in einer Geraden liegen, da sonst keine Ebene definiert werden kann. Aus den ermittelten Punkten lassen sich beispielsweise Grundrisse oder auch Querschnitte generieren.

Die Ebene im mathematischen Sinn ist ein Grundbegriff aus der Geometrie. Es handelt sich dabei um ein unbegrenzt ausgedehntes, flaches Objekt. Die Ebene ist also immer zweidimensional. Die Bestimmung der Gleichung einer Ebene, die für diesen Algorithmus benötigt wird, kann auf mehrere Weisen durchgeführt werden. Für diesen Anwendungsfall ist jedoch nur eine Möglichkeit interessant,

[5] Geben das Mischungsverhältnis der drei Grundfarben Rot, Grün, Blau an und bestimmen so eine neue Farbe

die Berechnung der Ebenengleichung aus drei Punkten. Für diese Berechnung wird die hessische Normalform verwendet. Die hessische Normalform (siehe Formel 1) beschreibt eine Ebene im euklidischen Raum. Der Wert "$\vec{r}$" steht dabei für den Ortsvektor eines Punktes. Der Wert "$\overrightarrow{n_0}$" bezeichnet den normierten Normalenvektor, der vom Koordinatenursprung zur Ebene zeigt. Der Wert "d" kennzeichnet den Abstand des Koordinatenursprunges von der Ebene. Die hessische Normalform der Ebene ergibt sich somit aus dem Skalarprodukt des Ortsvektors eines Punktes mit dem normierten Normalenvektor minus dem Abstand der Ebene zum Koordinatenursprung. Um die Ebenengleichung zu bestimmen, muss also der normierte Normalenvektor und der Abstand der Ebene zum Koordinatenursprung berechnetet werden. Aus drei Punkten (A, B, C), die zu einer Ebene gehören und nicht auf einer geraden liegen, kann der normierte Normalenvektor mittels dem Kreuzprodukt der Richtungsvektoren ($\vec{a}$, $\vec{b}$, $\vec{c}$) dieser drei Punkte berechnet werden (siehe Formel 2). Der Abstand der Ebene vom Koordinatenursprung kann mittels des Richtungsvektors eines Punktes der Ebene und dem normierten Normalenvektor berechnet werden (siehe Formel 3). (Abschnitt nach (1))

Formel 1: hessische Normalform

$$\vec{r} \cdot \overrightarrow{n_0} - d = 0$$

Formel 2: Bestimmung des normierten Normalenvektors

$$\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$$
$$\overrightarrow{n_0} = \frac{\vec{n}}{|\vec{n}|}$$

Formel 3: Abstand der Ebene von Koordinatenursprung

$$d = \vec{a} \cdot \overrightarrow{n_0}$$

Die Verarbeitung des Eingangslaserscans erfolgt zeilenweise, das hat den Vorteil, dass der Arbeitsspeicher nicht mit Daten belastet wird, die noch gar nicht benötigt werden. Ein Nachteil besteht jedoch darin, dass ein Lesen der Daten als großer Block effizienter ist als ein zeilenweises Lesen. Beim zeilenweisen Lesen muss zuvor ein Zeiger auf die entsprechende Zeile in der Datendatei gesetzt werden und dieses benötigt Zeit. Die Abbildung 2-1 zeigt einen vereinfachten Ablauf des Algorithmus. Dazu wird jeder Punkt in einer gelesen Zeile über die im Header mitgelieferte Orientierungsmatrix vom Laserscannerkoordinatensystem in das Koordinatensystem der Ebene transformiert. Nun kann für jeden Punkt der Abstand zur berechneten Ebenengleichung bestimmt werden. Der Abstand eines Punktes zu einer Ebene wird mittels der hessischen Normalform berechnet (siehe Formel 4). Der Wert "s" steht dabei für den Abstand und der Wert "$\vec{p}$" für den Richtungsvektor des Punktes, für den der Abstand ermittelt werden soll. Liegt das Ergebnis dann wiederum innerhalb der angegebenen Toleranz zur Ebene, wird der Punkt als zugehörig gewertet und als Ergebnis ausgegeben.

Formel 4: Abstand eines Punktes von einer Ebene

$$s = \vec{p} \cdot \overrightarrow{n_0} - d$$

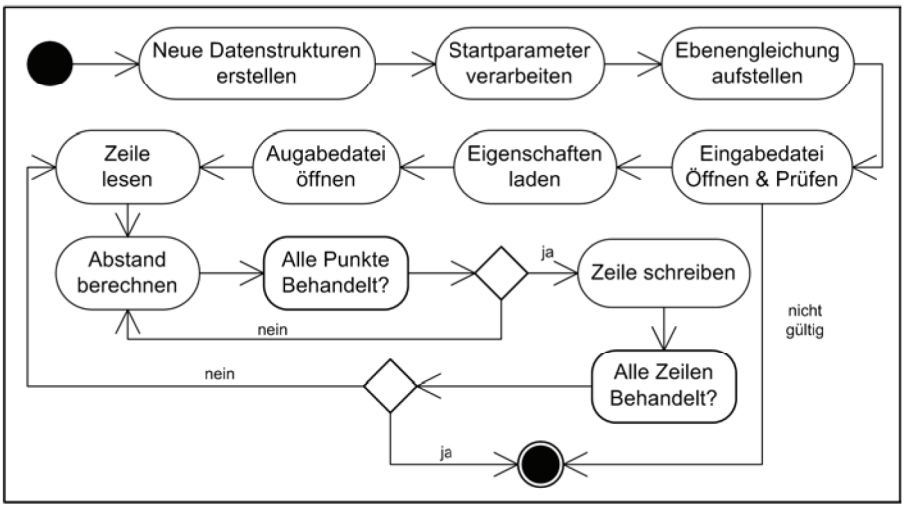

Abbildung 2-1: Schnitte - Ablauf

2.3 Photo-2-Scan

Beim Photo2Scan Algorithmus handelt es sich um einen komplexen und zeitintensiven Algorithmus. Dabei kommen Kombinationen verschiedener photogrammetrischer Berechnungen, wie zum Beispiel Transformationen vor. Der Algorithmus dient dazu, Fotos in einen Laserscan einzurechnen. Für den Betrachter ergibt sich als Ergebnis somit eine Ansicht, bei der die aufgenommen Flächen mit einer Textur hinterlegt sind.

Für die Ausführung des Algorithmus werden verschiedene Daten als Grundlage benötigt. Zum einen muss ein orientierter Laserscan im „ptb"-Format vorliegen. Orientiert heißt, dass Rotations-, Translations- und Maßstabsparameter bekannt seinen müssen, um eine Transformation vom lokalen Laserscannerkoordinatensystem in ein globales Koordinatensystem durchführen zu können. Eine zweite Gruppe von notwendigen Daten bezieht sich auf das einzurechnende Foto. Neben dem Foto an sich müssen auch die Parameter der inneren und äußeren Orientierung bekannt sein.

Die Parameter der inneren Orientierung beziehen sich auf die Aufnahmekamera und beschreiben das geometrische Kameramodell, die Abbildung 2-2 zeigt die Parameter an einem Modell einer Lochkamera. Einer der wichtigsten Parameter ist die Kamerakonstante, diese beschreibt den Abstand zwischen Projektionszentrum und der Bildebene der Kamera. Mit der inneren Orientierung wird somit die räumliche Lage des Projektionszentrums im Bildkoordinatensystem mit eventuell auftretenden Abbildungsfehlern beschrieben. Die Parameter der inneren Orientierung müssen für jedes Kamerasystem erneut bestimmt werden. Die Parameter der äußeren Orientierung beziehen sich auf die absolute räumliche Lage der Kamera in dem globalen Koordinatensystem des Laserscans. Die Parameter sind somit für die Orientierung des Fotos zuständig. Mit den Parametern wird beschrieben, wie ein Bild zu drehen und zu zerren ist, damit gemeinsame Bezugspunkte im Bildkoordinatensystem und dem globalen Koordinatensystem übereinstimmen. Die äußere Orientierung besteht somit aus sechs Parametern, drei Rotationen[6] und drei Translationen[7]. Die Abbildung 2-3 zeigt schematisch die äußere Orientierung und ihre Parameter. (Abschnitt nach (2 S. 9-10; 118-119; 235-236))

[6] Drehungen
[7] Verschiebungen

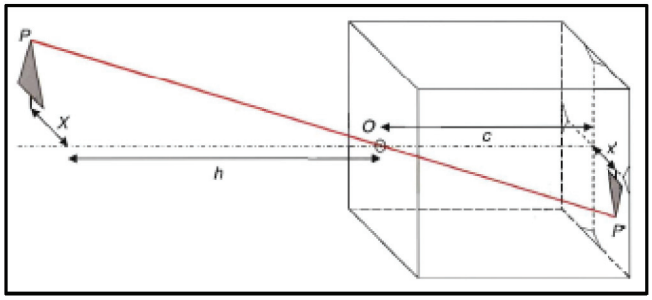

Abbildung 2-2: Innere Orientierung

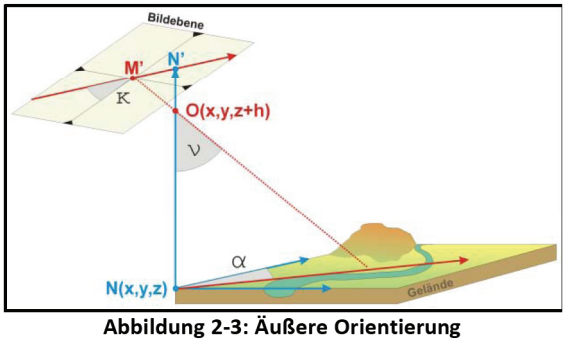

Abbildung 2-3: Äußere Orientierung

Die Verarbeitung des Eingangslaserscans erfolgt genau wie beim Schnitte-Algorithmus zeilenweise, dadurch kann die Belastung für den Arbeitsspeicher minimal gehalten werden, siehe Kapitel 2.2. Die Abbildung 2-4 zeigt den Ablauf des Algorithmus in einer groben Auflösung. Bevor mit der Verarbeitung begonnen werden kann, müssen noch Initialarbeiten vorgenommen werden. Dazu gehört zum einen das Auslesen der inneren und äußeren Orientierungsparameter aus den entsprechenden Orientierungsdateien, das Bestimmen der „RGB“-Farbinformationen des Fotos. Und zum anderen die Prüfung der Gültigkeit des Laserscans, anhand eines Prüfstrings sowie das Auslesen seiner Eigenschaften. Bei den zu bestimmenden Eigenschaften handelt es sich um Angaben über die Anzahl der Zeilen und Punkte pro Zeile im Laserscan, aber auch die Orientierungsparameter der Orientierungsmatrix des Laserscans. Ebenso muss der Typ, der in der Laserscandatei gespeicherten, Punkte bestimmt werden, siehe Kapitel 2.1. Mit den ermittelten Eigenschaften, wir nun eine neue Laserscandatei angelegt. Diese fungiert als Kopie des Eingangslaserscans, damit bei Problem keine irreversiblen Schäden an der Originaldatei auftreten. Die Kopie des Laserscans enthält als Ergebnis die Daten des originalen Scans und zusätzlich noch die eingerechneten „RGB“-Farbinformationen des Fotos.

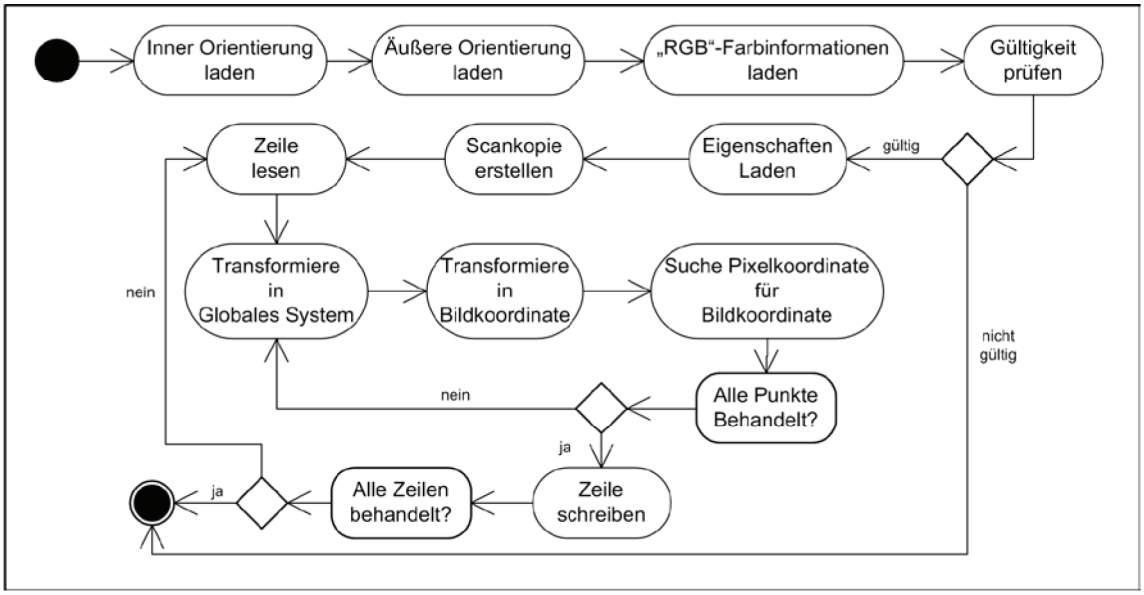

Abbildung 2-4: Photo2Scan - Ablauf

3 Grundlagen der Parallelisierung

Die folgenden Kapitel befassen sich mit der Theorie, die hinter einer Parallelisierung steckt, sowohl von der Seite der Hardware als auch der Software. Das Kapitel 4 behandelt später speziell die theoretischen Betrachtungen, die bei der Parallelisierung der in den Kapiteln 2.2 und 2.3 vorgestellten Algorithmen eine Rolle spielen.

3.1 Hardware

Die Hardware bildet die Grundlage für die Parallelisierung. Wenn diese Komponente schon den Weg für die parallele Ausführung von Anwendungen versperrt, lässt sich auch bei der Entwicklung von Software keine Parallelisierung durchführen.

3.1.1 Mehrkern-Prozessoren

Der Prozessor (auch CPU[8]) bildet das Herz eines Computers, wie er heute in fast jedem Haushalt vorkommt. In der Vergangenheit schlug nur eines dieser Herzen in einem Rechner, heutzutage scheinen dem jedoch fast keine Grenzen mehr gesetzt zu sein. Zwei oder vier Prozessoren in einem Desktop PC sind alltäglich geworden, doch geht es noch mehr. Dieses Kapitel beschreibt den Basisaufbau eines jeden Prozessors und geht anschließen auf den Aufbau von Mehrkern-Prozessoren ein.

Ein Prozessor besteht nach der „von-Neumann-Architektur“[9] aus verschiedenen Komponenten, die mit ihrer Zusammenarbeit das Objekt des Prozessors bilden. Zu den Bestandteilen gehören verschiedene Register (Daten-, Adress-, Steuerregister), ein Rechenwerk (ALU[10], Datenprozessor), ein Steuerwerk (CU[11], Befehlsprozessor) sowie Datenleitungen (Busse). In der Abbildung 3-1[12] kann der schematische Aufbau einer CPU und peripherer Komponenten betrachtet werden. Die Register sind prozessorinterne Speicher und dienen zur Aufnahme von temporären Daten, Befehlen, Ergebnissen und Adressen. Sie stehen an der ersten Stelle der Speicherhierarchie, da sie sehr nahe und schnelle Verbindungen zu anderen Prozessorkomponenten bilden. Nur mit Daten, die in Registern vorgehalten werden, können auch direkt Operationen ausgeführt werden. Das Steuerwerk ist für die Kontrolle der Befehlsausführung zuständig. Es legt somit fest, wo welche Aktion in einem Takt ausgeführt wird. Das Rechenwerk ist für die elementaren Operationen des Rechners zuständig. Es führt also arithmetische und logische Operationen (z. B.: Zahlen addieren und AND- bzw. OR-Verknüpfungen) aus. Das Rechenwerk besitzt jedoch keine eigene Steuerung. Bei den verschiedenen Bussen handelt es sich um Signalleitungen, die den Prozessor mit anderen Komponenten verbinden. Unterschieden wird dabei in Daten-, Adress- und Steuerbus. Mithilfe des Datenbusses tauscht der Prozessor Informationen mit dem Arbeitsspeicher aus, dabei handelt es sich um Informationen für das Daten- und Steuerregister. Der Adressbus enthält die Adresse einer Speicherzelle des Arbeitsspeichers, auf die je nach Signal des Steuerbusses Daten vom Datenbus geschrieben oder gelesen werden. Der Steuerbus ist für die Verbindung zwischen dem Prozessor und peripheren Komponenten. Über diese Verbindung ist es dem Prozessor möglich, die Kontrolle zu übernehmen.

---

[8] CPU – **C**entral **P**rocessing **U**nit
[9] Siehe z. B.: **Hartmut Ernst**: Grundkurs Informatik; Aufl. 4; Seite 200-202; ISBN: 978-3-8348-0362-7
[10] ALU – **A**rithmetic **L**ogical **U**nit
[11] CU – **C**ontrol **U**nit
[12] Nach (3 S. 4)

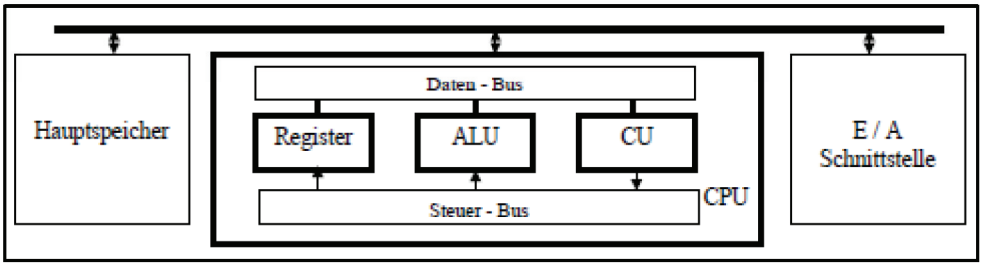

Abbildung 3-1: Aufbau einer CPU

Bei Mehrkern-Prozessoren handelt es sich um Mikroprozessoren, die auf einem einzigen Chip mehrere vollständige Hauptprozessoren besitzen. Die Ressourcen die ein Hauptprozessor, wie er zuvor beschrieben wurde, besitzt sind bei Mehrkern-Prozessoren auch mehrfach vorhanden. Eine Ausnahme bilden die Busse und der Cache. Wobei beim Cache mehrere Möglichkeiten existieren, die Prozessoren können sich einen Cache-Speicher teilen oder auch jeder über seinen eigenen verfügen. Gängige Mehrkern-Prozessoren sind heutzutage Dual-Core (zwei Kerne) und Quad-Core (vier Kerne). Mehrkern-Prozessoren gewannen seit 2005 immer mehr an Bedeutung, denn eine weitere Steigerung der Leistung von Prozessoren war ab einer Taktfrequenz von vier Gigahertz nicht mehr so leicht möglich wie zuvor. Die Wärmeentwicklung solcher Prozessoren machte den Herstellern zu schaffen. Die Entwicklung ging also hin zu mehr Prozessoren mit geringer Taktfrequenz. Ein weiterer Vorteil ergab sich in der Produktion der Mehrkern-Prozessoren, es ist kostengünstiger einen einzelnen Chip mit mehreren Ressourcen zu produzieren (Multicore-Prozessor) als mehrere Chips mit nur einem Kern (Muli-Prozessor). Anders gesagt ist es effizienter, wenn man mehrere Hauptprozessoren auf einem Chip in einem Socket des Mainboards unterbringen kann, als wenn man mehrere Sockets auf dem Mainboard benötigt. Die Abbildung 3-2[13] dient der Verdeutlichung der unterschiedlichen Architekturen. Des Weiteren existieren unterschiedliche Mehrkern-Techniken. Zum einen die homogenen Mehrkern-Prozessoren und zum anderen die heterogenen Mehrkern-Prozessoren. Bei heterogenen sind unterschiedliche Kerne in einem Chip verbaut, der Chip wirkt für das Betriebssystem trotz mehrerer Kerne wie ein einziger Prozessor. Sinnvoll wird diese Technik dann, wenn die Kerne spezialisiert sind, zum Beispiel für Decodierung und Verschlüsselung. Die homogene Technik ist am weitesten verbreitet, dabei sind alle Kerne identisch und erscheinen dem Betriebssystem auch wie mehrere Prozessoren. Diese Technik macht es besonders Hard- und Softwareentwicklern einfach, denn sie können über einem gemeinsamen Speicher der Prozessoren arbeiten. Die Einführung von Mehrkern-Prozessoren im Desktopbereich eröffnete neue Möglichkeit im Bereich der Softwareentwicklung. Die Parallelität von Software spielte eine größere Rolle und auch das parallele Ausführen mehrerer Programme macht das Arbeiten für den Nutzer flüssiger.

Heute existieren, wie auch schon zuvor, zwei große Anbieter von Prozessoren AMD und Intel. Beide Hersteller führen die unterschiedlichsten Produkte im Bereich der Mehrkern-Prozessoren. Von der Leistung her nehmen sich die Produkte jedoch nichts, lediglich bei Preis sind Unterschiede erkennbar. AMD-Prozessoren sind in der Anschaffung etwas günstiger als Intel-Prozessoren. (Kapitel nach (3; 4; 5; 6; 7; 8 S. 21-23))

[13] Nach (7 S. 6)

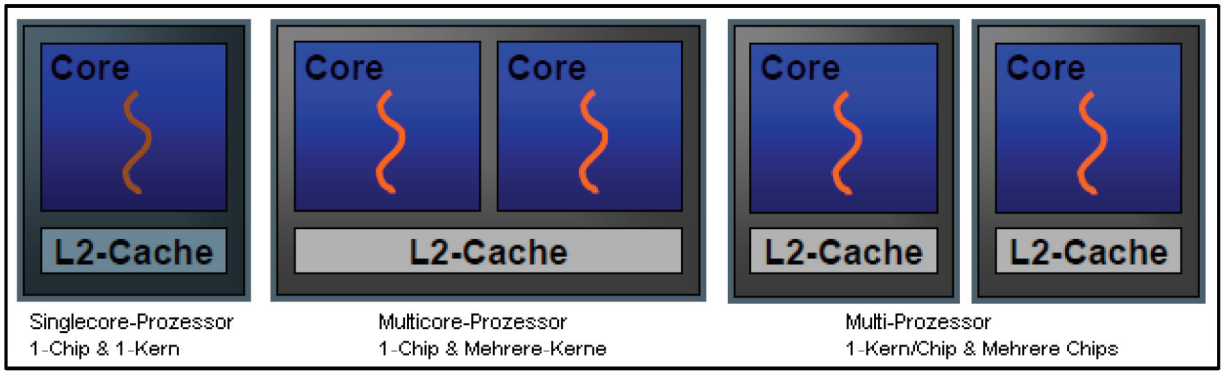

Abbildung 3-2: Mehrkern-Prozessor Architekturen

3.1.2 Cluster

Ein Cluster ist ein weiterer Ansatz in der Architektur der parallelen Verarbeitungen. Der im Kapitel 3.1.1 vorgestellte Ansatz beruht auf dem Prinzip des gemeinsamen Speichers von Prozessoren, ein Cluster hingegen setzt auf einen verteilten Speicher. Praktisch gesehen enthalten Systeme mit einem gemeinsamen Speicher selten mehr als 64 Prozessoren, bei einem System mit verteiltem Speicher lassen sich viel mehr Prozessoren zusammen schließen. Jeder Prozessor verfügt in einem solchem System über einen eigenen lokalen Speicher, auf den nur er zugreifen kann. Eine solche Einheit von Prozessor und Speicher wird auch Knoten genannt. Bei der Kommunikation zwischen den Knoten setzen Systeme mit verteiltem Speicher auf Verbindungsnetzwerke. Der Datentransport über solche Netzwerke ist jedoch viel langsamer als bei Systemen mit einem gemeinsamen Speicher. Wo der Transport lediglich zwischen CPU und Speicher erfolgt. Somit eignen sich Systeme mit verteiltem Speicher aufgrund der Transportproblematik nicht für sehr feingranulare Parallelisierungen. Die Transporteffizienz ist beschränkt durch die Eigenschaften des Verbindungsnetzwerkes (Bandbreite, Latenz) und der Menge der zu übertragenden Daten. Die Bandbreite bestimmt, mit welcher Rate Daten von einem Knoten zum anderen übertragen werden können und die Latenz ist die vergangene Zeit, bis das erste gesendete Bit beim Empfänger eintrifft.

Ganz grob könnte man sagen, dass ein Cluster eine Ansammlung von Standard-PCs ist, die zur Gewinnung höher Rechenleistungen mittels eines Netzwerks verbunden sind. Wie schon angesprochen wird die Einheit von Prozessor und Speicher als Knoten bezeichnet, jedoch existieren zwei unterschiedliche Arten von Knoten. Die Abbildung 3-3[14] zeigt wie ein kleines Cluster aufgebaut ist und welche Arten von Knoten in einem Cluster vorkommen. Ein Rechen Knoten (engl. compute node) ist der hart arbeitende Teil eines Clusters, auf ihm werden die rechenintensiven Probleme gelöst. Die zweite Knotenart ist der Server Knoten (engl. server node), dieser stellt die für den Betrieb des Clusters benötigte Infrastruktur zur Verfügung. So verteilt der Server Knoten ein Netzwerkdateisystem an die Rechen Knoten, dass es den einzelnen Knoten ermöglicht auf den gleichen Datenbestand zuzugreifen. Ein Server Knoten stellt auch die zentrale Arbeitsplattform eines Cluster Nutzers dar. Bei den meisten Clustern eingesetzte Verbindungsnetzwerke sind sogenannte „switched networks“ mit einer sternförmigen Topologie. Diese Netzwerke besitzen aufgrund ihrer Topologie eine hohe Flexibilität und bieten auch noch weitere Vorteile. So steigt beispielsweise die Zahl der Leitungen nur noch linear. Eine Vollvernetzung wird durch ein eingesetztes Switch elektronisch simuliert. Jeder Knoten ist nur mit dem Switch verbunden, eine Weiterleitung der Datenpakete kann somit auch nur über das Switch erfolgen. Die Verzögerung bei der Latenzzeit, die ein Switch bei der Weiterleitung verursacht, bleibt trotzdem nur sehr gering. Der Vorteil des Switch

[14] Nach (8 S. 52)

ist, dass bei einer Kommunikation zwischen zwei Knoten die anderen Leitungen unberührt bleiben. Dadurch ist es möglich, dass mehrere unabhängige Knoten-Paare gleichzeitig mit der vollen Bandbreite kommunizieren können.

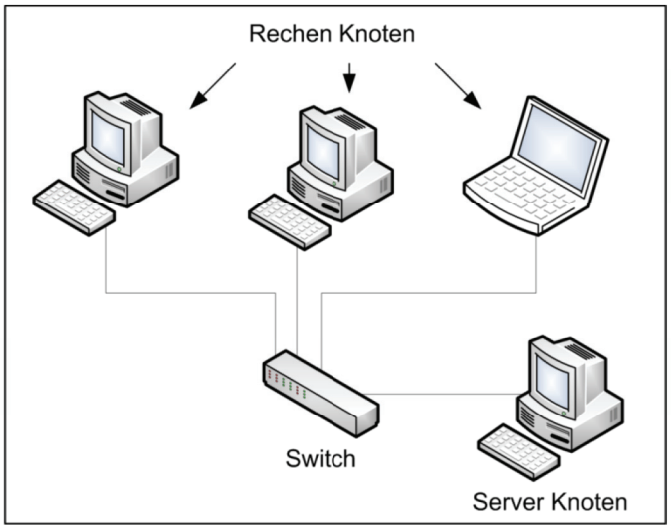

Abbildung 3-3: Aufbau eines kleinen Clusters

Um ein Cluster in einem Computernetzwerk einzurichten, bedarf es einer speziellen Cluster-Software. Diese sind heutzutage schon als Open Source Lösungen vorhanden. (Kapitel nach (8 S. 23-27, 51-52))

3.1.3 Grafikkarten

Die Grafikkarten sind ein Bestandteil jedes Computers, sie sind für die Verarbeitung grafischer Daten zuständig. Dabei beruhen Sie auf derselben Technologie wie die Prozessoren, die im Kapitel 3.1.1 vorgestellt wurden, jedoch existieren kleine Unterschiede. Im Folgenden soll nun die Entwicklung der Grafikkarten und ihre Beziehung zu den CPUs genauer betrachtet werden.

Die Hersteller von Prozessoren (CPUs) lieferten sich, in den 1990er Jahren, einen richtigen Krieg um die höheren Taktraten und stießen dabei immer mehr an die physikalischen Grenzen der Machbarkeit. Bei den Herstellern von Grafikchips (GPUs[15]) war von einer solchen Entwicklung jedoch nichts merkbar, wie die Abbildung 3-4 zeigt. Die Leistungsfähigkeit der GPUs gemessen an den Giga FLOPS[16] stieg immer mehr an und hatte bald einen enormen Vorsprung vor den CPUs.

[15] GPU – **G**raphics **P**rocessing **U**nit

[16] FLOPS – **Fl**oating Point **O**perations **P**er **S**econd (Gleitkommaoperationen pro Sekunde; Maßeinheit für die Geschwindigkeit von Prozessoren)

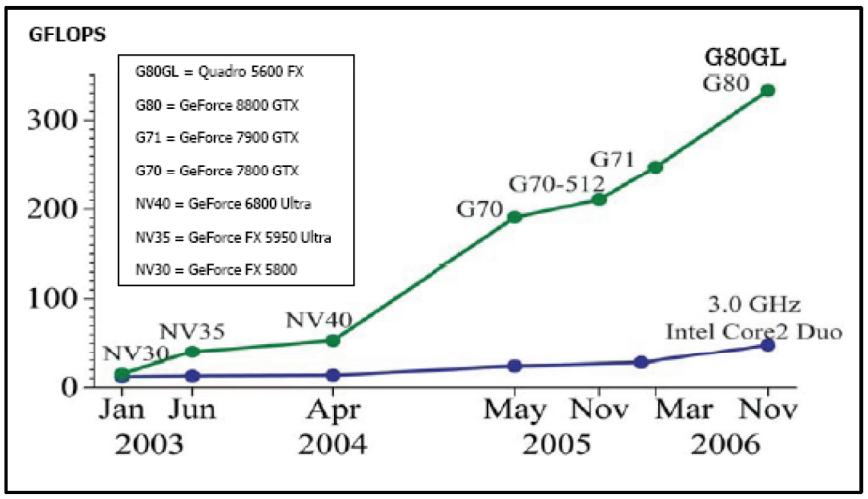

Abbildung 3-4: Leistungsentwicklung von GPUs und CPUs

Die Probleme, mit denen die CPU-Hersteller zu kämpfen hatten, spielten bei den Grafikkarten kaum eine Rolle. Das liegt an der Nutzungsart für die die Prozessoren entwickelt wurden. CPUs sind dafür konzeptioniert, die größte mögliche Leistung aus einem Befehlssatz herauszuholen. Dabei arbeiten CPUs über den unterschiedlichsten Daten und führen willkürliche Speicherzugriffe und Verzweigungen durch. Das Problem der CPUs ist, dass die parallele Ausführung von Befehlssätzen begrenzt ist. Auch eine Erhöhung der Anzahl von Verarbeitungskomponenten bringt nichts, wenn diese ungenutzt bleiben. Die GPUs haben dahingegen nur einen sehr kleinen Befehlssatz. Sie sind lediglich für die Erstellung von Pixeln aus Polygonen zuständig. Da Pixel und Polygone aber unabhängig voneinander sind, können sie von parallelen Komponenten bearbeitet werden. Somit ist es Grafikkartenherstellern möglich mehr ALUs auf einen Chip unterzubringen als CPU-Herstellern. Zum Vergleich dient die Abbildung 3-5 in der grob die Anzahl der ALUs auf einer CPU und einer GPU verglichen werden. Ein weiterer Unterschied zwischen CPU und GPU ist, dass aufgrund der Spezialisierung der GPU eine intelligente Speicherorganisation viel räumlichen Platz auf einem Chip sparen kann. Eine GPU kommt mit nur sehr wenig Cache-Speicher aus, nur einige Kilobyte. Die Architektur der Chips und ihre Spezialisierung auf bestimmte Aufgaben sind somit der Grund für die immer noch steigende Leistungsfähigkeit der Grafikkarten im Gegensatz zu den CPUs. (Kapitel nach (9))

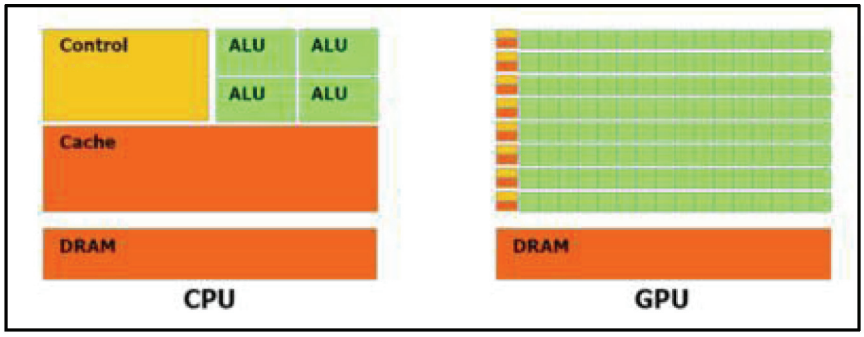

Abbildung 3-5: ALUs auf einer CPU und einer GPU

3.2 Entwurfsmuster

Die folgenden Kapitel beschreiben eine Reihe von Möglichkeiten, zur Erstellung paralleler Anwendungen in der Ebene der Softwareentwicklung.

3.2.1 Allgemeine Entwurfsmuster

Als ein Entwurfsmuster wird die Kombination eines Problems und einer Strategie zur Lösung des Problems bezeichnet. Bei objektorientierten Entwurfsmustern wird die Lösungsstrategie mittels Objekten und Schnittstellen ausgedrückt. Die beschriebenen Muster stellen somit generalisierte Lösungsideen für wiederkehrende Probleme dar. Dabei handelt es sich nicht um konkrete fertiggestellte (fertigcodierte) Lösungen, sondern um die Beschreibung eines Lösungswegs. Die Beschreibung eines Musters kann sich, zum einfacheren Austausch zwischen Entwicklern, aus verschiedenen Standardinformationen zusammensetzen. Ein festgelegter Standard für den Austausch existiert jedoch nicht. Als kleinste Form für den Austausch von Entwurfsmustern gilt die Angabe von vier grundlegenden Informationen. (Abschnitt nach (10 S. 2-4; 11 S. 67-68; 12 S. 641)

- Name des Musters
- Problemabschnitt
- Lösungsabschnitt
- Konsequenzenabschnitt

Bei dem Namen des Musters sollte es sich um ein prägnantes Stichwort handeln, welches die Identifikation des Problems eindeutig macht. Der Problemabschnitt beschreibt, wo das Entwurfsmuster angewendet werden kann. Dem Entwickler wird in dem Abschnitt eine Liste von Bedingungen aufgezeigt, welche erfüllt sein sollten, damit eine sinnvolle Anwendung des Musters stattfinden kann. Im Lösungsabschnitt wird beschrieben, aus welchen Elementen der Entwurf besteht und welche Beziehungen, Zuständigkeiten und Interaktionen existieren. Eine konkrete Implementierung enthält der Abschnitt jedoch nicht, vielmehr handelt es sich dabei um eine abstrakte Beschreibung des Lösungsansatzes. Im Konsequenzenabschnitt werden die Vor- und Nachteile des Entwurfsmusters diskutiert. Dieses ist bei der Entscheidung für ein Entwurfsmuster oder eine Alternative von großer Wichtigkeit. (Abschnitt nach (10 S. 3))

Im Allgemeinen sollte ein Entwurf immer unabhängig von einer Programmiersprache sein, dass gleiche gilt auch für Entwurfsmuster. In der Praxis kann dieses jedoch nicht immer gewährleistet werden, da manche Muster sich speziell auf Abhängigkeiten einer Programmiersprache beziehen. Bei einem Entwurfsmuster, welches an eine Programmiersprache gebunden ist, lässt sich für einen Entwickler viel leichter erkennen, welche Stellen des Musters besonders kompliziert oder besonders leicht zu implementieren sind. Wenn ein Entwurfsmuster eine starke Abhängigkeit von einer speziellen Sprache aufweist und der Entwickler dieses besonders betonen möchte, so spricht man von einem Idiom. (Abschnitt nach (10 S. 4; 11 S. 67))

Bei der Katalogisierung von Entwurfsmustern spielt die Beschreibung eines speziellen Musters eine Rolle. Es existiert eine Vielzahl von Mustern und Variationen dieser Muster. Für Entwickler ist es daher interessant zu wissen, wann ein neu entwickeltes Entwurfsmuster wirklich neu ist, oder wann es sich lediglich um eine Variation eines schon bekannten Musters handelt. In verschiedenen veröffentlichten Katalogen sind bekannte Muster verzeichnet. Mithilfe dieser Kataloge lassen sich ungefähr 23 allgemeine Entwurfsmuster spezifizieren, siehe Tabelle 3-1[17]. Daneben existieren noch spezielle Muster für bestimmte Aufgabebereiche. Je nach Eigenschaft und Anwendungsbereich teilen

[17] Nach (12 S. 642)

die Kataloge die Entwurfsmuster in drei Bereiche ein, Erzeugungsmuster, Strukturierungsmuster und Verhaltensmuster. Jeder dieser drei Bereiche lässt sich wiederum in 2 Abschnitte teilen, dem klassenbasierten und dem objektbasierten Muster. (Abschnitt nach (11 S. 68))

	Erzeugungsmuster	**Strukturmuster**	**Verhaltensmuster**
Klassenbasiert	Fabrikmethode	Adapter (Klasse)	Schablonenmethode Interpreter
Objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter (Objekt) Brücke Kompositum Dekorierer Fassade Fliegengewicht Proxy	Zuständigkeitskette Befehl Iterator Vermittler Memento Beobachter Zustand Strategie Besucher

Tabelle 3-1: Liste der Entwurfsmuster

Die sogenannten Erzeugungsmuster sind für die Erstellung von Objekten zuständig. Sie ermöglichen es den Erzeugungsprozess eines Objekte zu verstecken, damit machen Erzeugungsmuster ein System unabhängig von der Erzeugung, Zusammensetzung und Repräsentation von Objekten. Bei dieser Art von Mustern treten zwei immer wiederkehrende Leitmotive auf. Zum einen kapseln Erzeugungsmuster Wissen der konkreten, vom System, verwendeten Klassen. Zum anderen verstecken sie, wie Exemplare von Klassen erzeugt und zusammengesetzt werden. Somit stammt alles was eine Anwendung über ein Objekt weiß von definierten Schnittstellen. Erzeugungsmuster sind bedeutend für Systeme, die mehr von Objektkompositionen als von Vererbung abhängen. Die Programmierung solcher Systeme bewegt sich somit weg von einem festgelegten Verhalten, hin zur Definition von kleineren grundlegenden Verhaltenseinheiten. Diese Einheiten können dann beliebig zusammengestellt werden, um ein gewünschtes Verhalten zu erzeugen. Klassenbasierte Erzeugungsmuster setzen auf Vererbung, um die Klasse des zu erstellenden Objektes zu variieren. Bei objektbasierten Erzeugungsmustern wird die Erzeugung eines Objektes an ein anderes Objekt delegiert. (Abschnitt nach (10 S. 87-92; 12 S. 641))

Strukturierungsmuster sind alle Entwurfsmuster, die sich mit der Komposition von Klassen und Objekten befassen. Somit erzeugen einzelne simplere Klassen oder Objekte größere Strukturen. Klassenbasierte Muster nutzen die Vererbung, um Schnittstellen oder Implementierungen zusammenzufassen. An der Mehrfachvererbung lässt sich das verdeutlichen. Damit werden mehrere Klassen zu einer einzigen vereinigt, welche im Ergebnis die Eigenschaften der Elternklassen aufweist. Das ist besonders hilfreich, wenn unabhängig voneinander entwickelte Bibliotheken zusammenarbeiten müssen. Objektbasierte Strukturmuster beschreiben Mittel und Wege für die Zusammenführung von Objekten, zur Gewinnung neuer Funktionalitäten. Muster mit einem objektbasierten Ansatz sind flexibler als klassenbasierte Muster. Denn Objektkompositionen machen es möglich während der Laufzeit Änderungen vorzunehmen, während Klassenkompositionen statisch sind. (Abschnitt nach (10 S. 149-150; 12 S. 642))

Unter die Kategorie der Verhaltensmuster fallen alle Entwurfsmuster, die sich mit Algorithmen oder der Zuweisung von Zuständigkeit an Objekte befassen. Neben der Beschreibung von Mustern für Objekte und Klassen gehen Verhaltensmuster auch auf Interaktionen zwischen Objekten und Klassen ein. Sie dienen somit der Erläuterung von Kontrollflüssen für den Entwickler, da diese zur Laufzeit bei einigen Anwendungen möglicherweise nur schwer nachvollziehbar sind. Bei

klassenbasierten Verhaltensmustern wird das Prinzip der Vererbung genutzt, um ein Verhalten zu verteilen. Dabei wird der eigentliche Algorithmus schrittweise definiert. Jede Unterklasse ruft dafür entweder abstrakte oder primitive Operationen auf. Während bei klassenbasierten Verhaltensmustern, wie schon erwähnt, auf Vererbung gesetzt wird, so wird bei objektbasierten Verhaltensmustern auf das Prinzip der Objektkomposition gesetzt. Es wird dafür definiert, wie eine Gruppe von Objekten an einer Aufgabe arbeitet, die für ein Objekt alleine nicht lösbar wäre. Wichtig ist dabei zu wissen, wie genau die Zusammenarbeit aussieht. Kennt ein Objekt auch alle anderen Objekte oder lediglich die Referenz auf einen Nachfolger. (Abschnitt nach (10 S. 243-244; 12 S. 642))

3.2.2 Parallele Entwurfsmuster

Wie schon im Kapitel 3.2.1 erläutert, befassen sich Entwurfsmuster mit wiederkehrenden Problemen und der Schaffung wiederverwendbarer Lösungen für diese Probleme. Nicht anders ist es Entwurfsmustern für parallele Anwendungen. Die Muster bringen auch bei diesem Anwendungsgebiet Vorteile in der Entwicklung neuer Applikationen, besonders in dem Bezug der Entwicklungszeiteffizienz. Viele Konzepte in der parallelen Programmierung arbeiten oft mit sogenannten „Task-farming“ oder einem „Divide and Conquer“-Prinzip. Programme, die auf der Basis dieser Konzepte entwickelt wurden und werden, benötigen viel Zeit für die Entwicklung. Es taucht bei der parallelen Programmierung immer wieder das Problem auf, dass der sequenzielle Code nicht vom parallelen Code getrennt wird. Das erhöht die Komplexität einer Applikation enorm und führt somit auch zu einem weiteren daraus unvermeidlichen Problem. Falls in einem parallelen Programm einmal mit einer anderen parallelen Struktur (Muster) experimentiert werden sollte, erfordert das immer viel Aufwand und auch Wissen des Entwicklers.

In dem Dokument aus der Quelle (13) wird ein System für parallele Programmierung beschreiben, genannt „DPnDP“[18]. Dabei handelt es sich um ein Entwurfsmuster orientiertes System. Es separiert parallele Strukturen, wie zum Beispiel Synchronisation, Kommunikation und das Prozess-Prozessor-Mapping, vom zu parallelisierenden Code. Implementiert in dieses System sind Unteranderem vier parallele Entwurfsmuster, die Pipeline oder auch "Chain of Responibiliy", ein Master-Slave-Muster, ein Replizierungsmuster (engl. Replicable) und ein Muster für das Teile-und-Herrsche-Prinzip (engl. Divide and Conquer). (Kapitel nach (14; 15; 16; 17))

[18] DPnDP: Design Pattern and Distributed Processes

3.2.2.1 Replizierungsmuster

Das Replizierungsmuster wird verwendet, wenn Operationen durchgeführt, werden die aufgrund von Abhängigkeiten an eine globale Datenmenge gebunden sind, siehe Abbildung 3-6[19]. Dabei wird die globale Datenmenge repliziert und an alle beteiligten Operationen weitergeleitet. Diese Operationen arbeiten eigenständig an der Lösung einer speziellen Aufgabe, das Ergebnis aller Operationen muss anschließend wieder zu einer gemeinsamen Lösung zusammengefasst werden. In seiner Struktur gleicht das Muster von außen einem einzigen Prozess, im Inneren besteht es jedoch aus vielen Kopien des einzigen Prozesses, die eigenständig arbeiten, siehe Abbildung 3-7[20].

Zusätzlich zu den eigenständigen, internen Prozessen besteht das Replizierungsmuster noch aus einem Koordinator, dieser agiert als eine Schnittstelle zwischen der Außenwelt und den internen Prozessen. Dabei kontrolliert er die Input- und Output-Operationen und übernimmt die Zuweisung des Inputs an einen internen Prozess. Nach Beendigung der Operationen sendet jeder der internen Prozesse das Ergebnis zurück an den Koordinator und dieser leitet es an den Empfänger weiter. (Kapitel nach (14; 15; 16; 17))

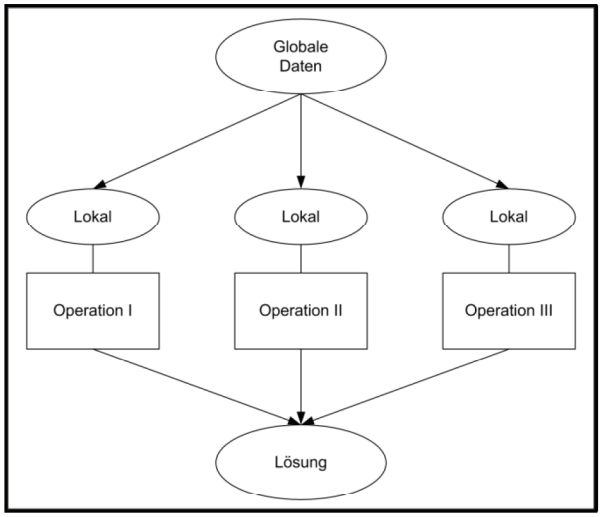

Abbildung 3-6: Datenorganisation Replizierungsmuster

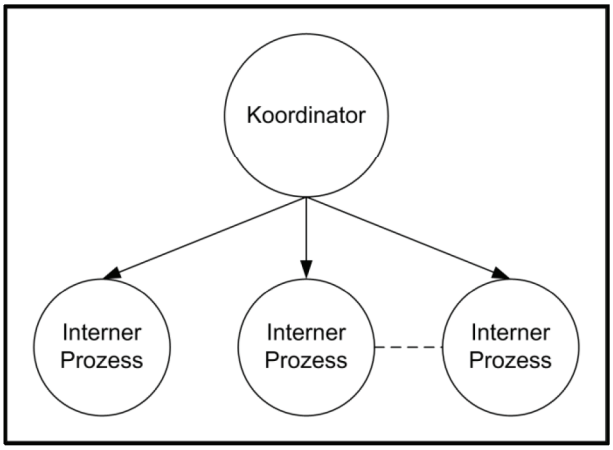

Abbildung 3-7: Aufbau Replizierungsmuster

3.2.2.2 Divide-and-Conquer-Muster

Das Divide-and-Conquer-Muster setzt auf das Prinzip der Teilung. Ein Problem wird bei diesem Ansatz so lange in kleinere Teilprobleme zerlegt, bis diese beherrschbar sind. Dabei ist im Sinne der parallelen Programmierung darauf zu achten, dass die Teilprobleme alle individuell bearbeitet werden können. Die Ergebnisse der einzelnen Berechnungen werden anschließend zu einem gemeinsamen Ergebnis zusammengefasst. Die Abbildung 3-8[21] zeigt den Ablauf eines mit dem Divide-and-Conquer-Muster gelösten Problems. Der Aufbau eines Musters der Größe 2x2 ist in Abbildung 3-9[22] erkennbar. Die Parameter der Größe stehen für die Breite und die Höhe des Musters. Bei jedem Kreis, der in der -Abbildung 3-4- zu sehen ist, handelt es sich um einen eigenständigen Prozess. Insgesamt bildet das Divide-and-Conquer-Muster somit einen Baum von eigenständigen Prozessen, in dem die Eltern-Prozesse die Arbeit rekursiv an die Kinder-Prozesse weiterleiten. Hat ein Kinder-Prozess seine Arbeit erledigt, sendet dieser das Ergebnis an seinen Eltern-Prozess zurück. Dieser wartet, bis alle seine Kinder-Prozesse ein Ergebnis zurückgaben, und fasst die Teilergebnisse zu

[19] Nach (15 S. 9)
[20] Nach (14 S. 9)
[21] Nach (15 S. 11)
[22] Nach (14 S. 9)

einem zusammen und leitet dieses wiederum weiter an seinen Eltern-Prozess. Die Wurzel des Divide-and-Conquer-Baumes bildet ein Prozess, der als Interface für das Entwurfsmuster fungiert. (Kapitel nach (14; 15; 16; 17))

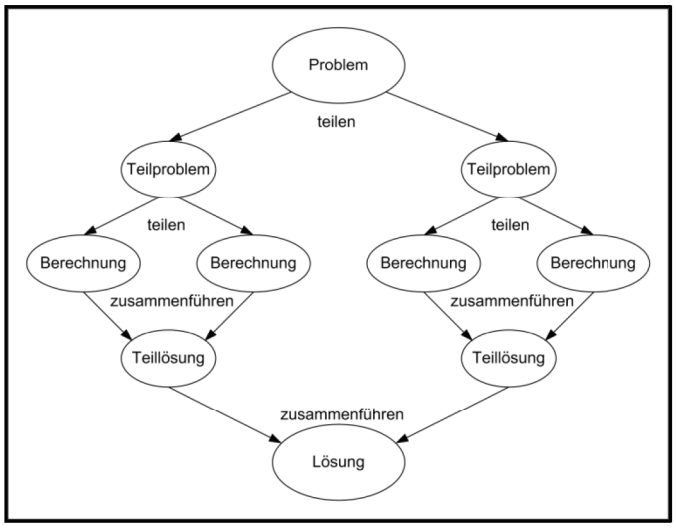

Abbildung 3-8: Divide-and-Conquer Ablauf

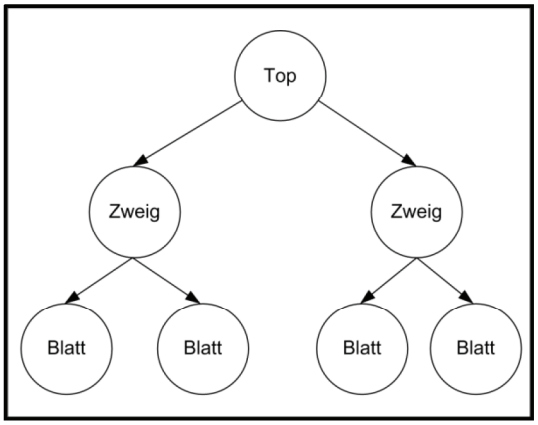

Abbildung 3-9: Divide-and-Conquer Aufbau

3.2.2.3 Master/Worker-Muster

Beim Master/Worker-Muster (auch Boss/Worker oder Master/Slave) handelt es sich um ein sogenanntes „Task-farm"-Muster. Es besteht aus zwei Komponenten dem Master und dem Worker. Bei beiden Komponenten handelt es sich um eigenständige Prozesse. Durch die Verfügbarkeit mehrerer Worker können Aufgaben parallel verarbeitet werden. Der Master kontrolliert die Verteilung von Aufgaben an die Worker-Komponenten, in dem er entsprechenden Anfragen an einen Worker erstellt und versendet. Nach der abgeschlossenen Bearbeitung einer Aufgabe durch einen Worker sendet dieser das Ergebnis zurück an den Master. Nach dem der Master alle Teilergebnisse empfangen hat, wird aus diesen ein Gesamtergebnis bestimmt. Die Kommunikation zwischen Master und Worker erfolgt asynchron das bedeutet, der Master steht immer nur mit einem Worker in Verbindung. Das Muster kann auf zwei Arten aufgebaut werden. Zum einen in seiner ursprünglichen Form (Abbildung 3-10 Grafik ‚a') in der ein Master über die gesamte Laufzeit die Worker überwacht und kontrolliert und zum Andern in einer weiterentwickelten Form (Abbildung 3-10 Grafik ‚b') bei der sich der Master nach der Verteilung der Aufgaben an die Worker zusätzlich als Worker einreiht und somit danach gleichberechtigt ist, siehe Abbildung 3-10[23] auf Seite 17. Die Verteilung der Last (Aufgaben) kann bei dem Master/Worker-Muster sowohl statisch als auch dynamisch erfolgen. Bei der statischen Verteilung werden alle Aufgaben zu Beginn der Berechnungen verteilt und bei der dynamischen werden die Aufgaben zur Laufzeit verteilt. Die dynamische Verteilung ist dann sinnvoll, wenn die Anzahl der Aufgaben zu Beginn der Anwendung unbekannt ist oder die Anzahl der Aufgaben die Anzahl der maximal möglichen Worker (Prozessoren oder Rechner im Cluster) überschreitet. In dem Fall der dynamischen Verteilung würde ein fertiger Worker beim Master eine neue Aufgabe aus einem Pool von Aufgaben anfragen. Somit passen sich Anwendungen mit dynamischer Verteilung an Änderungen des Systems an und stehen damit für robuste Anwendungen mit hoher Rechengeschwindigkeit und hohem Skalierbarkeitsgrad. Jedoch kann der Master bei einer großen Anzahl von Workern ein Flaschenhals werden, weil zu viele Worker gleichzeitig neue Aufgaben anzufragen versuchen. (Kapitel nach (14; 15; 16; 17))

[23] Nach (16 S. 57)

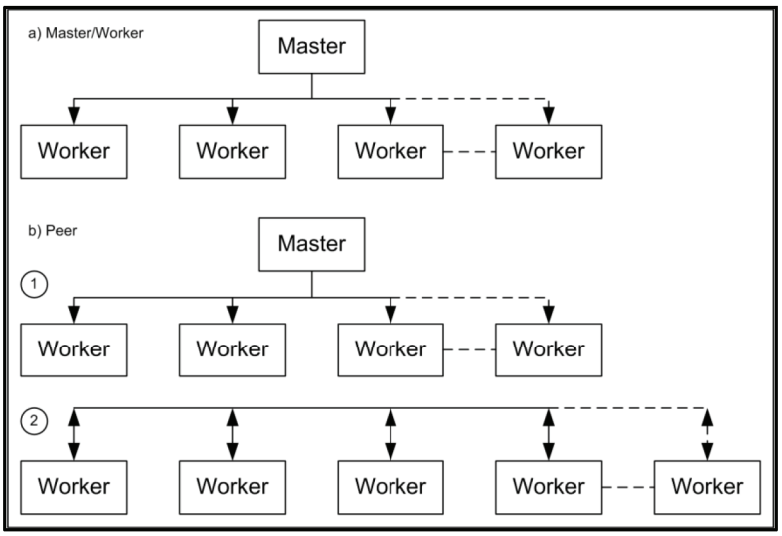

Abbildung 3-10: Master/Worker-Muster

3.2.2.4 Chain of Responsibility

Eine Zuständigkeitskette (engl. chain of responsibility) ist ein Entwurfsmuster aus der Kategorie der objektorientierten Verhaltensmuster (engl. behavior patterns). Bei dieser Art von Mustern sind mehrere Objekte für die Lösung/Bearbeitung einer Aufgabe zuständig. Die Objekte, die für die Bearbeitung zuständig sind, werden dafür hintereinander aufgereiht wie die Glieder einer Kette. Die anfallende Aufgabe wird entlang dieser Kette weitergereicht, bis ein Objekt zuständig ist und die Bearbeitung übernimmt. Das Prinzip ist vergleichbar mit einem Produktionsfließband in einer Autofabrik. Zunächst wird von einem Bearbeiter die Karosserie zusammengesetzt, danach von weiteren der Motor und die Räder montiert und so weiter. Am Ende dieser Kette rollt dann ein fertiges Auto vom Band. Das Ziel einer Zuständigkeitskette ist die Verringerung beziehungsweise die komplette Auflösung der Kopplung zwischen dem Auslöser einer Aufgabe und dem Bearbeiter der Aufgabe. Bei dem Beispiel der Autofabrik würde das bedeuten, der Käufer des Autos erteilt die Aufgabe zum Bauen. Es interessiert den Käufer dabei nicht, wie das Auto gebaut wird oder von wem noch in welcher Reihenfolge die einzelnen Schritte ablaufen. Den Käufer interessiert lediglich wieder das Endresultat, sein fertiges Auto.

Bei einer Zuständigkeitskette sind ganz grob betrachtet immer drei Teilnehmer beteiligt. Der sogenannte Bearbeiter stellt eine übergeordnete Struktur dar (Interface). Er definiert Schnittstellen für die Bearbeitung von Aufgaben. Des Weiteren kann der Bearbeiter auch optional schon das Nachfolgerobjekt festlegen. Der konkrete Bearbeiter bezieht sich auf die von Bearbeiter definierten Schnittstellen für die Bearbeitung einer Aufgabe. Dieses Objekt ist also für einen ganz speziellen Bearbeitungsteil zuständig. Jeder konkrete Bearbeiter hat Kenntnis über den nach ihm folgenden konkreten Bearbeiter, jedoch nicht über seinen Vorgänger. Wenn die Aufgabe einen konkreten Bearbeiter erreicht, entscheidet dieser, ob er für die Bearbeitung zuständig ist. Falls das der Fall ist, wird die Aufgabe bearbeitet, und falls nicht leitet der konkrete Bearbeiter die Aufgabe an seinen Nachfolger weiter. Bei dem Beispiel der Autofabrik ist das etwas anders. Dort spielt weniger die Prüfung der Zuständigkeit eine Rolle, als vielmehr die Reihenfolge der Bearbeitung. Denn in dem Beispiel wird die Aufgabe von mehreren Objekten bearbeitet und das Ergebnis an den Nachfolger weitergeleitet. Ein für Objekt/konkreter Bearbeiter wäre die Lösung der Aufgabe nicht möglich. Der letzte Teilnehmer der Kette ist der Klient. Er löst die Aufgabe aus, hat aber keine Kenntnis über die

Bearbeitung. Deutlich wird das noch einmal in der Abbildung 3-11[24], dort ist die Struktur einer Zuständigkeitskette mit ihren Beteiligten dargestellt.

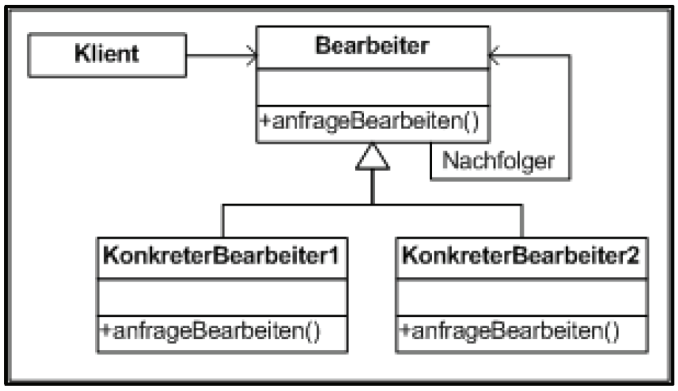

Abbildung 3-11: Struktur einer Zuständigkeitskette

Eine Entscheidung für das Entwurfsmuster steht immer in Verbindung mit bestimmten Kriterien. Erstens, wenn eine Aufgabe von mehr als einem Objekt bearbeitet werden kann und das Objekt, dass die Bearbeitung vornimmt, nicht bekannt ist. Das Objekt für die Bearbeitung wird zur Laufzeit der Anwendung bestimmt. Zweitens, wenn eine Aufgabe an eins von mehreren Objekten gestickt werden soll, ohne den Empfänger explizit anzugeben. Drittens, wenn die Objekte, die die Aufgabe bearbeiten zur Laufzeit der Anwendung dynamisch festgelegt werden sollen. (Abschnitt nach (12 S. 745))

Die Verwendung der Zuständigkeitskette bringt sowohl Vorteile als auch Nachteile mit sich. Zu den Vorteilen zählt Unteranderem die reduzierte Kopplung vom Sender und Empfänger einer Aufgabe. Durch die reduzierte Kopplung muss die Aufgabe nicht wissen, von welchem Objekt sie bearbeitet wird. Sie weiß lediglich das eine Bearbeitung vorgenommen wird. Die lose Kopplung sorgt auch dafür, dass die Struktur der gesamten Kette nicht in jedem Objekt bekannt sein muss. Einem Objekt genügt die Information über sein nachfolgendes Objekt. Somit vereinfachen Zuständigkeitsketten die Objektbeziehungen, die Objekte einer Kette müssen nicht mehr alle Referenzen auf mögliche Empfängerobjekte verwalten sondern nur noch eine auf den eigenen Nachfolger. Ein weiterer Vorteil ist die starke Flexibilität einer Zuständigkeitskette. Die Zuständigkeit von Objekten bei der Bearbeitung einer Aufgabe kann leicht durch Hinzufügen von Objekten in die Kette oder durch das Anpassen der Struktur verändert werden. Als Nachteil kann die Abarbeitungsgarantie gesehen werden. Eine solche kann es aufgrund der Struktur einer Zuständigkeitskette und der Anonymität der konkreten Bearbeiter (Empfänger nicht explizit festgelegt) nicht geben. Zwei weitere Punkte für die nicht vorhandene Abarbeitungsgarantie sind, dass eine Aufgabe am Ende einer Kette ins leere fällt. Sie bleibt unbearbeitet, weil sich keines der Bearbeitungsobjekte zuständig fühlt. Der zweite Punkt ist, dass eine Aufgabe verloren gehen kann, wenn die Kette fehlerhaft konfiguriert ist. Ein weiterer Nachteil ist das mögliche Vorkommen einer Endlosschleife. Zubeachten ist, dass jeder konkrete Bearbeiter nur einmal vorkommen darf. Sollte ein konkreter Bearbeiter mehrmals vorkommen, bleibt sein Nachfolger jedoch gleich, dann entstehen Kreise und somit befindet sich die Anwendung in einer Endlosschleife. (Abschnitt nach (10 S. 370-371; 18))

[24] Nach (10 S. 369; 12 S. 744)

3.3 C++: Spracherweiterungen & Bibliotheken

In den folgenden Kapiteln werden einige Erweiterungen der Programmiersprache C++ und Bibliotheken vorgestellt, mit deren Unterstützung eine Parallelisierung von Software vorgenommen werden kann. Dabei wird sowohl auf die Entwicklung von Software, als auch auf Systeme mit gemeinsamem und verteiltem Speicher eingegangen.

3.3.1 MPI

Das Message Passing Interface beschreibt einen Standard zur Kommunikation zwischen Computersystemen. Daten und Informationen werden dabei über Nachrichten ausgetauscht. Der MPI-Standard legt eine Programmierschnittstelle fest, in dem er Operationen und ihre Semantik bereitstellt. Es handelt sich jedoch nicht um eine konkrete Implementation oder ein Protokoll.

Der Standard wurde 1994 veröffentlicht und stellte Möglichkeiten für Punkt-zu-Punkt, globale und Gruppenkommunikation zur Verfügung. Bei der ersten Veröffentlichung bestand lediglich eine Programmiersprachenanbindung an C und Fortran77. Mit der, im Jahr 2003, veröffentlichten und erneuerten Version des MPI-Standards folgten weitere Funktionen wie parallele Daten Ein- und Ausgabe, eine dynamische Prozessverwaltung und die Möglichkeit des Zugriffs auf den Speicher anderer Prozesse. Als eine weitere Erneuerung gab es eine Anbindung an die Sprachen C++ und Fortran90. Der Standard ist weiterhin in Bearbeitung, momentan wird an der neusten Version MPI-3 durch das MPI-Forum gearbeitet.

Wie schon angedeutet verwendet das Message Passing Interface einen Mechanismus der es erlaubt Programme auch auf Systemen mit einem verteilten Speicher parallel auszuführen. Diese Systeme werden als Cluster bezeichnet, siehe Kapitel 3.1.2. Die Nachrichten die MPI verwendet um in einem Cluster zu kommunizieren sollten dabei über schnelle Netzwerke verteilt werden. Für den Entwickler ist es nicht notwendig zu wissen, welches Netz für den Transport genutzt wird (z. B.: Myrinet, QsNet, Infiniband oder LANs mit Gigabit Geschwindigkeiten). Der Entwickler sollte lediglich wissen, dass der Geschwindigkeitsgewinn der Parallelisierung abhängig ist, von der Größe der zu übertragenden Nachrichten und der Übertragungsgeschwindigkeit. Dadurch, dass es sich beim MPI-Standard um eine plattformübergreifende Schnittstelle handelt, ist es möglich besonders portable Anwendungen zu erstellen, die sich auch gut skalieren lassen. In Bezug auf die in Kapitel 3.2.2 vorgestellten parallelen Entwurfsmuster bleibt anzumerken, dass MPI das Master/Worker-Muster[25] unterstützt. Ebenso ist eine weitere Parallelisierung bei der Kombination von MPI und OpenMP denkbar (19 S. 436-449). Voraussetzung sind Mehr-Prozessor-Systeme im MPI-Cluster.

Im Gegensatz zu Multi-Threading und OpenMP benötigt MPI jedoch etwas mehr Entwicklungsaufwand. Werden jedoch Anwendungen erstellt, die viel rechnen müssen (z. B.: Simulationen oder Audio/Video-Codierungen) sollte zu MPI gegriffen werden, sofern die notwendige Hardware zur Verfügung steht. Ein weiterer Nachteil von MPI ist der hohe Aufwand der Installation und Konfiguration, der sicherlich relativ betrachtet werden kann. MPI muss auf jedem Rechner des Clusters separat installiert und teilweise auch konfiguriert werden. Eine genaue Installationsanleitung soll an dieser Stelle nicht gegeben werden, sondern lediglich ein Verweis auf die Dokumentationen des MPI-Forums[26] und der freien Implementierung von MPICH2.[27]

Eine Implementation der Lupos3D Algorithmen für MPI kam aufgrund der schon angeführten Nachteile nicht infrage. Zum einen ist die für unerfahrene Nutzer schwierige Installation ein Problem. Lupos3D müsste seine Kunden schulen bzw. jederzeit für Supportanfragen zur Verfügung stehen.

[25] Siehe z. B.: **H. Bauke und S. Mertens**: Cluster Computing; Berlin : Springer, 2006. ISBN: 3-540-42299-4
[26] Siehe http://www.mpi-forum.org
[27] Siehe http://www.mcs.anl.gov/research/projects/mpich2

Zum anderen müssen für die Bearbeitung der Daten große Datenmenge über ein Netzwerk verteilt werden das bedeutet, dass der Kunde über ein Hochgeschwindigkeitsnetz verfügen muss. Das kann jedoch nicht gewährleistet werden. Ebenso ist bei der Menge der zu übertragenden Daten ein Geschwindigkeitsgewinn wohl eher nicht zu erwarten. Aus diesen Gründen hielt Lupos3D eine solche Implementierung nicht für sinnvoll. (Kapitel nach (16 S. 64-72; 20; 8 S. 265-275))

3.3.2 Thread-Bibliotheken

Ein Thread ist ein Ausführungsstrang eines Programmes. Er ist Teil eines Prozesses und für die Bearbeitung eines Teils von Code verantwortlich. In der Vergangenheit existierte nur ein Thread pro Prozess das bedeutet, das Programme nur von einem Ausführungsstrang bearbeitet wurden. Sie waren also komplett sequenziell. Seit ein paar Jahren stehen Entwicklern nun jedoch die nötigen Werkzeuge und auch die entsprechende Hardware zur Verfügung, um mehrere Threads in einem Prozess zu erstellen. Somit ist es nun möglich, Teile eines Codes parallel ausführen zu lassen und dadurch schnellere Anwendungen zu entwickeln.

Die Threads können in zwei Arten unterteilt werden, die Kernel- und die User-Threads. Die Kernel-Threads sind abhängig von der Unterstützung des Betriebssystems. Ab dem Jahr 1995 entwickelte Betriebssysteme sollten sowohl Kernel- als auch User-Threads unterstützen. Ein Kernel-Thread ist recht ähnlich zu einem Prozess, er kann genauso mehrere Zustände einnehmen wie zum Beispiel „rechnend“, „rechenbereit“ und „blockierend“. Ein Kernel-Thread ist ein sequenzieller Ausführungsstrang eines Prozesses, in einem Prozess können mehrere Kernel-Threads existieren. Diese Threads teilen sich die Ressourcen die dem Prozess zugeteilt sind zu dem Sie gehören. Die Steuerung der Kernel-Threads übernimmt das Betriebssystem. Es entscheidet, wann zwischen den Kernel-Threads umgeschaltet wird. Die User-Threads hingegen sind eine Implementierung, die dem Betriebssystem unbekannt ist. Bei dieser Art muss der Entwickler die Verwaltung der Threads programmieren, also das wechseln zwischen den Threads und die Zeitplanung (engl. „Scheduling“). (Abschnitt nach (21; 22))

In den folgenden Kapiteln wird nun zunächst der Hintergrund der Thread Programmierung, die Mehrkern-Prozessoren, etwas näher erläutert. Danach folgen Kapitel, die sich mit zwei typischen Arten der Thread Programmierung befassen.

3.3.2.1 Windows-Thread

Leider existiert in der Programmiersprache C/C++ keine native Unterstützung von Parallelität, deshalb benutzten Programmierer Bibliotheken um dennoch Parallelität erzeugen zu können. (23) Eine dieser Bibliotheken ist MFC[28], dabei handelt es sich um eine Sammlung objektorientierter Klassenbibliotheken. Diese Sammlung wurde von Microsoft entwickelt für die Programmierung von Anwendungen mit einer grafischen Benutzeroberfläche für Windows unter C++. MFC bildet eine Schnittstelle zu den nicht objektorientierten Funktionen des Betriebssystems. Mit der Bibliothek wird auch gleichzeitig eine Unterstützung für die Thread Programmierung mitgeliefert. Ein kleiner Nachteil beim Entwicklen von MFC-Programmen mit Microsofts Visual Studio Express ist, das die Bibliothek für diese Entwicklungsumgebung nicht mitgeliefert wird. MFC ist lediglich bei den vollständigen Visual Studio Versionen enthalten(24).

Es wird eine Klasse mit dem Namen „CWinThread“ eingeführt, über Objekten dieser Klasse lassen sich Thread-Funktionen aufrufen. Thread-Funktionen sind zum Beispiel die Definition eines Thread über „AfxBeginThread()“, das Synchronisieren mit Semaphoren oder Mutex-Abschnitten sowie das Warten auf die Beendigung aller gestarteter Threads.

[28] MFC – **M**icrosoft **F**oundation **C**lasses

Für den erfolgreichen Start eines Thread muss vom Entwickler zunächst eine Funktion geschrieben werden, die der Thread ausführen soll. Der Funktion „AfxBeginThread()“ wird die geschriebene Funktion für den Thread und weitere Parameter übergeben. Weitere Parameter sind zum Beispiel Übergabewerte an die Funktion des Threads oder auch die Festlegung der Thread-Priorität (Echtzeit, Normal, unter Normal usw.). Hat der gestartete Thread seine Arbeit beendet, kann dies über entsprechende Funktionen festgestellt werden und die Ergebnisse, sofern es welche gibt, zusammengefasst werden. Der Name „Afx“ steht dabei für „Application Framework Extensions“ und ist der frühere Name der MFC.

Für die Verwendung von Windows-Thread bei der Entwicklung von parallelen Programmen für die beschriebenen Algorithmen aus den Kapiteln 2.2 und 2.3 wurde sich deshalb entschieden, weil das von Lupos3D entwickelte Programm speziell für Windowsnutzer entwickelt wurde und auch schon die MFC-Bibliothek verwendete.

Ein Nachteil der MFC-Bibliothek ist jedoch, dass sie nicht Linux kompatibel ist, eine Portierung des Lupos3D Programmes nach Linux ist jedoch auch nicht vorgesehen.

Neben der Verwendung von MFC können unter Windows auch noch Threads zur Laufzeit erstellt werden. Dazu werden die Funktionen „_beginthread“ und „_beginthreadex“ genutzt (25). Diese Funktionen stammen aus der Klassendefinition der Headerdatei „process.h“. Im Gegensatz zu den Thread-Funktionen der MFC-Bibliothek sind die so erzeugten Threads jedoch nicht „Thread Safe“.

Der Begriff „Thread Safe“ steht, in der Softwareentwicklung, für die Sicherheit beim Umgang mit Programmteilen die gleichzeitig ablaufen können (Thread). Es ist oft der Fall, dass die Threads auf gemeinsame Daten oder Ressourcen zugreifen müssen. Somit greifen die Threads auch auf den gleichen Bereich im Speicher zu. Es ist möglich, dass die Threads sich dabei behindern und so ein Chaos im Speicher entsteht. Beispielsweise können zwei Threads gleichzeitig einen Datensatz aus dem Speicher lesen, ohne dass Probleme entstehen. Soll dieser Datensatz aber nun von beiden Threads verändert werden, durch eine Berechnung zum Beispiel, entstehen Probleme. Nach der Fertigstellung der Berechnung versuchen beide Threads ihr Ergebnis zurück in den Speicher zuschreiben. Dabei geht allerdings das Ergebnis des ersten Threads verloren, da der zweite Thread dieses mit seinem Ergebnis überschreibt. Es wird eine Synchronisation zwischen den Threads benötigt, die den Zugriff kontrolliert. Der Ausdruck „Thread Safe“ besagt somit, dass Daten oder Ressourcen von verschieden parallelen Programmteilen genutzt werden können, ohne sich gegenseitig zu behindern.

3.3.2.2 PThreads

Bei den POSIX Thread handelt es sich wie auch bei den Windows-Threads aus dem Kapitel 3.3.2.1 um eine Bibliothek die Entwickler benutzen, um parallele Anwendungen für C++ zu erzeugen. Die PThreads bilden genauso wie die Windows-Threads eine Schnittstelle zwischen der Anwendung und dem Betriebssystem. Zunächst wurden PThreads nur von Linux-Distributionen unterstützt, mittlerweile sind sie aber auch für Windows erhältlich (PThreads-Win32). Die Verteilung der Threads auf die Prozessoren bzw. Kerne und die Aufgabenplanung übernimmt dabei das Betriebssystem. Die POSIX-Threads sind unter der LGPL[29] veröffentlicht und damit ein OpenSource-Produkt. Die Bibliothek darf für beliebige Zwecke eingesetzt, vervielfältigt, verändert und weitergegeben werden.

In der PThreads Schnittstelle sind mehr als 60 Funktionen implementiert, die eine Parallelisierung ermöglichen. Um die PThreads in einer Anwendung nutzen zu können, muss der Entwickler die

[29] LGPL – **L**esser **G**eneral **P**ublic **L**icense

entsprechende Headerdatei der Bibliothek einbinden. Alle Funktionen, die in der Schnittstelle verfügbar sind, sind mit dem Präfix „pthread_“ gekennzeichnet.

Bei der Anwendung ähneln die PThreads den Windows-Thread, es muss ebenso zunächst eine Funktion definiert werden. Diese Funktion enthält alle Anweisungen, die ein Thread ausführen soll. Für kritische Sektionen stehen die Funktionalitäten der Mutex und Semaphoren zur Verfügung und auch für das Beenden und Warten auf Threads existieren entsprechenden Funktionen.

Bei der Parallelisierung der Algorithmen kamen die PThreads zum Einsatz, um einen Vergleich mit den Windows-Threads anstellen zu können. Gegebenenfalls ergeben sich Verbesserungen der Laufzeit oder der Speichernutzung. (Kapitel nach (23))

3.3.3 Berechnungen auf Grafikkarten

Der Ansatz mit Grafikkarten mathematische Berechnungen durchzuführen ist nicht neu, schon in den 1990er Jahren wurde primitive Berechnungen durchgeführt. Richtig aufmerksam wurde die Industrie und Forschung aber erst im Jahr 2003. Als die Entwicklung der Prozessoren (CPU) an die physikalische Machbarkeitsgrenze stieß, jedoch die Leistung von Grafikkarten sich immer weiter erhöhte. Der Grund, warum GPUs im Gegensatz zu CPUs immer Leistungsfähiger wurden, ist in Kapitel 3.1.3 schon detaillierter beschrieben. In den folgenden Abschnitten soll nun die Entwicklung der Berechnung von Aufgaben auf Grafikkarten betrachtet werden. Das Augenmerk liegt dabei auf CUDA[30] einer Technik, die vom Grafikkartenhersteller Nvidia entwickelt wurde.

Ein großes Problem, dass am Anfang der Verknüpfung von CPU und GPU stand, war die Terminologie. Die Grafikprogrammierer und Parallelprogrammierer verwendeten unterschiedliche Begriffe für gleiche Technologien. Beispielsweise ist ein Stream für Parallelprogrammierer ein Fluss von Daten gleichen Typs, während das Gleiche bei Grafikprogrammierern Textur heißt. Die Terminologie war nicht kompatibel. Bei der Vereinheitlichung der Terminologie der unterschiedlichen Technologien half die BrookGPU. Bei Brook handelt es sich um eine Erweiterung der Programmiersprache C, die von der Stanforduniversität entwickelt wurde. Die GPU sollte sich dank Brook wie ein Coprozessor für parallele Berechnungen verhalten. Dafür bestand Brook aus zwei Komponenten. Zum einen einem Compiler, der aus einer Quelldatei im Brook-Format (.br) einen Standard C++-Code generiert und zum anderen einer Brook Run-Time-Library mit deren Hilfe unterschiedliche Run-Time-Engines (z. B.: DirectX) hinzugelinkt werden konnten. Brook erwies sich als sehr erfolgreich und somit wurde nun auch das Interesse von Grafikartenherstellern, wie Nvidia, an der GPGPU[31] geweckt.

Das Entwicklungsteam von Nvidia erkannte schnell das wachsende Interesse des Marktes und beschloss eine globale Strategie, die direkt auf die Bedürfnisse des Marktes abgestimmt war. Nvidia entwickelte ein komplettes Paket aus Hardware und Software. Die von Nvidia CUDA getaufte Technik besteht aus drei Softwareschichten. Zwei davon sind APIs[32], die CUDA-Runtime-API und die CUDA-Driver-API. Die Abbildung 3-12 zeigt die Hierarchie der Softwareschichten. Für einen Entwickler wichtig zu wissen ist, dass er sich für eine API entscheiden muss. Eine Vermischung von Anweisungen verschiedener APIs ist nicht vorgesehen. Die Verwendung einer CUDA-API schließt die Verwendung der Anderen somit aus. Beispielsweise wird bei einem Aufruf der CUDA-Runtime-API die Funktion in einfache Grundbefehle zerlegt, die von der CUDA-Driver-API verwaltet werden. Eine Verwendung der CUDA-Driver-API hat den Vorteil gegenüber der CUDA-Runtime-API, dass sie noch komplexer ist und

[30] CUDA – **C**ompute **U**nified **D**evice **A**rchitecture
[31] GPGPU – **G**eneral **P**urpose Computation on **G**raphics **P**rocessing **U**nit: Bezeichnet die Verwendung von GPUs für Berechnungen, die sich nicht auf Grafik beziehen.
[32] API – Application Programming Interface

der Entwickler somit noch mehr Kontrolle erhält. Die dritte Softwareschicht besteht wiederum aus zwei Bibliotheken, der CUBLAS und der CUFFT. Die CUFFT-Bibliothek ermöglicht die Berechnungen von Fouriertransformationen und die CULBAS- Bibliothek stellt einen Basissatz für lineare Algebra-Berechnungen auf der GPU zur Verfügung.

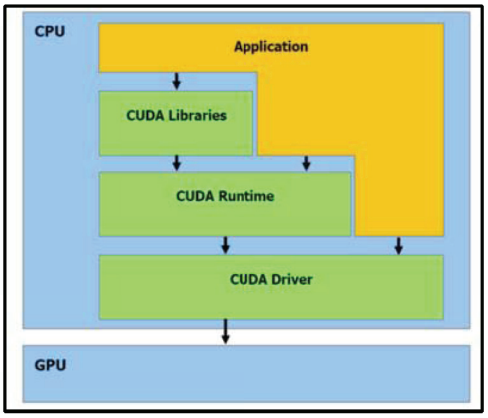

Abbildung 3-12: CUDA zugriff auf die GPU

Eine Implementierung der Algorithmen aus den Kapiteln 2.2 und 2.3 wäre mit Sicherheit ein schönes Experiment gewesen, dennoch passt es nicht in das Konzept der Vermarktung eines kommerziellen Softwareproduktes. Der Kunde müsste über die entsprechende Hardware verfügen, um die Parallelität des Programmes zu nutzen. Gleichzeitig muss an dieser Stelle auch angemerkt werden, dass Grafikkarten mit sehr guten Leistungen in dem Bereich relativ teuer sind. Lupos3D hielt die Möglichkeit der Berechnung mit Grafikkarten zwar für einen guten Ansatz, war aber für eine Implementierung, die eine breite Masse an Nutzern anspricht, eher zu begeistern. (Kapitel nach (9))

3.3.4 Threading Building Blocks

Im Jahr 2006 stellte Intel eine „Template"-Bibliothek für die Programmiersprache C++ vor und nannte sie kurz TBB. (26) Diese Bibliothek sollte es Programmierern erlauben, maximale Leistungen aus Mehrkern-Rechnersystem herauszuholen. Aufgabenstellungen, die wegen ihrer Struktur parallelisierbar schienen, sollten leicht und mit wenig Aufwand parallelisierbar sein. TBB besteht dazu aus Datenstrukturen und Algorithmen, mit denen Programmierer Probleme umgehen können, die zuvor bei der Verwendung von nativen Threading Frameworks wie POSIX Thread und Windows Threads aufgetreten sind. Bei diesen Paketen lag es in der Verantwortung des Programmierers entsprechende Threads zu erstellen, zu synchronisieren und zu beenden. Die TBB übernehmen diese Operationen automatisch. Wenn die Bibliothek beispielsweise feststellt, dass ein Prozessor seine Aufgaben schon bearbeitet hat, während andere Prozessoren noch arbeiten bzw. mehr als genug Arbeit zur Verfügung steht. Dann verteilt TBB die restliche Arbeit neu, sodass das System besser ausgelastet ist.

Die Einsatzmöglichkeiten von TBB sind sehr vielseitig, neben der Beschleunigung von komplexen mathematischen Berechnungen wird die Bibliothek auch von der Computerspieleindustrie eingesetzt, um die Performance zu erhöhen. (27) TBB liefert eine Reihe von Algorithmen und Funktion, um Problemstellungen schnell zu parallelisieren. Dazu gehören Unteranderem eine parallele „For"-Schleife und eine parallele Pipeline.

Die Programmierung einer parallelen „For"-Schleife orientiert sich an der Funktion „parallel_for()" aus der Standardbibliothek von TBB. In diese Funktion müssen lediglich die Anzahl der Durchläufe und die Aufgabe, als Funktion, hineingereicht werden. (28) Bei der parallelen Pipeline gestaltet sich die Erstellung etwas aufwendiger. Es ist notwendig neue Klassen für einzelne Aufgaben der Pipeline

zu erstellen. (28) Eine detailliertere Beschreibung einer parallelen Pipeline Implementation ist in Kapitel 5.4 nachlesbar.

3.3.5 Weitere Möglichkeiten

Neben den in den vorherigen Kapiteln beschriebenen praktischen Möglichkeiten zur Umsetzung einer parallelen Anwendung sollen an dieser Stelle noch zwei Varianten ergänzend erwähnt werden.

Zum einen sind das Compiler, die automatisch Parallelisierungen vornehmen, dazu gehören Compiler von Intel, PGI und Sun. Allerdings existieren für diese Compiler nur kommerzielle Lizenzen oder sie sind nicht Windows kompatibel (Voraussetzung für Funktionen[33] von Lupos3D). Vom Prinzip her versuchen diese automatisierten Compiler (nach setzen einer Option, bei Intel für Windows /Qparallel) Schleifen zu finden, die parallelisierbar sind. Die Schleifen müssen gewisse Anforderungen erfüllen, zum Beispiel eignen sich Grafikroutinen, bei denen jedes Pixel unabhängig berechnet wird. Aber auch Schleifen, die nur ein Ergebnis aufsummieren oder –multiplizieren. Zu dem ist die Parallelisierung abhängig von der Anzahl der Schleifendurchläufe. Bei dem Intel Compiler liegt die Mindestgröße standardmäßig bei 75, diese lässt sich jedoch auch ändern. (Abschnitt nach (16 S. 47f))

Die zweite ergänzende Möglichkeit ist der OpenMP-Standard, das Open steht dabei für die Offenheit der Spezifikation und das MP für Multi Processing. OpenMP funktioniert mit gängigen Programmiersprachen wie C/C++ und Fortran auch die gängigen Compiler (z. B. Microsoft Visual Studio (C++), Intel C++ Compiler und GNU Compiler Collection (GCC)) kommen mit OpenMP-Anweisungen zurecht. OpenMP eignet sich jedoch nur für „shared memory“ Systeme, das hat den Vorteil, dass Programmierer sich nicht um die Verteilung von Daten oder die Synchronisation der Threads kümmern müssen. Dadurch wird der für die Programmierung benötigte Aufwand geringer, gleichzeitig nimmt aber auch der Grad der Parallelisierbarkeit und der Skalierbarkeitsgrad ab. Der Code der Anwendung ist unabhängig von der OpenMP-Fähigkeit des Compilers, da der Code nicht verändert sondern lediglich ergänzt wird. Sollte ein Compiler zum Kompilieren einer Anwendung benutzt werden, der OpenMP nicht unterstützt, stellt das kein Problem dar. Die OpenMP-Anweisungen werden in dem Fall als Kommentar betrachtet oder einfach ignoriert. Dadurch, dass OpenMP ein Hersteller übergreifender Standard ist, sollte die OpenMP-Anweisungen von jedem OpenMP fähigen Compiler unterstützt werden. (Abschnitt nach (16 S. 50-55))

[33] z. B.: Lesen und Schreiben des eigenen Datenformats, DLL Zugriffe

4 Vorbetrachtungen

Um festzustellen, ob eine weiterführende Bearbeitung der Algorithmen überhaupt lohnenswert ist, sollten zuvor kritische Bereiche genauer betrachtet werden. Der kritische Bereich bei den in Kapitel 2.2 und 2.3 vorgestellten Algorithmen ist das Verhältnis von sequenziellen und parallelisierbaren Operationen. Sequenzielle Operationen sind in beiden Beispielen Festplattenoperationen, Lesen und Schreiben von Daten sowie Initialisierungsoperationen.

Eine Analyse des Verhältnisses zwischen sequenziellen und parallelisierbaren Operationen kann nicht ohne eine prototypische Implementierung vorgenommen werden. Für die Analyse wurde pro Algorithmus jeweils ein Programm entwickelt, welches Daten aus einer Eingangsdatendatei, mit den Funktionen von Lupos3D, liest, schreibt und die Berechnung durchführt. Die Programme sind dabei komplett sequenziell.

Die Organisation der Zeitmessungen in den Prototypprogrammen zeigt die Abbildung 4-1.

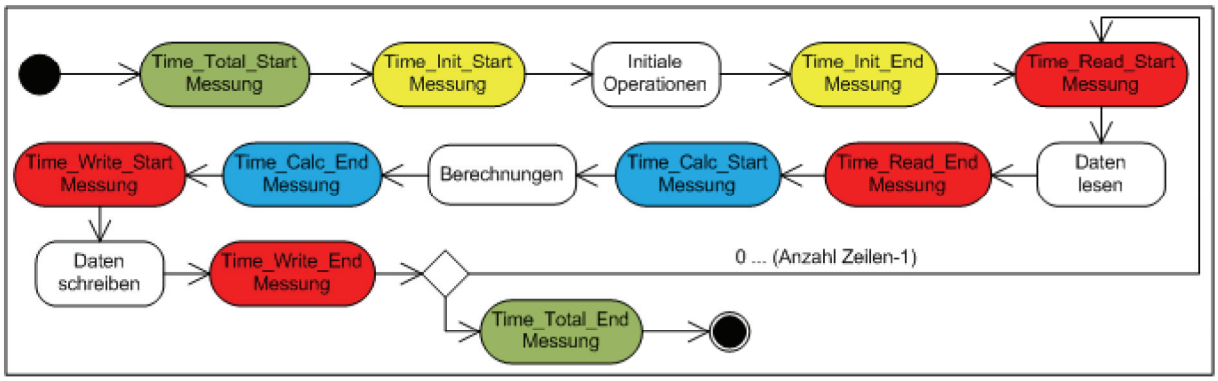

Abbildung 4-1: Grobe Zeitorganisation

Die folgenden Farbangaben beziehen sich auf die Abbildung 4-1. Zum Start und zum Ende der Programme wird jeweils eine Zeitmessung (grün) vorgenommen und die Differenz der Werte berechnet. Aus der Differenz dieser Messungen ergibt sich die Gesamtlaufzeit des jeweiligen Programmes. Initiale Operationen, die möglicherweise in den Programmen vorkommen, werden ebenfalls mit zwei Zeitmessungen (gelb) bestimmt. Die Differenz der Werte (gelb) ergibt somit die benötigte Zeit. Vor dem Beginn und am Ende der Lese- und Schreiboperationen (rot) und der Berechnungen (blau) wird zunächst jeweils eine Zeitmessung durchgeführt. Gelesen, geschrieben und berechnet wird in einer „For“-Schleife. Mit jedem Durchlauf der „For“-Schleife wird eine Zeile der Eingangsdaten komplett verarbeitet. Die in der „For“-Schleife ermittelten Zeitdifferenzen werden über die Gesamtdurchläufe der „For“-Schleife aufsummiert. Zur Kontrolle, der gemessen Zeiten dient, die Summe der Einzelzeiten nach der Formel 5:

Formel 5: Bestimmung der Gesamtlaufzeit

$$Gesamtlaufzeit := Zeit_{gelb} + \left(Zeit_{rot_{read}} + Zeit_{rot_{write}}\right) + Zeit_{blau}$$

Die nach der Formel 5 errechnete Zeit muss mit der gemessenen Gesamtlaufzeit übereinstimmen, beziehungsweise darf sie aufgrund von Rundungen nur minimal abweichen. Sollte das nicht der Fall sein, liegt ein Fehler in der Organisation der Zeitmessung vor.

Nach der Kontrolle der vorgenommenen Zeitmessungen kann die Analyse des Verhältnisses von sequenziellen und parallelisierbaren Operationen in Bezug auf die Gesamtlaufzeit der jeweiligen Prototypimplementierung vorgenommen werden. Die Zeit für die sequenziellen Operationen setzt sich zusammen aus der Zeit für die Initialisierungen (gelb) und der Gesamtsumme der Lese- und Schreiboperationen (rot), siehe Formel 6.

Formel 6: Zeit sequenzielle Operationen

$$Gesamtzeit\ sequenzielle\ Operationen := Zeit_{gelb} + \left(Zeit_{rot_{read}} + Zeit_{rot_{write}}\right)$$

Die parallelisierbaren Operationen befinden sich nur im Bereich der Berechnungen. Es kann für die Analyse somit direkte die gemessene Zeit (blau) verwendet werden. Das Verhältnis zwischen sequenziellen und parallelisierbaren Operationen wurde von Gene Amdahl untersucht. Nach seiner Theorie wird ein Geschwindigkeitszuwachs eines parallelisierten Programmes durch den Anteil der sequenziellen Operation an der Gesamtlaufzeit bestimmt. Der Anteil der jeweiligen Operationen an der Gesamtlaufzeit wird durch die Formel 7 ermittelt:

Formel 7: Bestimmung des Anteils einer Operation

$$Anteil_{Operation\ an\ Gesamtzeit} := \frac{Zeit_{Operation}}{Zeit_{Gesamt}}$$

Die Gesamtlaufzeit eines Programmes setzt sich aus der Summe der einzelnen Anteile zusammen, so ergibt sich für die Gesamtlaufzeit eines Programmes die Formel 8:

Formel 8: Zusammensetzung der Gesamtlaufzeit

$$Zeit_{Gesamt} := Zeit_{\ sequentielle\ Operationen} + Zeit_{parallele\ Operationen}$$
$$Allgemein: 1 = (1 - P) + P$$

Die theoretische Geschwindigkeitssteigerung eines Programmes bei einer Parallelisierung wird als SpeedUp bezeichnet. Dabei wird das auf einem Ein-Kern Computer ausgeführte Programm mit dem auf einem Multi-Kern Computer ausgeführten Programm ins Verhältnis gesetzt. Der SpeedUp berechnet sich nach der Formel 9:

Formel 9: Theoretischer SpeedUp

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \leq \frac{1}{1 - P}$$

In der Formel 9 ist „S“ der theoretische SpeedUp, „P“ der parallele Anteil des Programms, „N“ die Anzahl der Prozessoren, „(1-P)“ der sequenzielle Anteil des Programmes und „(P/N)“ der beschleunigte parallele Anteil. Ein Beispiel wäre, wenn ein Programm 20 Stunden läuft und davon der sequenzielle Anteil eine Stunde beträgt, so ist der parallele Anteil 19 Stunden. Selbst bei unendlich vielen verwendeten Prozessoren würde das Programm immer eine Laufzeit von mindesten einer Stunde aufweisen. Der theoretische SpeedUp würde also dem Faktor 20 entsprechen. Die Abbildung 4-2 zeigt wie sich nach Amdahls-Gesetz der parallele Anteil eines Programmes und der SpeedUp verhalten, wenn eine unterschiedliche Anzahl von Prozessoren zum Einsatz kommt.

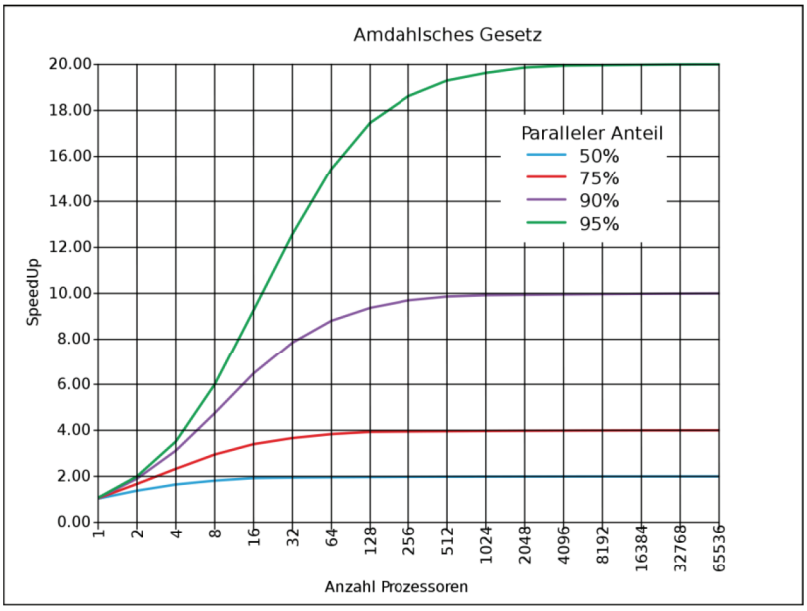

Abbildung 4-2: Amdahlsches Gesetz

Für die Betrachtungen der Anteile von sequenziellen und parallelisierbaren Operationen der zuvor beschriebenen Algorithmen (Kapitel 2.2 und 2.3) werden Zeitmessungen nach dem Schema aus Abbildung 4-1 durchgeführt.

Für den Schnitte-Algorithmus (Kapitel 2.2) konnte nach der Durchführung einer Messreihe ein durchschnittliches Verhältnis ermittelt werden. Die Berechnungen benötigten demnach zwischen 87,09% und 92,57% der gesamten Zeiten. Im Durchschnitt wurden 90% der Zeit zum Berechnen der Ergebnisse verwendet und 10% für Festplattenoperationen. Da sehr wenig Zeit für die Festplattenoperationen benötigt wird, ist die Festplatte zwar ein limitierender Faktor. Aber dieser spielt nur eine untergeordnete Rolle. Nach Amdahls-Gesetz könnte also ein theoretisch maximaler Geschwindigkeitsvorteil um den Faktor 10 erreicht werden. Auch unendliche viele Prozessoren könnten diesen Faktor nicht mehr erhöhen, da er vom sequenziellen Anteil der Ausführungen limitiert wird.

Der zweite Algorithmus (Photo-2-Scan aus Kapitel 2.3) ist ebenfalls mit einer Messreihe hinsichtlich der sequenziellen und parallelisierbaren Anteile des Programmes unter sucht worden. Der parallelisierbare Anteil des Programmes lag zwischen dem Minimalwert von 36,12% und dem Maximalwert von 88,22% der kompletten Laufzeit des Programmes. Die weite Streuung der Messergebnisse des parallelisierbaren Anteils drückt den durchschnittlichen parallelisierbaren Anteil auf 62,17%. Nach Amdahls-Gesetz und der in die Formel 9 eingesetzten Werte ergibt sich somit ein theoretischer SpeedUp um den Faktor 2,64. Beim Photo-2-Scan-Algorithmus zeigt sich umso mehr, dass die Festplatte ein stark limitierender Faktor ist. Werden die Festplattenoperationen jedoch außer Acht gelassen, zeigt sich ein anderes Bild bei diesem Algorithmus. Dann liegen der minimale Wert des Anteils paralleler Operationen bei 84,75% und der Maximalwert bei 95,36%. Der Durchschnitt ergibt einen Wert von 90%. Wird der theoretische SpeedUp mit diesen Werten nach Formel 9 erneut berechnet, ergibt sich ein Geschwindigkeitsvorteil um den Faktor 10. Durch die Herausnahme der Festplattenoperationen aus der Messreihe wird der Algorithmus um ein wichtiges Merkmal vereinfacht. Dieses Vorgehen könnte damit erklärt werden, dass auf die Entwicklung und Verbreitung schneller Festplatten spekuliert wird, wie zum Beispiel Solid State Drive[34] (SSD). Eine Parallelisierung des Photo-2-Scan-Algorithmus scheint auf den ersten Blick keinen reellen Vorteil

[34] Siehe z. B.: Wikipedia: Solid-Sate-Drive; http://de.wikipedia.org/wiki/Solid_State_Disk

zubringen. Die Betrachtung des Algorithmus kann aber mit einem kritischen Auge auf die Entwicklung der Festplattentechnologie weiter fortgesetzt werden.

Neben dem theoretischen SpeedUp existiert eine weitere Variante, der reale SpeedUp, siehe Formel 10. Der reale SpeedUp beschreibt die tatsächliche Beschleunigung eines parallel verarbeiteten Programmes gegenüber einem Referenzprogramm. Der SpeedUp ergibt sich somit aus der benötigten Zeit für die schnellste sequenzielle Variante des Programmes (T(1)) geteilt durch die benötigte Zeit bei der parallelen Verarbeitung mit „p“ Prozessoren (T(p)). Aussagen über den realen SpeedUp können erst nach entsprechenden Messungen der Laufzeiten der beiden verendeten Algorithmen gemacht werden, diese können im Kapitel 6 auf Seite 68 nachgelesen werden.

Formel 10: Realer SpeedUp

$$S(p) = \frac{T(1)}{T(p)}$$

5 Implementierung

5.1 Computerdetails

In den folgenden Kapiteln wird ein wiederkehrender Bezug zu den verwendeten Testcomputern hergestellt. Um die jeweiligen Bezüge eindeutig zu halten, wird eine eindeutige Kennzeichnung eingeführt, die aus der Tabelle 5-2 und Tabelle 5-3 zu entnehmen ist. Diese Tabellen dienen auch der Einordnung der Leistungsfähigkeit der Testcomputer und lassen somit eine Einschätzung der Ergebnisse zu.

Wie aus den Tabellen unten entnehmbar ist, handelt es sich bei beiden Testcomputern um 32Bit Systeme mit einem Microsoft Windows XP Betriebssystem. Beide Computer sind auf dem aktuellsten Stand (Updates)[35]. Beim TC2 entfällt die Angabe von Hersteller und Modell, weil es sich bei diesem Computer im Gegensatz zu TC1 um einen stationären Computer handelt. Dieser ist nicht wie der TC1 von einem einzigen Hersteller konfiguriert worden, sondern besteht aus Komponenten unterschiedlicher Hersteller. Die Angabe der Cache-Größe und des Cache-Typs spielt eine Rolle bei der Effizienz der Programme.

Wenn bei den Testcomputern von Cache gesprochen wird, ist dabei nur der Prozessor-Cache gemeint. Ein Cache ist ein Speicher, z. B. zwischen Prozessor und dem Hauptspeicher. Der Zugriff auf den Zwischenspeicher ist sehr schnell im Vergleich zum Zugriff auf andere Hardwarekomponenten. Genutzt wird der Cache um oft benötigte Daten zwischen zu speichern, damit diese nicht immer wieder neu von einem langsamen Medium geladen werden müssen. Bei Prozessoren kann der Einsatz eines Cache das Flaschenhalsproblem der „von-Neumann-Architektur“[36] verringern. (29) Bei der Ausführung von Programmen kann dadurch im Durchschnitt eine schnellere Ausführungszeit erreicht werden. Ein großer und schneller Cache wird daher immer bevorzugt, jedoch ist es recht schwierig, einen solchen Cache herzustellen. Daher werden in der Praxis häufig mehrere kleine Caches verwendet. Diese reihen sich dann in eine sogenannte Cache-Hierarchie ein, dabei werden verschiedene Level benutzt. Je hardwarenäher ein Cache am Prozessor ist, desto schneller ist die Zugriffszeit und desto kleiner ist auch sein Level. Die Abbildung 5-1 zeigt anhand einer Grafik die Cache-Hierarchie. Als Abkürzung für die Level-Stufe wird ein großes „L“ verwendet gefolgt von der Level-Stufe als Nummer. Beispielsweise ist ein L2-Cache weiter von der Prozessorhardware entfernt als ein L1-Cache. Ein Cache des Levels Eins kann unter Umständen den gleichen Takt wie der Prozessor aufweisen, also mehrere Gigahertz. Beim Einsatz von mehreren Caches spielt die Cache-Hierarchie eine wichtige Rolle, denn der Cache mit dem niedrigsten Level (schnellste Zugriffszeit) wird zuerst nach den benötigten Daten durchsucht. Werden diese in dem Cache nicht gefunden (Cache-Miss), wird der Cache in nächsthöheren Level durchsucht. Das geschieht solange, bis die Daten gefunden wurden (Cache-Hit) oder kein Cache mehr vorhanden ist, dann werden die Daten von der langsameren Hardware nachgeladen. Einen kleinen Überblick, über die Zugriffszeiten und Größen der unterschiedlichen Zugriffsmedien bietet die Tabelle 5-1.

[35] Stand: 15.06.2010

[36] Siehe z. B.: **Hartmut Ernst**: Grundkurs Informatik; Aufl. 4; Seite 200-202; ISBN: 978-3-8348-0362-7

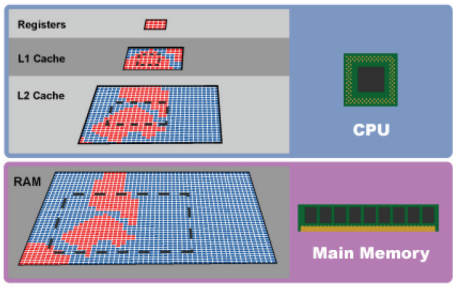

Abbildung 5-1: Cache-Hierarchie

Level	Zugriffszeit	Typische Größe	Technologie	Organisiert von
Register	1-3 ns	1 KB	CMOS	Compiler
Level 1 Cache (auf dem Chip)	2-8 ns	8-128 KB	SRAM	Hardware
Level 2 Cache (nicht auf dem Chip)	5-12 ns	0,5-8 MB	SRAM	Hardware
Hauptspeicher	10-60 ns	64 MB – 1 GB	DRAM	Betriebssystem
Festplatte	3.000.000-10.000.000 ns	20-100 GB	Magnetisch	Betriebssystem

Tabelle 5-1: Zugriffszeit auf den Cache[37]

Kennung:	**TC1**
Hersteller:	LG Electronics
Modell:	R500-C.CB11V
Betriebssystem:	Microsoft Windows XP Home Edition Version 5.1.2600 Service Pack 3 Build 2600
Systemtyp:	X86-basierter PC
Prozessor:	x86 Family 6 Model 15 Stepping 10 GenuineIntel (Intel Core2 Duo T7300 @ 2GHz)
Cache:	1 x 4 MB (L2)
Arbeitsspeicher:	2.048,00 MB
Festplatte:	Fujitsu MHW2160BH
Typ:	SATA
RPM[38]:	5400
Datentransfer Host to Drive:	150 MB/Sek.
Puffergröße:	8 MB

Tabelle 5-2: Details Testcomputer 1 (Notebook)

[37] Vgl. Quelle (35)
[38] **r**evolutions **p**er **m**inute: Gibt die Anzahl der Umdrehungen der Festplatte pro Minute an

Kennung:	**TC2**
Betriebssystem:	Microsoft Windows XP Professional Version 5.1.2600 Service Pack 3 Build 2600
Systemtyp:	X86-basierter PC
Prozessor:	x86 Family 6 Model 23 Stepping 10 GenuineIntel (Intel Core2 Quad Q8200 @ 2,3GHz)
Cache:	2 x 2 MB (L2)
Arbeitsspeicher:	4.096,00 MB
Festplatte:	Western Digital WD3200AAJS
Typ:	SATA
RPM:	7200
Datentransfer Host to Drive:	126 MB/Sek.
Puffergröße:	8 MB

Tabelle 5-3: Details Testcomputer 2

5.2 Laufzeitbestimmung

Die in den folgenden Kapiteln beschriebenen Zeitmessungen und Ergebnisse, basieren auf der Bestimmung von Laufzeitunterschieden. Zur Verdeutlichung der korrekten Implementation der Zeitmessung dient die Abbildung 5-2, sie zeigt, wie eine korrekte Messung im Quellcode zu programmieren ist. Die folgenden Zeilenangaben beziehen sich auf die eben erwähnte Abbildung. Die Basis für die Messungen bildet die Headerdatei „time.h“, die dem Programm hinzugefügt werden muss (Zeile 1). In der Datei enthalten sind Datentypen, Datenstrukturen, Funktionen und Variablen, die den Umgang mit der Zeit vereinfachen. In der Zeile 2 werden Variablen, die die ermittelten Zeiten aufnehmen können, angelegt. Dazu wird der spezielle Datentyp „clock_t“ verwendet, dieser beruht auf dem Standarddatentyp „long“. Der Datentyp „long“ hat eine Größe von vier Byte und kann somit Werte in einem Bereich von -2,147,483,648 bis 2,147,483,647 aufnehmen. Gestartet wird die Laufzeitmessung mit der Funktion „clock()“ und der Zuweisung des Rückgabewertes der Funktion an eine der zuvor definierten Variablen (Zeile 8). Die Funktion „clock()“ ermittelt die verbrauchte CPU-Zeit das heißt, Zeiten von anderen parallel ausgeführten Prozessen fließen nicht in die Messung mit ein. Sind alle Operationen, für die die Laufzeit bestimmt werden sollte, ausgeführt, wird die Zeitmessung mit dem erneuten bestimmen der CPU-Zeit (Funktion „clock()“) beendet. Die Zeile 12 zeigt die Differenzbildung der Start- und Endzeit. Das Ergebnis ist die verbrauchte Zeit in CPU-Ticks. Um die CPU-Ticks in Sekunden umzuwandeln, muss die Zeit durch die Konstante „CLOCKS_PER_SEC“ geteilt werden. Die Auflösung in Sekunden ist jedoch nicht immer ausreichend, darum wird das Ergebnis der Division mit dem Wert 1000 multipliziert, um eine Auflösung in Millisekunden zu erreichen (Zeile 16). Die Multiplikation ist durchaus möglich, da der Datentyp „clock_t“, wie schon erwähnt, auf dem Datentyp „long“ basiert.

Sollte ein Programm mehrere Zeitmessungen enthalten, die sich auch überschneiden, so bietet es sich an, die Zeile 16 nach der Beendigung aller wichtigen Operationen durchzuführen. Das hat den Vorteil, dass andere Zeitmessungen nicht beeinflusst werden. Falls eine Ausgabe auf den Bildschirm erfolgt, ist es sinnvoll dies auch am Ende des Programmes durchzuführen. Der Grund ist das Ausgaben auf den Bildschirm durchaus einige Millisekunden dauern können, und dass das andere Messungen beeinflusst.

```c
#include <time.h>

int main(int argc, char *argv[]){
    clock_t start=0, end=0;                 // Variablen für die Zeitmessung
    // Weitere Operationen
            ...

    start = clock();                        // Beginn der Zeitmessung

    /* zu messende Operationen */

    end = clock()-start;                    // Ende der Zeitmessung
    // Weitere Operationen
            ...

    end = (end/CLOCKS_PER_SEC)*1000;        // Berechnung der Laufzeit
    // Weitere Operationen
            ...

    return 0;
}
```

Abbildung 5-2: Implementation einer Zeitmessung

Bei der Implementation der Programme dieser Arbeit wurden, am Ende des jeweiligen Programmes, alle ermittelten Zeiten auf einmal auf den Bildschirm ausgegeben. Zur Vereinfachung einer späteren Auswertung wurden die Zeiten nur mit einem Semikolon voneinander getrennt. Eine Dokumentation bei welchem Wert es sich um welche Zeit handelt ist im Quellcode kommentiert. Für die Ermittlung von geeigneten Durchschnittswerten wurde eine Batchdatei geschrieben. Diese Datei führt alle Programme der Reihe nach 51-mal aus und erstellt jeweils eine neue Ergebnisdatei mit allen ermittelten Zeiten. Die Zeiten sind in der Ergebnisdatei zeilenweise gespeichert, das vereinfacht die Auswertung zum Beispiel mit Microsofts Excel. Ebenfalls enthält die Ergebnisdatei in der ersten Zeile die Dokumentation, bei welchem Wert es sich um welche Zeit handelt. Wie schon erwähnt wurden alle Programme 51-mal ausgeführt, wobei das erste Ergebnis bei einer Auswertung nicht berücksichtigt wird. Das hat den Grund, dass beim ersten Aufruf eines Programmes dieses erst in den Arbeitsspeicher geladen wird und das kann die Ergebnisse verfälschen. Dadurch, dass ein Programm immer wiederholt ausgeführt wird, bleibt das Programm im Arbeitsspeicher erhalten. Die Folge ist, dass die Ergebnisse ab dem zweiten Durchlauf repräsentativer für den eigentlichen Zeitbedarf sind als die Ergebnisse des ersten Durchlaufs. Zur Veranschaulichung des ebnen beschrieben Formates dient die Abbildung 5-3. Sie zeigt einen Ausschnitt einer Ergebnisdatei und auch das die erste Ergebniszeile stark von den Anderen abweicht. Ins Auge fällt besonders der dritte Wert, der dadurch die gesamte Laufzeit (erster Wert) erhöht.

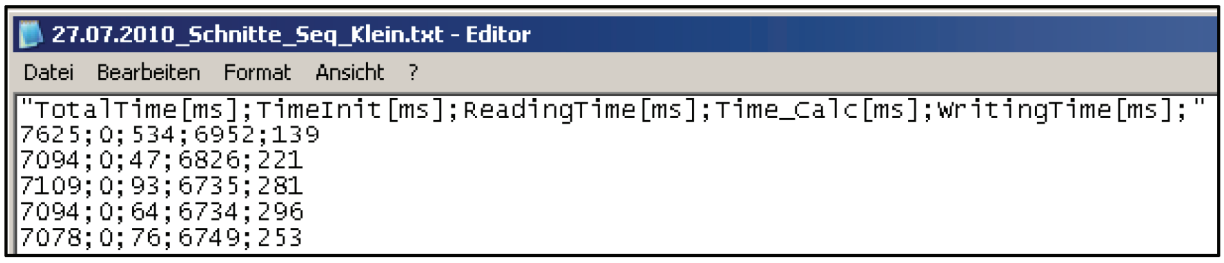

Abbildung 5-3: Format der Ergebnisdateien

5.3 Schnitte-Algorithmus

Der Schnitte-Algorithmus kann am besten mit dem im Kapitel 3.2.2.2 beschriebenen Entwurfsmuster Divide`n`Conquer verglichen werden. Der Algorithmus ist so datenunabhängig, dass die Anzahl der Aufgaben, die verarbeitet werden müssen, direkt auf mehrere Threads verteilt werden kann. Dabei kommt das schon erwähnte "embarrassing parallel" Parallelisierungskonzept zum Tragen.

Der Ausdruck „embarrassing parallel“ (dt.: peinlich parallel) beschreibt die Art, wie ein Algorithmus in Bezug auf seine Parallelisierbarkeit angesehen wird. Im Allgemeinen wird der Ausdruck für sequenzielle Algorithmen oder Aufgaben verwendet, die mit sehr geringem Aufwand in eine parallele Struktur umgewandelt werden können. Der Aufwand ist oft bei Algorithmen sehr gering, wenn diese keine Datenabhängigkeit aufweisen. Das bedeutet, es muss keine oder nur sehr wenig Kommunikation zwischen den parallel durchgeführten Aufgaben stattfinden.
In dem Fall des Schnitte-Algorithmus kann jede Zeile des Datenbereiches, der Laserscandatei, individuell betrachtet und verarbeitet werden. Lediglich das Ergebnis muss mittels einer Kommunikation der ausgeführten Threads zusammengesetzt werden. Die Kommunikation ist der Tatsache geschuldet, dass das Ergebnis des Algorithmus in eine einzige Ergebnisdatei geschrieben wird und in diese Datei darf jeweils nur einer der Threads schreiben. Die Kommunikation zwischen den parallelen Prozessen ist somit auf ein Minimum beschränkt.

Der eigentliche Ablauf des Schnitte-Algorithmus gliedert sich, bei der vorgenommenen Implementation, in drei Bereiche. Von den Bereichen sind zwei sequenziell und lassen sich auch nicht parallelisieren und ein Bereich ist parallelisiert. In der Abbildung 5-4 ist in einem Programmablaufdiagramm die Vorgehensweise der Implementation dargestellt. In dieser Abbildung sind ebenfalls die schon erwähnten Bereiche farblich dargestellt.
In einem Initialbereich (gelb) des Programmes werden Daten geladen und verarbeitet, die für alle gestarteten Threads identisch sind. In dem mit der Farbe Rot gekennzeichnetem Bereich wird die Ergebniszeile der Berechnung in die Ergebnisdatei geschrieben. Der blaue Bereich der Abbildung 5-4 kennzeichnet die parallel ablaufenden Teile des Algorithmus.

Detailliertere Erläuterungen zu den einzelnen Teilaufgaben können im Kapitel 5.3.1 und den entsprechenden Unterkapiteln nachgelesen werden.

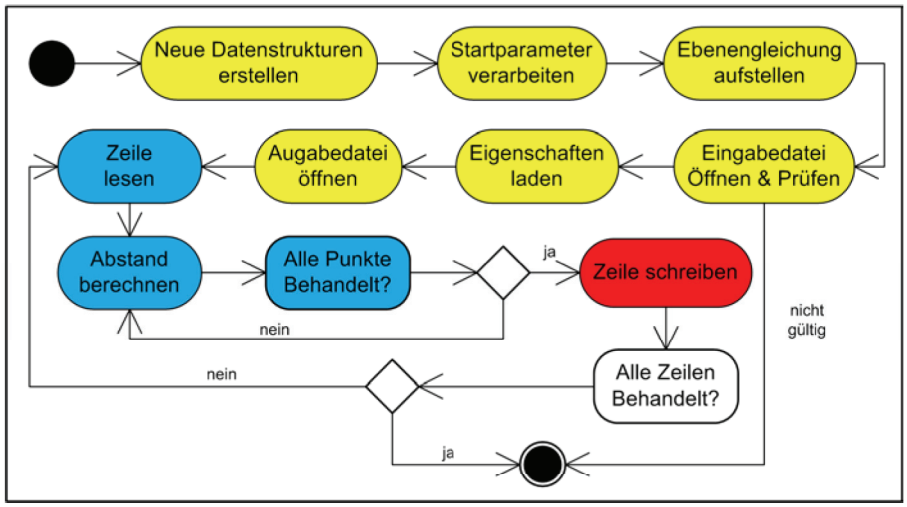

Abbildung 5-4: Schnitte: Ablauf und Arbeitsbereiche

5.3.1 Beschreibung der Teilaufgaben

Die folgenden Kapitel dienen der genaueren Beschreibung der Tätigkeiten der einzelnen Teilaufgaben des Schnitte-Algorithmus und der Klärung einer sequenziellen Implementierung. Auf die Implementierung bei einer Parallelisierung wird in späteren Kapiteln eingegangen.

5.3.1.1 Datenstrukturen anlegen

Das Erstellen von eignen Datenstrukturen soll die Entwicklung erleichtern, denn so besteht die Möglichkeit zusammengehörige Daten eines unterschiedlichen Datentyps, auch zusammen in einem Objekt zu verwalten. Die Programmiersprache C++ bietet dafür die Funktionalität der Typendefinition

an, dabei handelt es sich um eine Compiler-Anweisung. Die Abbildung 5-5 zeigt die Implementation einer neuen Datenstruktur und ihre Verwendung. Als erster Schritt muss der Compiler auf die neue Definition hingewiesen werden, das geschieht in der ersten Zeile durch den Ausdruck „typedef Struct“. In einer Blockanweisung[39], wird die eigentliche Datenstruktur definiert. Dafür ist es lediglich notwendig, den Namen und den Datentyp eines Elements der Struktur festzulegen. Die Initialisierung erfolgt später beim Anlegen der neuen Datenstruktur als Objekt. Nach der Blockanweisung folgt der Name der Datenstruktur, mit dem Objekte der Struktur erstellt werden können (Zeile 3). Die Verwendung der neuen Datenstruktur gestaltet sich genauso wie bei anderen Strukturen. Der Name der neuen Struktur dient als Datentyp für die Erstellung von Objekten. Auf die einzelnen Elemente der so erstellten Objekte kann über den Punkt-Operator (Objekt ist lokal) bzw. über den Pfeil-Operator bei Pointern zugegriffen werden.

```
1  typedef struct {                                          // Anlegen einer neuen Datentypstruktur
2      double p1x, p1y, p1z, p2x, p2y, p2z, p3x, p3y, p3z;   // Struktur besteht aus mehreren double Werten
3  } Ebene;                                                  // Name des neuen Datentyps
4
5  typedef struct {
6      Ebene def_Ebene;                                      // Wiederverwenden des neuen Datentyps
7      double abstand, toleranz;
8      std::fstream * out;
9  } Schnitte;
```

Abbildung 5-5: Definition einer Datenstruktur

5.3.1.2 Ebenengleichung aufstellen

Das Aufstellen einer Ebenengleichung ist notwendig, um den Abstand von Punkten zu dieser Ebene zu berechnen. Bei dem Schnitte-Algorithmus wird eine DLL-Datei von Lupos3D für die Aufstellung der Gleichung genutzt. Wie die DLL verwendet wird, beschreibt das Kapitel 5.3.1.4 genauer. Das Bestimmen der Gleichung könnte auch neu implementiert werden, aber dann lassen sich keine Vergleiche mehr zwischen dem sequenziellen und dem parallelen Schnitt-Algorithmus, wie er bei Lupos3D zum Einsatz kommt, ziehen.

Für eine eigene Implementation der Bestimmung der Ebenengleichung, im dreidimensionalen Raum, ist es lediglich notwendig, die Berechnung der hessischen Normalform zu implementieren. Die hessische Normalform wurde schon im Kapitel 2.2 auf der Seite 3 vorgestellt und besteht im Wesentlichen aus dem Skalarprodukt, der Subtraktion und dem Kreuzprodukt von Vektoren. Diese mathematischen Operationen lassen sich jedoch schnell programmieren, da die Anzahl der Vektoren auf drei beschränkt ist und diese auch nur im dreidimensionalen Raum liegen. Dazu werden die Vektoren zurückgeführt auf ihre einzelnen Elemente, dadurch lässt sich die Implementation weiter vereinfachen. Die Formel 11, Formel 12 und Formel 13 zeigen, wie die Vektorberechnung in die grundlegenden mathematischen Berechnungen umgeformt wird.

Formel 11: Subtraktion von Vektoren[40]

$$\vec{a} - \vec{b} = \begin{pmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{pmatrix}$$

Formel 12: Skalarprodukt zweier Vektoren[41]

$$\vec{a} \cdot \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

[39] Inhalt zwischen geschweiften Klammern „{}“
[40] Vgl. Wikipedia – http://de.wikipedia.org/wiki/Vektor#Addition_sowie_Subtraktion
[41] Vgl. Wikipedia – http://de.wikipedia.org/wiki/Vektor#Skalarprodukt

Formel 13: Kreuzprodukt zweier Vektoren

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

5.3.1.3 Transformation in globales System

Die Transformation in ein globales Koordinatensystem ist notwendig, weil die Punkte, die der Laserscanner aufgezeichnet, als Koordinatenursprung die Position des Scanners besitzen. Der Laserscanner wiederum steht aber nicht im Ursprung des übergeordneten Koordinatensystems. Die Punkte, die aufgezeichnet wurden, müssen also zunächst in ein globales/übergeordnetes System transformiert werden. Der Sinn der Transformation ist, dass der Laserscan nach der Transformation mit anderen Scans, Karten oder Bildern, die sich ebenfalls im globalen Koordinatensystem befinden, zusammengefügt werden kann.

Für die Berechnung der Transformation muss die Orientierung des Koordinatensystems des Laserscanners in Bezug zum globalen Koordinatensystem bekannt sein. Im Laserscan ist dafür eine Orientierungsmatrix gespeichert. Die Matrix enthält die Parameter, die beschreiben, wie Koordinaten im Laserscannerkoordinatensystem verschoben und gedreht werden müssen, damit sie an der korrekten Stelle im globalen Koordinatensystem stehen. Die Berechnung der Punkte im globalen System geschieht mit der Formel 14. Weil die Matrizen der Orientierung und des Objektpunktes nicht in ihrer Dimension übereinstimmen, muss die Matrix des Objektpunktes mit dem Wert „1“ erweitert werden. Dies ist aber nur für die theoretische Berechnung entscheidend, der Maßstab spielt bei der Bestimmung der Koordinaten im globalen System keine Rolle, daher kann bei der Implementierung auf die Erweiterung des Objektpunktes verzichtet werden. Die eigentliche Transformation geschieht für jedes Element einzeln und besteht aus Multiplikationen und Additionen.

Formel 14: Transformation in das globale Koordinatensystem

$$Orientierungsmatrix: OM = \begin{bmatrix} Rotation_{11} & Rotation_{12} & Rotation_{13} & Translation_{14} \\ Rotation_{21} & Rotation_{22} & Rotation_{23} & Translation_{24} \\ Rotation_{31} & Rotation_{32} & Rotation_{33} & Translation_{34} \\ 0 & 0 & 0 & Maßstab \end{bmatrix}$$

$$Objektpunkt: OP_{Lokales\ System} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \rightarrow erweitert \rightarrow \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$X_{Globales\ System} = (OM_{11} \cdot OP_X) + (OM_{12} \cdot OP_Y) + (OM_{13} \cdot OP_Z) + OM_{14}$$
$$Y_{Globales\ System} = (OM_{21} \cdot OP_X) + (OM_{22} \cdot OP_Y) + (OM_{23} \cdot OP_Z) + OM_{24}$$
$$Z_{Globales\ System} = (OM_{31} \cdot OP_X) + (OM_{32} \cdot OP_Y) + (OM_{33} \cdot OP_Z) + OM_{34}$$

5.3.1.4 Abstand berechnen

Für die Berechnung des Abstandes eines Laserscanpunktes zur festgelegten Ebene steht eine Implementation von Lupos3D, in Form einer DLL-Datei, zur Verfügung. Die DLL-Datei ist der Grund dafür, dass eine Implementation des Algorithmus auf einem Linux-System nicht möglich ist. Diese Art von Dateien ist nur für Windows-Systeme gedacht. Für eine Übertragung auf ein Linux-System müsste die Bibliothek neu übersetzt werden. Dafür ist der Quellcode nötig, diesen wollte Lupos3D

jedoch nicht zur Verfügung stellen. Eine eigene Neuimplementation wäre jedoch nicht vergleichbar mit dem Berechnungsalgorithmus von Lupos3D.

Der Ausdruck DLL steht für „**D**ynamic **L**ink **L**ibary“, also eine dynamische Bibliothek. Diese Bibliotheken enthalten Programmcode, der in der Regel von mehreren Programmen ausgeführt wird. DLL-Dateien existieren nur auf dem Microsoft Betriebssystem Windows zur Verfügung. Unter Linux existiert die gleiche Technologie der dynamischen Bibliothek für Programme, jedoch sind die beiden Technologien inkompatibel. Der Quellcode für die Bibliothek muss somit für jedes Betriebssystem neu kompiliert werden. Dennoch hat die dynamische Bibliothek den Vorteil, dass nicht jedes Programm, das den gleichen Code verwendet, den kompletten Programmcode selbst implementieren muss. Somit kann der von Programmen benötigte Speicherplatz reduziert werden. Mit der mehrfachen Verwendung einer DLL-Datei ergibt sich ein weiterer Vorteil, sollten kleine Fehler im Code der DLL erkannt werden. Solche Fehler lassen sich durch nur eine Änderung an der DLL-Datei auf alle Programme übertragen, die die DLL-Datei nutzen.
Um die Funktionalitäten, die in der DLL-Datei stecken, anzusprechen, wird eine entsprechende Headerdatei benötigt. Diese Headerdatei beschreibt die Funktionen und Klassen der DLL-Datei und mit welchen Parametern die Funktionen angesprochen werden und welche Rückgabewerte zu erwarten sind. Weiterhin wird bei der Implementation eine vorkompilierte Bibliotheksdatei (*.lib) benötigt. Diese wird dem Linker beim Erstellen des eigentlichen Programmes hinzugefügt. Nach dem Kompilieren des Programmes werden dann jedoch nur noch die ausführbare Datei des Programmes und die DLL-Datei benötigt. Danach ist die DLL-Datei auch austauschbar das heißt, eine neue Version kann an ein schon existierendes Programm gegeben werden.

Wie schon angesprochen, wird für die Berechnung des Abstandes eines Laserscanpunktes von der ermittelten Ebene eine Funktion aus einer DLL-Datei von Lupos3D benutzt. Die Abbildung 5-6 zeigt den Aufruf dieser Funktion. Die Funktion verarbeitet dabei genau eine komplette Datenzeile des Laserscans. Die Funktion aus der DLL-Datei berechnet für jede Zeile des Laserscans die Ebene aus drei festgelegten Punkten. Des Weiteren bestimmt die Funktion den Abstand jedes Messpunktes, der aktuellen Datenzeile, von der Ebene und prüft anschießend ob der Abstand im Toleranzbereich liegt. Der Toleranzbereich gibt an, ob ein Punkt, der nicht genau auf der Ebene liegt, aufgrund seines Abstandes noch als zur Ebene gehörend gewertet wird. Ist das der Fall, wird der Messpunkt in den Ergebnisvektor geschrieben und anschließend in die Ergebnisdatei. In der Zeile 1 der Abbildung 5-6 wird die, für den Zugriff auf die Funktionen der DLL, benötigte Headerdatei in den Programmcode eingebunden. In der Zeile 3-7 werden für die DLL-Funktion benötigten Variablen Objekte erzeugt. Bei dem Objekt in der Zeile 3 handelt es sich um eine selbst definierte Struktur, diese wurde schon im Kapitel 5.3.1.1 eingeführt. Der Aufruf der DLL-Funktion erfolgt in der Zeile 9 über den in der DLL erstellten Namensraum (dllTest). Bei den Parametern 1-9 der Funktion handelt es sich um die Koordinaten der Punkte, mit denen die Ebene gebildet wird. Der Parameter 10 steht für einen Offset der Höhe der Ebene. Grundrisse werden normalerweise nicht in Bodenhöhe erstellt, sondern auf einer Höhe von rund einem Meter. Der Bezug ist jedoch trotzdem der Boden, deshalb wird die Ebene mit den drei Koordinaten in den Boden gelegt. Mittels des Offsets kann dann die komplette Ebene in der Höhe verschoben werden, ohne die Neigung zu verändern. Der Parameter 11 steht für den Toleranzbereich. Die Parameter 12 und 13 stehen für den Vektor mit der Eingangsdatenzeile und die Anzahl der Messpunkt in dieser Zeile. Die letzten Parameter, 14 und 15, stehen für den Vektor, der die Messpunkte aufnimmt, die zur Ebene gehören und die Anzahl der gefundenen Punkte im Ergebnis.

```
1  #include "dll_easy.h"           //Einbinden der Headerdatei der DLL
2
3  Schnitte    def_Schnitt;        //Selbst definierte Struktur
4  float       *pointsIn;          //Eingangszeile
5  float       *pointsOut;         //Ergebniszeile
6  int         width;              //Anzahl Messpunkte/Zeile
7  int         countOut;           //Anzahl der Punkte in der Ebene
8
9  int result = dllTest::Schnitt( //Aufruf der Schnitte-Funktion
10         def_Schnitte.def_Ebene.p1x,
11         def_Schnitte.def_Ebene.p1y,
12         def_Schnitte.def_Ebene.p1z,
13         def_Schnitte.def_Ebene.p2x,
14         def_Schnitte.def_Ebene.p2y,
15         def_Schnitte.def_Ebene.p2z,
16         def_Schnitte.def_Ebene.p3x,
17         def_Schnitte.def_Ebene.p3y,
18         def_Schnitte.def_Ebene.p3z,
19         def_Schnitte.abstand,
20         def_Schnitte.toleranz,
21         pointsIn, width,
22         pointsOut,countOut);
```

Abbildung 5-6: Zugriff auf eine DLL-Funktion

Wie der Abstand eines Punktes von der Ebene berechnet wird, ist bei der Funktion nicht ersichtlich und der genaue Quellcode ist auch nicht verfügbar. Aber der Abstand berechnet sich nach der Formel 4 und der dazugehörigen Vereinfachung durch die Formel 12.

5.3.1.5 Lesen / Schreiben

Bei diesen beiden Teilaufgaben handelt es sich theoretisch um zwei einzelne Aufgaben. Beide Teilaufgaben beschreiben jedoch die Arbeit mit Dateien, deshalb sind sie in einem Punkt zusammengefasst.

Das Lesen bezieht sich auf die Datei, die die Daten des Laserscans enthält und welche die Grundlage für die Berechnungen bildet. Da es sich bei der Eingangsdatendatei um ein von Lupos3D entwickeltes binäres Format handelt, muss ganz genau auf die Formatbeschreibung geachtet werden. Eine fehlerhafte Lesefunktion erzeugt auch fehlerhafte bzw. unbrauchbare Daten. Um korrekte Lesevorgänge zu gewährleisten, werden bei dem Algorithmus von Lupos3D geschriebenen und zur Verfügung gestellten Funktionen verwendet. Mit diesen Funktionen lassen sich beispielsweise die Headerinformationen, eine ganze Zeile oder auch nur Teile einer Zeile aus der Eingangsdatendatei lesen. Die Abbildung 5-7 zeigt wie eine Datei im Lupos3D-Format (*.ptb) geöffnet und verarbeitet wird. Zunächst ist es notwendig von Lupos3D erstellte Klasse, welche die Implementierungen der von Dateiarbeitsfunktionen über dem PTB-Format ermöglicht, zu verwenden. Dafür muss die Headerdatei (mffptb.h) inkludiert und die Implementationsdatei (mffptb.cpp) kompiliert werden. Somit ist es nun möglich ein Objekt dieser Klasse zu erstellen (Zeile 3). Die Definition des Konstruktors der Klasse verlangt, dass beim Erstellen eines Objektes eine Datei an das Objekt übergeben wird. Jeweils der erste Schritt nach dem Anlegen eines Objektes der Klasse ist das Prüfen der Gültigkeit der übergebenen Datei (Zeile 4). Dabei liest die verwendete Funktion an einer, in der Formatbeschreibung definierten, Stelle einen Teil der binären Daten. Dieser Teil wird in eine Zeichenkette umgewandelt und mit einer Kontrollzeichenkette verglichen. Die Datei gilt als gültig, wenn die beiden Zeichenketten übereinstimmen. Durch diese Prüfung ist sichergestellt, dass alle weiteren Funktionen über der Eingangsdatendatei korrekt arbeiten können. Es können aber denn noch Probleme auftreten, falls die Datei durch unvorhersehbare Ereignisse beschädigt ist. Die Prüfung bezieht sich also nur darauf, ob eine binäre Datei im korrekten Format vorliegt und nicht ob die Datei fehlerfrei ist. In den Zeilen 5-7 sind drei Beispiele von Funktionen aufgeführt, mit denen es möglich ist, Headerinformationen aus einer Eingangsdatendatei zu extrahieren. Die Funktion „height()“ liefert als Rückgabewert die Anzahl von Zeilen, die sich der Datei befinden. Während die Funktion „width()“ die Anzahl von Punkten pro Zeile zurückgibt. Ein Punkt kann je nach dem Format

des Punktes (Typ0-Typ3) aus einer unterschiedlichen Anzahl von Elementen bestehen, siehe Kapitel 2.1 auf Seite 2. Aus wie vielen Elementen ein Punkt besteht, kann mit der Funktion „valuesPerPixel()“ ermittelt werden. Das Lesen einer Zeile aus der Eingangsdatendatei findet in den Zeilen 9 und 10 statt. Für die Funktion „readRow()“ müssen bestimmte Parameter gesetzt werden, je nachdem wie die Zeile gelesen werden soll. Zunächst muss immer ein Pointer auf ein Feld von „float“-Werten angelegt werden, in dem das Ergebnis der Leseoperation gespeichert wird. Die Größe des benötigten Feldes ergibt sich dabei daraus, wie viele Punkte aus der Zeile gelesen werden sollen. In dem Beispiel in Zeile 9 ist das Feld so groß, dass es eine komplette Zeile in sich aufnehmen kann. Die Anzahl der maximalen Punkte in einer Zeile (width) multipliziert mit der Anzahl der Elemente pro Punkt (valPerPix) ergibt die maximale Anzahl von „float“-Werten in einer Zeile. Das eigentliche Lesen findet in der Zeile 10 statt. Die Funktion „readRow()“ muss jedoch korrekt initialisiert werden. Beim ersten Parameter der Funktion handelt es sich um eine Referenz auf ein Feld, in dem die gelesenen Werte gespeichert werden sollen. Der zweite Parameter gibt die Zeile an, die gelesen werden soll, in diesem Fall die Zeile 15. Soll eine komplette Eingangsdatendatei eingelesen werden, wird nur eine „for“-Schleife benötigt mit der alle Zeilen der Datei iterativ durchlaufen werden beginnend bei null und endet bei der maximalen Anzahl an Zeilen (height). Der dritte und der vierte Parameter geben an, von welchem Punkt in der Zeile bis zu welchem Punkt in der Zeile gelesen werden soll. Im Beispiel wird die komplette Zeile eingelesen von Punkt Null bis zum Maximum an verfügbaren Punkten (width).

```
1  #include "mffptb.h"                                    // Inkludieren der Klasse mit den Lese- & Schreiboperationen
2
3   MFFPTB ptb_file(InFile);                              // Ein Objekt der Klasse erzeugen -> Datei zum lesen übergeben
4   if(ptb_file.isValid()){                               // Datei auf Gültigkeit prüfen
5       unsigned int height = ptb_file.height();          // Möglichkeit des lesens von Headerinformationen
6       unsigned int width = ptb_file.width();            //              ...
7       unsigned int valPerPix = ptb_file.valuesPerPixel();//             ...
8
9       float* PointsIn = new float[width*valPerPix];     // Pointer für die Aufnahme einer gelesenen Zeile
10      ptb_file.readRow(PointsIn, 15, 0, width);         // Lesen der ganzen Zeile '15' -> speichern in PointsIn
11  }
```

Abbildung 5-7: Lupos3D Funktion -> Lesen einer Zeile

Zum Schreiben existieren in der inkludierten Klasse auch Funktionen von Lupos3D, jedoch werden diese beim Schnitte-Algorithmus nicht benötigt. Da sie nur zum Schreiben von Dateien im PTB-Format verwendet werden können. Für das Ausgeben der Ergebnisse des Schnitte-Algorithmus reicht eine einfache Textdatei. Zum Schreiben von Dateien mit einem freien Format existieren in C++ Standardfunktionen. Um die Ergebnisse später korrekt darstellen zu können, wird auch bei dieser Ausgabe eine gewisse Struktur benötigt. Diese kann jedoch durch geschicktes Verketten der einzelnen Ergebnisse erreicht werden. Die Abbildung 5-8 zeigt die Implementierung des Schreibens der Ergebnisse. Notwendig zum Erstellen der Ergebnisdatei ist die Klasse „fstream“. Durch das Inkludieren der Klasse (Zeile 1) erhält der Entwickler Zugriff auf Ein- und Ausgabestreams über Dateiobjekten. In der zweiten Zeile wird ein Objekt der „fstream“-Klasse erzeugt. Mit der Funktion „open()“ in der vierten Zeile wird der File-Stream (fstream) auf eine Datei verwiesen (erster Parameter). Gleichzeitig wird auch noch die Richtung des Streams festgelegt (zweiter Parameter), in diesem Fall handelt es sich um einen Ausgabestream. Mit dem zweiten Parameter können verschiedene Modi zum Öffnen der Datei angeben werden. Beispielsweise kann die Datei nur lesend, lesend und schreibend oder auch nur schreibend, wie in der Abbildung 5-8, geöffnet werden. Bei den Operationen der Zeilen 6-9 handelt es sich um das eigentliche Schreiben in die Datei. In dem Beispiel ist auch schon die Struktur der Ergebnisdatei des Schnitte-Algorithmus zu erkennen. Bei der Berechnung wird die Koordinate (X, Y, Z) eines Punktes der zur definierten Ebene gehört, jeweils in eine Zeile der Ergebnisdatei geschrieben. Als Trennzeichen zwischen den Werten wird das

Leerzeichen benutzt. Die Reihenfolge der Werte in eine Zeile lautet „X-Wert␣Y-Wert␣Z-Wert“. Die „close()“-Funktion in der Zeile 10 schließt die zuvor geöffnete Datei. Bevor die Datei aber vom Stream gelöst wird, werden alle noch nicht in die Datei geschriebenen Daten verarbeitet. Das Schreiben bei der parallelen Implementierung ist etwas komplexer und wird in dem Kapitel 5.3.2 detaillierter betrachtet.

```
1  #include <fstream>                    // Inkludiert die Klasse FileStream
2  fstream out;                          // Ein Objekt der Klasse erzeugen
3
4  out.open(OutFile, std::ios::out);     // Datei zum schreiben öffnen
5
6  out << Ergebiss1 << " "               // Ergebnisausgabe strukturieren
7      << Ergebiss2 << " "               // und dem Datei Stream
8      << PErgebiss3                     // übergeben
9  << endl;
10 out.close();                          // geöffnete Datei schließen
```

Abbildung 5-8: Ergebnisse in Datei schreiben

5.3.2 Parallelisierung

Neben der Implementation einer sequenziellen Variante des Algorithmus zu Vergleichszwecken wurden auch zwei parallele Implementierungen vorgenommen. Beide parallelen Varianten beruhen auf dem Thread-Ansatz, jedoch wurden zwei unterschiedliche Thread-Bibliotheken verwendetet. Zum einen die Windows-Threads[42] und zum anderen die Posix-Threads[43]. Aus der Implementierung zweier unterschiedlicher Bibliotheken lässt sich vielleicht ein Rückschluss auf die effizientere Bibliothek ziehen. Es kann durchaus vorkommen, dass eine Bibliothek für die Initialisierung von Threads weniger Zeit benötigt. Das wiederum kann einen Vorteil bei der Laufzeit der Berechnungen zur Folge haben.

5.3.2.1 Automatisches Erkennen der Prozessoranzahl

Die Implementierung der Threads orientiert sich dabei an der zeilenweisen Verarbeitung der Eingangslaserscandatei. Ziel der parallelen Implementierung ist es jedem Thread einen Bearbeitungsbereich in der Laserscandatei zu zuteilen. Dabei stellt sich das Problem, wie viele Threads gestartet werden sollten. Bei dieser Implementation des Schnitte-Algorithmus existiert die Möglichkeit einer manuellen und einer automatischen Festlegung. Die manuelle Festlegung wird über das Angeben der zu startenden Threads als Startparameter des Programmes realisiert. Für die automatische Festlegung wird die Anzahl der im System zur Verfügung stehenden Prozessoren genutzt. Diese Informationen sind unter Windows in den Systemeigenschaften gespeichert. Die Abbildung 5-9 zeigt, wie auf Systemeigenschaften unter C++ zugegriffen werden kann. Dazu werden Funktionen der Standardbibliothek (windows.h) verwendet.

Die eigentliche Bibliothek, die für die Ermittlung der Systeminformationen zuständig ist, ist „kernel32.dll“ auf Windows Systemen. Der Zugriff auf die Funktionen der DLL erfolgt über die Headerdatei „winbase.h“. Diese Bibliothek ist für die Kerneldienste zuständig. Der Kernel regelt die Prozess- und Datenorganisation und ist somit ein zentraler Bestandteil des Betriebssystems. Da es sich bei der Bibliothek um eine so systemnahe Komponente handelt, ist es nicht möglich diese direkt in ein Programm einzubinden, da zu viele Abhängigkeiten zu anderen Bibliotheken bestehen. Deshalb wird die Headerdatei „windows.h“ inkludiert, diese bildet den Container für diese Funktionalitäten.

Das Ergebnis der Systeminformationsabfrage wird auf einer entsprechenden Variablen gespeichert. Die mit dieser Methode ermittelte Anzahl an physisch verfügbaren Prozessoren ist dynamisch das bedeutet, wird das Programm auf einem anderen Windows Computer ausgeführt

[42] Siehe Kapitel 3.3.2.1 auf Seite 20
[43] Siehe Kapitel 3.3.2.2 auf Seite 21

passt es sich automatisch an das dortige System an. Mit der Information der verfügbaren Prozessoren kann das Programm nun für jeden Prozessor einen Thread starten.

```
1  unsigned int Prozessor = 0;                  // Anlegen einer Variablen für die Prozessoranzahl
2
3  _SYSTEM_INFO si;                             // Anlegen eines Objektes für die Systemeigenschaften
4  GetSystemInfo(&si);                          // Speichern der Systemeigenschaften auf dem Objekt
5  Prozessor = si.dwNumberOfProcessors;         // Abfragen der Prozessorenanzahl
```

Abbildung 5-9: Ermittelung der Prozessoranzahl

5.3.2.2 Definieren des Arbeitsbereiches

Nachdem die Anzahl der Prozessoren des aktuellen Systems bestimmt wurden, muss eine große Aufgabe in kleinere Teilaufgaben zerlegt werden. Damit das Programm auch weiterhin dynamisch arbeitet, muss die Zerteilung ebenfalls dynamisch sein. Die zeilenweise organisierten Daten von Lupos3D eignen sich sehr gut für eine dynamische Aufteilung. Der Theorie nach müssen die verfügbaren Zeilen/Aufgaben in der Laserscandatei lediglich gleichmäßig auf die verfügbaren Threads aufgeteilt werden. Die Formel 15 beschreibt mathematisch das Vorgehen bei der Aufteilung.

Formel 15: Berechnung des Arbeitsbereiches

$$Arbeitsbereich\colon\ Anzahl_{Zeilen} / Anzahl_{Prozessoren}$$

$$Rest\colon Anzahl_{Zeilen} \% Anzahl_{Prozessoren}$$

Zunächst muss ein Grundarbeitsbereich gefunden werden, dieser ergibt sich aus der Anzahl der zu verarbeitenden Zeilen in der Laserscandatei und der Anzahl der zur Verfügung stehenden Prozessoren bzw. der Anzahl der zu startenden Threads. Der Arbeitsbereich definiert dabei nicht, welche Zeilen ein Thread bearbeiten soll, sondern wie viele jeder Thread mindestens bearbeiten muss. Es ist auch möglich, dass eine gewisse Anzahl an Zeilen bei der Festlegung des Arbeitsbereiches übrig bleibt. Diese Zeilen werden als Rest bezeichnet, die Anzahl der Rest-Zeilen kann durch eine Modulo-Operation bestimmt werden. Die nächste Frage, die nun aufkommt ist, wie die restlichen Zeilen auf die Threads aufgeteilt werden. Für diese gerechte Verteilung kommen bei der hier implementierten Prozessparallelität mehrere Konzepte infrage. Zum einen können die Zeilen absolut gleich verteilt werden das heißt, jeder Prozess bekommt als Grundlage die gleiche Menge an Zeilen, und falls ein Rest bestehen bleibt, wird dieser ebenfalls gleichmäßig unter den Prozessen aufgeteilt. Ein Unterkonzept des soeben Beschriebenen ist, dass ein Rest nicht auf alle Prozesse verteilt wird, sondern der als Erstes gestartete Prozess diesen Rest verarbeiten muss. Dieser kann schon mit der Arbeit beginnen, während das System noch die verbleibenden Prozesse startet. Als Folge draus könnte sich die gesamte Laufzeit des Programmes verringern, obwohl ein Prozess länger arbeitet als die Anderen. An dieses Unterkonzept setzt auch das zweite Hauptkonzept an. Dort wird die Arbeit asymmetrisch verteilt. Die Prozesse, die als Erstes gestartet werden, erhalten grundsätzlich mehr Zeilen als die später gestarteten. Der Vorteil ist bei diesem Konzept, dass kein Rest bestehen bleibt und die Laufzeit optimiert wird, da es kaum zu Wartezeiten bei der Zusammenführung der Prozesse kommt. Ein Nachteil ist jedoch, dass sich eine ausgeglichene Mehrbelastung schwerer finden lässt als eine komplette Gleichverteilung. Bei der aktuellen Implementierung (Stand: 5. Juni 2010) kommt das Konzept der absoluten Gleichverteilung zum Einsatz, da sich dieses auch am leichtesten Implementieren lässt. Für jeden Thread werden ein Startwert („von“) und ein Endwert („bis) festgelegt. Der Startwert bezeichnet die Zeile der Eingangsdatei, von der an ein Thread mit der Berechnung beginnt und der Endwert bezeichnet die letzte Zeile, die vom Thread verarbeitet wird. Für den ersten Thread wird der Startwert auf „0“ festgelegt, die Thread-Funktion beginnt dann in der

ersten Zeile mit dem Lesen. Die Festlegung des Endwertes setzt sich aus zwei Fragen zusammen, „Gibt es einen Rest?“ und „Wie groß ist der Rest“. Jeder Thread bekommt die Grundaufgabenmenge plus eine Aufgabe vom Rest zugeteilt. Die Restaufgabenmenge wird bei jeder Zuteilung um eins reduziert. Falls der Rest vollständig verteilt ist, bearbeiten die noch nicht gestarteten Threads lediglich ihre Grundaufgabenmenge. In Abbildung 5-10 ist ein Beispiel beschrieben, mit insgesamt 1432 Aufgaben/Zeilen und drei Prozessen/Threads. Wichtig ist bei diesem Beispiel, dass die Zeilen von „0“ anfangen werden zu zählen, deshalb ist das Ende bei 1431 erreicht.

$$Aufgaben/Zeilen = 1432$$

$$Prozessoren/Threads = 3$$

$$Arbeitsbereich = \frac{1432}{3} = 477$$

$$Rest = (1432\%3) = 1$$

i	von	bis	Rest
0	$i \cdot ws = 0$	$\big((i+1) \cdot ws\big) + 1 = 478$	1 → 0
1	$(i \cdot ws) + 1 = 478$	$\big((i+1) \cdot ws\big) = 954$	0 → -1
2	$i \cdot ws = 954$	$\big((i+1) \cdot ws\big) = 1431$	-1

Abbildung 5-10: Beispielrechnung für den Arbeitsbereich

5.3.2.3 Konfigurieren der Threads

Mit der Bestimmung des Arbeitsbereiches des jeweiligen Threads ist die Vorbereitung der Konfiguration der Threads abgeschlossen. In dem nun folgenden Kapitel wird das direkte Vorgehen bei der Konfiguration der Threads beschrieben.

Der erste Schritt war im Quellcode einen Bereich zu identifizieren, der sich für eine parallele Bearbeitung eignet. Dieser Schritt sollte an dieser Stelle schon beendet sein. In den weiteren Schritten wird der zu parallelisierende Quellcode, bei diesem Beispiel das Lesen, Berechnen und Schreiben von Informationen, in eine eigene Funktion ausgelagert. Die erstellte Thread-Funktion wird von jedem gestarteten Thread ausgeführt. Das kann in manchen Fällen dazu führen, dass der Teil des Quellcodes der in die Funktion ausgelagert wurde, in einem stärkeren Maß verallgemeinert werden muss. Der Sinn einer Verallgemeinerung liegt darin, dass die Spezialisierung vom Ein-Thread-Quellcode aufgebrochen wird und mehrere Threads mit dem gleichen Quellcode operieren können. In dem, in diesem Kapitel vorgestellten, Beispiel liegt die Verallgemeinerung darin, dass ein Thread mit gewissen Parametern ausgestattet wird. Durch die Verwendung der Parameter lässt sich die verallgemeinerte Thread-Funktion wieder so weit spezialisieren, dass der Thread nur noch die Zeilen der Eingangslaserscandatei bearbeitet die ihm durch die Bestimmung des Arbeitsbereiches zugeteilt wurden.

Für die einfache Verwaltung der Parameter der einzelnen Threads wurden verschiedene Strukturen definiert. Bei einer Struktur handelt es sich um einen aus mehreren Datentypen oder Strukturen zusammengesetzten neuen Datentyp. In der programmierten Anwendung werden insgesamt drei neue Datenstrukturen definiert. Die Abbildung 5-11 zeigt den Quelltext zu den drei neuen Strukturen. In der Struktur „Ebene“ sind die vom Nutzer festgelegten Parameter für die Definition der Ebene gespeichert. Die Struktur „Schnitte“ enthält die Struktur „Ebene“ plus drei

weitere Variablen, die die Schnitte-Funktion, der DLL von Lupos3D, für die Berechnungen benötigt. Unter anderem auch einen Pointer auf die Ergebnisdatei, in die abwechselnd von den Prozessen geschrieben werden soll, siehe Kapitel 5.3.2.4. Das Anlegen der Strukturen „Ebene“ und „Schnitte“ ist insofern sinnvoll, weil diese für jeden Prozess gleich sind und so übersichtlich gespeichert werden können. Die Zuweisung und der Zugriff auf die eigentlichen Werte hinter den Strukturen sind somit ebenfalls stark vereinfacht. Die Struktur „ThreadArg“ beinhaltet alle Parameter und Strukturen, die für jeden einzelnen Prozess benötigt werden. Es werden einige Variablen für die Zeitnahme bereitgestellt. Die Struktur enthält auch Variablen, die den ermittelten Arbeitsbereich eines Threads aufnehmen können, und weitere für die Steuerung der Prozesse. Des Weiteren werden auch jeden Thread die Eingangsdatendatei und ein Zeiger auf einen kritischen Abschnitt übergeben, siehe Kapitel 5.3.2.4. Das eigentliche Konfigurieren der Threads geschieht durch das füllen der Datenstrukturen mit Werten. Die Werte für die Definition der Ebene sind ausdrücklich (hart implementiert) in den Quelltext geschrieben und orientieren sich an den Vorgaben aus einem sequenziellen Beispielprogramm von Lupos3D für die Berechnung der Schnitte in einer Laserscandatei. Auch die Parameter für die Schnitte-Struktur sind aus diesem Programm übernommen, damit später die Ergebnisse verglichen werden können. Für die Ausgabedatei wird ein Stream angelegt, über den alle Threads ihre Ergebnisse schreiben. Dieser Stream wird jedem Thread als Zeiger übergeben. Durch das Definieren nur eines Ergebnisstreams wurde ein kritischer Abschnitt in dem Programm geschaffen. Wie die genaue Funktionalität hinter dem parallelen Schreiben von Informationen über einen gemeinsamen Stream aussieht, beschreibt das Kapitel 5.3.2.4. Die Struktur „ThreadArg“ beinhaltet letzten Endes alle Parameter, die ein Thread für die Berechnung benötigt. Durch die Definition unterschiedlicher Arbeitsbereiche für jeden Thread ist es notwendig mehrere Instanzen der „ThreadArg“-Struktur anzulegen und an den einen Thread zu binden. Wie viele Instanzen der Struktur erstellt werden müssen, muss dynamisch zur Laufzeit des Programmes ermittelt werden. Logischerweise werden genauso viele Instanzen benötigt, wie auch Threads gestartet werden. Eine dynamische Erzeugung der Struktur lässt sich nur mit einem dynamischen Speichermanagement realisieren. Es ist also notwendig, einen Pointer auf die Struktur „ThreadArg“ zu erstellen und somit dynamisch den benötigten Speicher zur Laufzeit des Programmes auszufassen. Das so erstellte Feld an „ThreadArg“-Strukturen lässt sich mittels einer „For“-Schleife leicht konfigurieren. Jede „ThreadArg“-Struktur innerhalb des Feldes wird individuell für einen Thread konfiguriert. Die Abbildung 5-12 zeigt, wie die Felder der „ThreadArg“ Strukturen mit Werten gefüllt werden und wie der Arbeitsbereich auf die einzelnen Threads verteilt wird. Es ist zu erkennen, dass die „ThreadArg“-Struktur zum Teil mit Default-Werten und schon ermittelten Werten gefüllt wird. Zu den Default-Werten zählen alle Variablen, die sich mit Zeitmessungen (siehe Abbildung 5-11 alle Variablen vom Typ „clock_t“) beschäftigen und der Zähler („cnt“) für die Anzahl der berechneten Zeilen, anhand der dort vergebenen Werte lässt sich später leichter feststellen, ob während der Ausführung eines Threads ein Fehler auftrat. Die ermittelten Werte wie „width“ und „valPerPixel“ werden als Variable mitgeführt. Sie sind zwar auch mit einem Aufruf der jeweiligen Funktion über der Eingangsdatendatei verfügbar, jedoch könnte es zu Zeitverzögerungen kommen, wenn mehrere Prozesse gleichzeitig versuchen die Funktionen über der gemeinsamen Datei auszuführen. Um dieses Problem auszuschließen, werden die beiden Werte einmalig vom Hauptprozess erhoben. Bei den Variablen „file“ und „def_schnitte“ handelt es sich lediglich um die zuvor definierten Werte, die an jeden Prozess weitergeleitet werden müssen. Der Variable „mutex“ wird ein Zeiger auf den im Hauptprozess angelegten kritischen Abschnitt übergeben, damit alle verwendeten Prozesse auf den gleich Abschnitt zugreifen. Wie die Threads gestartet werden, ist abhängig von der verwendeten

Bibliothek. Die Kapitel 5.3.2.5 und 5.3.2.6 erläutern diese Problematik genauer, anhand der bei diesem Programm verwendeten Thread-Bibliotheken.

```
typedef struct {
    double p1x, p1y, p1z, p2x, p2y, p2z, p3x, p3y, p3z;
} Ebene;

typedef struct {
    Ebene def_Ebene;
    double abstand, toleranz;
    std::fstream * out;
} Schnitte;

typedef struct {
    unsigned int von, bis, width, cnt, valPerPixel;
    clock_t time_calc, time_total, LOCK_TO_UNLOCK,
            WRITE_TO_UNLOCK, LOCK_TO_WRITE, read;
    std::string file;
    Schnitte def_Schnitte;
    CMutex * mutex;
} ThreadArg;
```

Abbildung 5-11: Definition neuer Datenstrukturen

```
ThreadArg *varg;
varg = new ThreadArg[Prozessor];
int rest = (height%Prozessor); //Rest berechnen für die spätere optimale Verteilung
for(unsigned int i(0); i<Prozessor; i++){//Anzahl der Zeilen überprüfen
    if(i==0){varg[i].von = (i*ws);}
    else if(rest>0) {varg[i].von = (i*ws)+1;}
    else if(rest==0) {varg[i].von = (i*ws)+1; rest--;}
    else {varg[i].von = (i*ws);}
    if((height%Prozessor != 0) && (rest > 0)){varg[i].bis = ((i+1)*ws)+1; rest--;}
    else{varg[i].bis = ((i+1)*ws);}
    varg[i].width = width;
    varg[i].valPerPixel = valPerPixel;
    varg[i].file = In_File;
    varg[i].cnt = -1;
    varg[i].time_total = -1;
    varg[i].time_calc = -1;
    varg[i].read = -1; // read
    varg[i].def_Schnitte = schnitte;
    varg[i].mutex = &s_mutex;
    varg[i].LOCK_TO_UNLOCK = 0;
    varg[i].LOCK_TO_WRITE = 0;
    varg[i].WRITE_TO_UNLOCK = 0;
}
```

Abbildung 5-12: Konfiguration der Thread-Parameter

5.3.2.4 Paralleles Schreiben

Im Gegensatz zum sequenziellen schreiben von Dateien auf die Festplatte, wie in Kapitel 5.3.1.5 beschrieben, bringt das parallele Schreiben ein paar Hindernisse mit sich. Vereinfacht ausgedrückt versuchen mehrere Personen gleichzeitig mit einem Stift auf ein Blatt Papier zu schreiben, das kann gar nicht ohne Probleme funktionieren. Der Stift in dem Beispiel stellt somit einen Flaschenhals und auch einen kritischen Abschnitt im Programm dar. Das Ergebnis eines unkontrollierten Schreibens ist ein totales Wirrwarr an Zeichen. Da nicht sichergestellt werden kann, wann Daten von welchem Akteur geschrieben werden. Damit ein paralleles Schreiben funktionieren kann, muss der Zugriff auf den kritischen Abschnitt kontrolliert werden. Dazu existieren in der Programmierung mit C++ zwei Möglichkeiten, zum einen die sogenannten Semaphoren und zum anderen der Mutex. Beide bezeichnen einen kritischen Abschnitt, unterscheiden sich jedoch in der Art der Kontrolle des Abschnittes. Die theoretische Funktionsweise ist recht simpel, Objekte können die Zuteilung einer bestimmten kritischen Ressource anfordern. Ist die Ressource gerade nicht in Verwendung oder

existieren noch Zugriffsberechtigungen, so kann das Objekt mit der Ressource arbeiten. Existieren jedoch keine freien Zugriffsberechtigungen mehr, so muss das Objekt warten, bis wieder eine Berechtigung frei wird. Bei einem Semaphor ist es möglich, dass ein oder mehrere berechtigte Objekte eine kritische Ressource verwenden können. Während ein Mutex nur einem Objekt einen Zugriff gewährt. Für das parallele Schreiben bietet sich ein Mutex an, denn es steht normalerweise nur ein Schreibkopf auf der Festplatte zur Verfügung.

Für die Implementation eines Mutex in C++ stehen Standardklassen zur Verfügung, mit denen sich leicht der Zugriff auf bestimmte Bereiche oder Ressourcen kontrollieren lässt. Die Standardklasse, die in diesem Fall für die Erzeugung eines Mutex genutzt wurde, stammt aus den Microsoft Foundation Classes (MFC)[44]. Um die Synchronisationsklassen verwenden zu können, ist es nötig die Headerdatei „afxmt.h“ zuinkludieren. Diese enthält die notwendigen Klassendefinitionen und ist ein Bestandteil der MFC. Der erste Schritt ist die Definition eines Synchronisationsobjektes, dieses besitzt den Basistyp „CSyncObject“. Abgeleitet von dieser Klasse sind verschiedene Synchronisationsobjekte. Darunter fallen die Klassen „CEvent“, „CCriticalSector“, „CSemaphore“ und auch die Klasse „CMutex“. Für die Implementation in diesem Fall ist die Klasse „CMutex“ am Sinnvollsten, denn sie erlaubt nur einem Thread den Zugriff auf die gemeinsam verwendeten Daten oder Ressourcen. Die Abbildung 5-13 zeigt eine mögliche Implementation. Der erste Schritt ist somit das Definieren des Synchronisationsobjektes, durch das Anlegen eines Objektes vom Typ „CMutex“ (Zeile 2).

In der dritten Zeile der Abbildung 5-13 wird die Zugriffsberechtigung für das Synchronisationsobjekt festgelegt. Bei der vorgenommenen Implementation handelt es sich dabei um ein „CSingleLock“-Objekt. Das Objekt regelt, dass die parallel ausgeführten Threads nur auf ein einziges Objekt warten müssen. Diese Zugriffsberechtigung kann nur gewählt werden, wenn sich die Threads nur eine gemeinsame Ressource teilen, in diesem Fall den Ausgangsstream[45]. Für den Fall das sich die Threads mehrere Ressourcen teilen müssen steht die Klasse „CMultiLock“ zur Verfügung. Dem Sperrobjekt vom Typ „CSingleLock“ stehen drei Member-Funktionen zur Verfügung, die wichtigsten sind das Sperren und das Freigeben von Ressourcen. Wie ein Objekt gesperrt wird, zeigt die Zeile 4 der Abbildung 5-13. Nach dem Sperren können die Operationen durchgeführt werden, die nur von einem Thread zur gleichen Zeit durchgeführt werden sollten. In diesem Fall, ist dass das Schreiben der Ergebnisdaten den Ausgabestream. Sollte ein Thread versuchen, das Sperrobjekt zu sperren, obwohl es schon gesperrt ist, so ist das erstens nicht möglich und zweitens wartet der Thread so lange an dieser Stelle, bis das Sperrobjekt wieder freigeben wurde. Dann versucht der Thread das Sperrobjekt erneut zu sperren. Ein Problem hierbei ist die möglicherweise entstehende Wartezeit. Standardmäßig ist diese Wartezeit bei der „Lock“-Funktion auf unendlich gesetzt. Die Zeit kann jedoch auch manuell festgelegt werden. Bei der zweiten wichtigen Funktion handelt es sich um das Gegenstück zum Sperren. Die Freigabefunktion wird in der Zeile 6, mit „Unlock“, aufgerufen. Eine dritte Funktion, die die Klasse „CSingleLock“ noch zur Verfügung stellt, ist eine Abfrage, ob das Sperrobjekt schon gesperrt wurde. Die Funktion wird mit der Anweisung „isLocked“ aufgerufen, und gibt als Rückgabewert „true“ oder „false“ zurück.

Beim Nutzen von Synchronisationen zwischen Threads kann es am Anfang eines kritischen Abschnitts durchaus zu Wartezeiten kommen. Um die Wartezeit kurz zu halten, sollten die Synchronisationsbereiche so klein und kurz wie nur möglich gehalten werden, um eine zu lange Verweildauer im kritischen Abschnitt zu verhindern.

[44] Siehe Kapitel 3.3.2.1 auf der Seite 20
[45] Siehe Kapitel 5.3.1.5 auf Seite 37

```
1  #include <afxmt.h>       //Inkludieren der Klassen (MFC)
2  CMutex mutex;            //Syncronisationsobjekt anlegen
3  CSingleLock SL(&mutex); //Sperrobjekt der kritischen Sektion zuweisen
4  SL.Lock();               //Sektion sperren
5      ...                  //Operationen durchführen
6  SL.Unlock();             //Sektion freigeben
```

Abbildung 5-13: Anlegen eines Mutex

Für die Implementation des parallelen Schreibens wurde jedem Prozess ein Pointer auf den kritischen Abschnitt („CMutex“) in die Thread-Konfiguration hineingereicht. Somit arbeiten alle Threads über dem gleichen kritischen Abschnitt.

5.3.2.5 Windows-Threads

Wie schon in dem Kapitel 3.3.2.1 auf der Seite 20 erwähnt, werden die Windows Thread in der Klassenbibliothek MFC eingeführt. Bei MFC handelt es sich um eine große Sammlung von objektorientierten Klassen für die Programmierung auf Windowssystemen. Nähe Informationen zu der vollständigen MFC-Bibliothek können im Microsoft Entwickler Netzwerk nachgelesen werden (30).

Die Windows-Threads sind nur ein kleiner Teil der MFC-Bibliothek. Mehrere Threads innerhalb einer Applikation lassen sich mittels der Klasse „CWinThread“ erstellen. Um die Klasse verwenden zu können, muss die Headerdatei „afxwin.h“ in das Programm inkludiert werden. Die Klasse „CWinThread“ kann für zwei Typen von Threads eingesetzt werden, zum einen für Arbeiterthreads und zum anderen für Nutzer-Interface-Threads. Die Thread Klasse von Microsoft, die für die Erzeugung von Threads angedacht ist, gilt als „Thread Safe“. Dafür werden die lokalen Threaddaten, die vom Framework genutzt werden, um die threadspezifischen Daten zu erzeugen, von einem erzeugen „CWinTread“-Objekt verwaltet um die Synchronisation zu gewährleisten.

Die Abbildung 5-14 zeigt, wie eine schnelle Implementation von Windows-Threads vorgenommen werden kann. Im Wesentlichen beruht die Umsetzung der Threads auf zwei Komponenten, einer Thread-Funktion um der „CWinThread“-Klasse/Objekt. Die Funktionen, die für die Implementation benötigt werden, sind alle in der Headerdatei „afxwin.h“ enthalten. Der erste Schritt ist daher das Inkludieren dieser Datei im Programm (Zeile 1). Die Funktion, die von jedem Thread ausgeführt wird, wird in der Zeile 3-7 der Abbildung 5-14 implementiert. Wichtig dabei ist lediglich, dass die Funktion einen „unsigned Integer“ zurückgeben muss und dass sie nur einen einzigen, Datentyp unabhängigen, Übergabeparameter besitzen darf. Durch die Begrenzung der Anzahl der Übergabeparameter ist die Nutzung in erster Linie etwas umständlich, aber auch sinnvoll. Wie im Kapitel 5.3.2.3 beschrieben, ist es mittels von selbst definierten Datenstrukturen auch möglich mehrere Parameter an einen Thread zu übergeben. Der erste Schritt in der Thread-Funktion sollte dann natürlich sein, den Datentyp unabhängigen Übergabeparameter wieder in den benötigen Datentyp zu casten (Zeile 4). Wenn die Thread-Funktion erfolgreich beendet wurde, gibt sie den Wert „0“ zurück (Zeile 6). Im Hauptprozess des Programmes lassen sich nun beliebig viele Threads starten. Die in der Zeile 11 angelegt eigene Verwaltung der gestarteten Threads ist optional. Sie ist nicht unbedingt notwendig, aber sehr hilfreich, wenn der Hauptprozess angehalten ist zu warten, bis alle gestarteten Threads beendet wurden (Zeile 18). Diese Funktionalität ist jedoch auch sehr riskant, da Probleme in den ausgeführten Threads den Hauptprozess möglicherweise zum ewigen Warten zwingen. Ein geeigneter Wert für die maximale Wartezeit ist daher zu empfehlen, wenn im Vorfeld abgeschätzt werden kann, wie lange die Threads arbeiten. In der Zeile 12 wird das, in den ersten Abschnitten dieses Kapitels, schon angesprochene Objekt der Klasse „CWinThread“ angelegt. Mittels einer Schleife ist es nun möglich beliebig viele Threads über dem Objekt zu starten. Zum Starten der Threads existieren zwei Möglichkeiten. Zum einen das Verwenden der Funktion „CreateThread“ und

zum anderen die Funktion „AfxBeginThread“. Bei der Implementation der Programme dieser Masterthesis wurde nur die Funktion „AfxBeginThread“ verwendet. Mit dieser kann ein Thread wesentlich detaillierter direkt erzeugt und gestartet werden, während die Funktion „CreateThread“ die Möglichkeit bietet zwischen dem Erzeugen und dem Starten eines Threads zu unterscheiden. Detaillierter heißt im Fall von „AfxBeginThread“, dass dem Thread weitere Parameter mitgegeben werden können. Darunter fällt zum Beispiel die Priorität, mit der der Thread vom Betriebssystem ausgeführt wird.

```
1  #include <afxwin.h>                                          //Inkludieren der Klassen (MFC)
2
3  UINT meineThreadFunktion(LPVOID parameter){                  //Thread-Funktion
4      meinParameter* para = (meinParameter*)parameter;         //Wiederherstellen des ThreadParameters
5      ...                                                      //Operationen des Threads
6      return(0);                                               //Rückgabe bei erfolgreichem Ende
7  }
8
9  int main(int argc, char * argv[]){                           //Hauptprozess
10     meinParameter* para;                                     //Parameter für die Threads
11     HANDLE* ausgeführteThreads = new HANDLE[AnzahlThreads];  //Eigene Thread-Verwaltung
12     CWinThread* thread;                                      //Thread-Objekt (Verwaltung)
13     for(unsigned int i(0); i<AnzahlThreads; i++){
14         thread = AfxBeginThread(meineThreadFunktion, para); //Starten der Threads
15         ausgeführteThreads[i]=thread->m_hThread;            //Thread zur eigenen Vewaltung hizufügen
16     }
17     ...                                                      //Weitere Operationen des Hauptprozesses
18     WaitForMultipleObjects(AnzahlThreads,                    //Warten bis alle Threads in der
19                            ausgeführteThreads,               //eigenen Verwaltung beendet worden sind
20                            TRUE, INFINITE);
21     return(0);
22 }
```

Abbildung 5-14: Anlegen und Starten von Windows-Threads

5.3.2.6 POSIX-Threads

Bei den POSIX-Threads handelt sich wie im Kapitel 3.3.2.2 auf Seite 21 erwähnt wurde um eine Threadbibliothek. Im Gegensatz zu MFC, welche nur mit einer kostenpflichtigen Version von Microsofts Visual Studio ausgeliefert werden, handelt es sich bei den POSIX-Threads oder auch PThreads um eine OpenSource-Bibliothek. Im Großen und Ganzen ist die Umgehensweise mit den POSIX-Threads jedoch recht ähnlich zu den Windows-Threads. Der Vorteil der PThreads gegenüber den Windows-Threads ist, dass diese sowohl auf Linux- als auch auf Windowssystem ausführbar sind.

Die Abbildung 5-15 zeigt, wie der Basisrahmen der Implementation der PThreads aussieht. Auf den ersten Blick ähnelt die Abbildung unten der Abbildung 5-14 aus Kapitel 5.3.2.5. Der Unterschied wird erst im Detail sichtbar. Die Implementation der PThreads ist, wie schon erwähnt, sehr ähnlich zu der Implementation der Windows-Threads. Daher ist die Anpassung des Quellcodes von einer zur anderen Thread-Bibliothek recht unkompliziert. Die Implementation beginnt auf bei den PThreads mit der Inkludierung der Header-Datei der Bibliothek (Zeile 1). Die Thread-Funktion (Zeile 3-7) kann nahe zu ohne Änderungen von den Windows-Threads übernommen werden. Die drei kleinen Änderungen, die vorgenommen werden müssen, beziehen sich auf die Definition der Funktion und die Rückgabewerte der Funktion. Die Rückgabewerte und der Übergabeparameter der Thread-Funktion sind Datentyp ungebunden, da es sich dabei um „void“-Pointer handelt. Im Gegensatz zu den Windows-Threads ist der „void“-Pointer ein Standardkonstrukt von C++ und somit Plattform unabhängig. Dadurch, dass der Übergabeparameter keinen Datentyp besitzt, muss auch bei den PThreads als Erstes ein casten in den gewünschten Datentyp erfolgen (Zeile 4). Das Gleiche gilt auch für jeden Rückgabewert der Thread-Funktion. Die Rückgabewerte müssen in einen Datentyp unabhängigen „void“-Pointer umgewandelt werden. Zum Starten der PThreads aus einem Hauptprozess heraus wird nicht wie bei den Windows-Threads ein gemeinsames Thread-Objekt für

die Verwaltung genutzt, sondern ein Feld von Thread-Objekten (Zeile 11). Dieses Feld wird so groß definiert, wie die maximale Anzahl an Threads sein darf. Für das eigentliche Starten der verschiedenen Threads eignet sich eine „for“-Schleife am besten (Zeile 12-15). In der Schleife werden mittels der Funktion „pthread_create“ die Threads konfiguriert und gestartet. Beim ersten Parameter der Funktion handelt es sich um einen Pointer auf das entsprechende Thread-Objekt, das für die Verwaltung des Threads zuständig ist. Der zweite Parameter kann mit optionalen Attributen belegt werden. Dabei handelt es sich um Angaben über das Scheduling oder die Priorität des Threads. Der Standardfall beim zweiten Parameter ist, dass der dort übergebene Wert „NULL“ ist. Der dritte Parameter steht für die Funktion, die ein Thread ausführen soll. Der Übergabeparameter, an die vom Thread ausgeführt Funktion, geht als vierter Parameter in die Funktion ein. Anschließend wird an beliebiger Stelle des Hauptthreads gewartet, bis ein bestimmter oder auch alle Threads beendet wurden. Dazu kann die Funktion „pthread_join“ verwendet werden. Diese Funktion wartet so lange, bis ein bestimmter Thread beendet wurde und führt den Thread dann wieder mit dem Hauptprozess zusammen. Wenn auf alle gestarteten Threads gewartet werden soll, ist es lediglich notwendig, durch das Feld von Thread-Objekten zu iterieren. Dazu kann, wie in den Zeilen 17-20 dargestellt, eine „for“-Schleife genutzt werden. Der erste Parameter der Funktion „pthread_join“ steht für den Thread, der mit dem Hauptprozess zusammen geführt werden soll. Der zweite Parameter kann eine Variable enthalten, welche den Rückgabewert der Thread-Funktion aufnimmt. Wird der Rückgabewert nicht benötigt, so ist der zweite Parameter „NULL“.

```
1  #include "pthread.h"                                    //Inkludieren der Klassen (PThreads)
2
3  void* meineThreadFunktion(void* parameter){            //Thread-Funktion
4      meinParameter* para = (meinParameter*)parameter;    //Wiederherstellen des ThreadParameters
5      ...                                                //Operationen des Threads
6      return((void*)returnValue);                        //Rückgabewert der Funktion
7  }
8
9  int main(int argc, char * argv[]){                     //Hauptprozess
10     meinParameter* para;                               //Parameter für die Threads
11     pthread_t hThread[AnzahlThreads];                  //Thread-Objekt (Verwaltung)
12     for(unsigned int i(0); i<AnzahlThreads; i++){
13         pthread_create(&hThread[i], NULL,              //Starten der Threads
14             ThreadFunction, &para[i]);
15     }
16     ...                                                //Weitere Operationen des Hauptprozesses
17     for(unsigned int i =0; i<AnzahlThreads; i++){      //Warten bis alle Threads beendet wurden
18             pthread_join(hThread[i], NULL);            //und wieder in der Hauptprozess
19     }                                                  //eingegliedert sind
20     return(0);
21 }
```

Abbildung 5-15: Anlegen und Starten von POSIX-Threads

5.4 Photo2Scan

Um eine Parallelisierung des Algorithmus vorzunehmen, ist zunächst eine Analyse der Arbeitsweise notwendig. Diese ist im Kapitel 2.3 schon grob vorgenommen worden. Der Algorithmus besteht aus drei Teilen, dabei handelt es sich um einen Initialteil, einen Berechnungsteil und Dateiarbeitsteil. Die Arbeit, die im Zusammenhang mit der eigentlichen Datendatei (Laserscandatei) steht, lässt sich in die zwei Teilbereiche unterteilen. Zum einen ist das, das Lesen von Daten und zum anderen das Schreiben von Daten. Zur Verdeutlichung soll an dieser Stelle noch einmal die Grafik vom Ablauf des Algorithmus, die schon aus Kapitel 2.3 bekannt ist, in leicht veränderter Form dienen. Bei der Abbildung 5-16 handelt es sich um die angesprochene Grafik in veränderter Form. Die drei Teilbereiche des Algorithmus sind in Abbildung farblich markiert. Gelb steht für alle Aufgaben des Initialteils, Blau für alle Aufgaben des Berechnungsteils und Rot für den Dateiarbeitsteil.

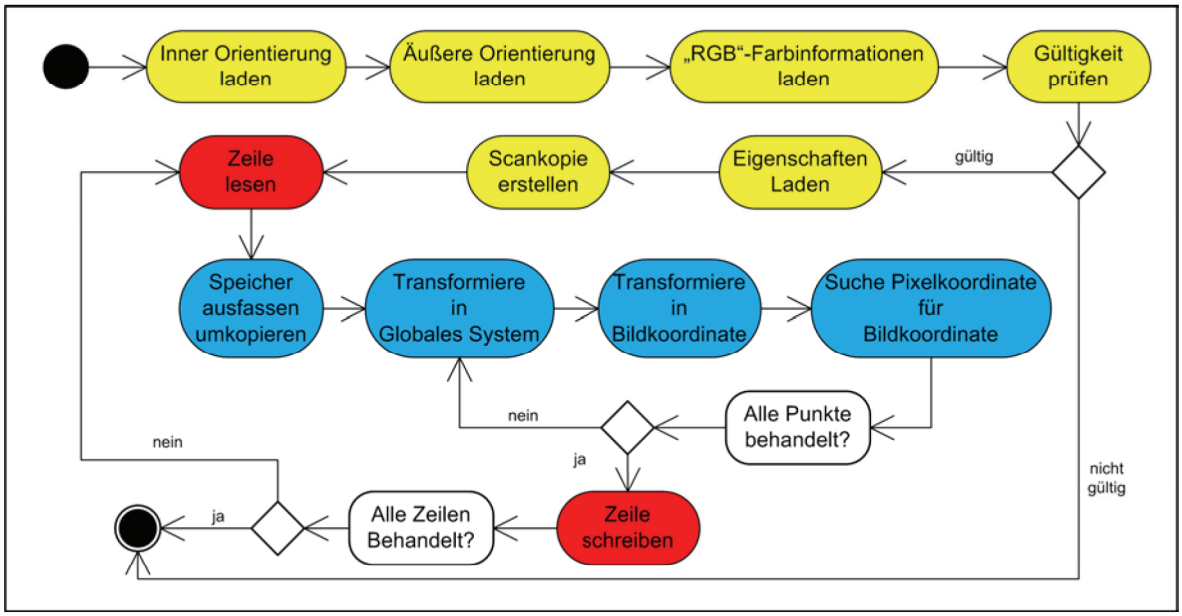

Abbildung 5-16: Photo2Scan: Ablauf und Arbeitsbereiche

Der Initialteil (gelb) befasst sich mit Aufgaben, die sich entweder nicht parallelisieren lassen oder bei denen eine Parallelisierung nicht sinnvoll wäre. Beispielsweise handelt es sich bei den Aufgaben „Gültigkeit prüfen", „Eigenschaften laden" und „Scankopie erstellen" um Aufgaben, die sich nicht parallelisieren lassen. Diese Aufgaben müssen in der richtigen Reihenfolge ausgeführt werden, basieren auf dem jeweiligen Ergebnis des Vorgängers und werden nur einmalig ausgeführt. Diese Eigenschaften der Aufgaben schließen die Verwendung von Threads aus. Auch wenn sich ein Pipeline-Muster anbieten würde, macht die Verwendung bei einer einmaligen Ausführung keinen Sinn. Bei den restlichen drei Aufgaben im Initialteil handelt es sich um Aufgaben, bei denen eine Parallelisierung zwar möglich wäre, jedoch nicht wirklich sinnvoll. Die Aufgaben können zwar sehr gut parallel ablaufen, da sie unabhängig voneinander sind. Jedoch ist die Zeit, die benötigt wird, so gering, dass eine Parallelisierung mehr Overhead erzeugen würde, als dass ein Zeitgewinn entsteht. Die Aufgaben „Innere Orientierung laden", „Äußere Orientierung laden" und „'RGB'-Farbinformationen laden" bieten sich jedoch sehr für eine Parallelisierung mit Threads an. Die drei Aufgaben beziehen sich auf ein Bild, das in einen Scan eingerechnet werden soll. Sofern mehrere Bilder eingerechnet werden, sollten an der Stelle Untersuchungen zu einer Parallelisierung vorgenommen werden. Zu bedenken ist jedoch, dass bei den beiden Aufgaben Daten von Dateien auf der Festplatte gelesen werden und die Hardware Probleme bereiten könnte, weil sie mit dem Lesen von Informationen nicht hinterher kommt. Weiterhin sollte auch berücksichtigt werden, dass die innere Orientierung nur ein einziges Mal für eine Kamera vorgenommen wird. Erst wenn unterschiedliche Kameras zum Einsatz kommen, macht eine Parallelisierung dieser Aufgabe wieder Sinn.

Beim Dateiarbeitsteil (rot) handelt es sich, wie schon erwähnt, um das Lesen und Schreiben von Informationen. Gelesen wird dabei aus der Eingangslaserscandatei jeweils eine Zeile, die dann in den Berechnungsteil weiter geleitet wird. Bei Schreiben werden die Ergebnisse des Berechnungsteils in eine Ergebnislaserscandatei zeilenweise geschrieben. Die beiden Aufgaben werden jeweils zum Anfang und zum Ende der Berechnung einer Zeile durchgeführt. Eine Parallelisierung der einzelnen Aufgaben macht eher wenig Sinn, da die Festplatte nur über einen Lese/Schreib-Kopf verfügt.

Beim Berechnungsteil (blau) ist im Vergleich zum groben Ablauf, dargestellt in Abbildung 2-4, eine Aufgabe hinzugekommen, da der Ablauf in der Abbildung 2-1 detaillierter betrachtet wird. Der

Berechnungsteil besteht aus einer Schleife, in der Berechnungen für alle Punkte einer Zeile der Reihe nach durchgeführt werden. Die Reihenfolge der Ausführung der Aufgaben und die Reihenfolge der Bearbeitung der Zeilen spielt eine wichtige Rolle. Eine korrekte Abarbeitung der Aufgaben ist wichtig, da jeweils die nachfolgende Aufgabe auf der vorherigen Aufgabe beruht. Sollte die korrekte Reihenfolge nicht eingehalten werden, ist das Endergebnis falsch. Das Muss der korrekten Bearbeitung der Zeilen, ist dem Programm zur Darstellung der Laserscans von Lupos3D geschuldet. In dem Programm existieren unterschiedliche Möglichkeiten der Darstellung. Zum einen ist das eine 3D-Ansicht, bei der die Reihenfolge der Zeilen keine Rolle spielt, da die Punkte einer Zeile gemäß ihrer Koordinate dargestellt werden. Zum anderen verfügt das Programm über eine 2D-Ansicht eines Laserscans, dort werden die Zeilen beginnend von der Ersten gelesen und dargestellt. Für das menschliche Auge ergibt sich dabei ein zusammenhängendes und logisches Bild. Falls jedoch die Zeilen nicht in der richtigen Reihenfolge in der Laserscandatei stehen, ergibt sich nur noch ein zusammenhangloses Bild. Die Abbildung 5-17 dient zur Verdeutlichung des beschriebenen Problems.

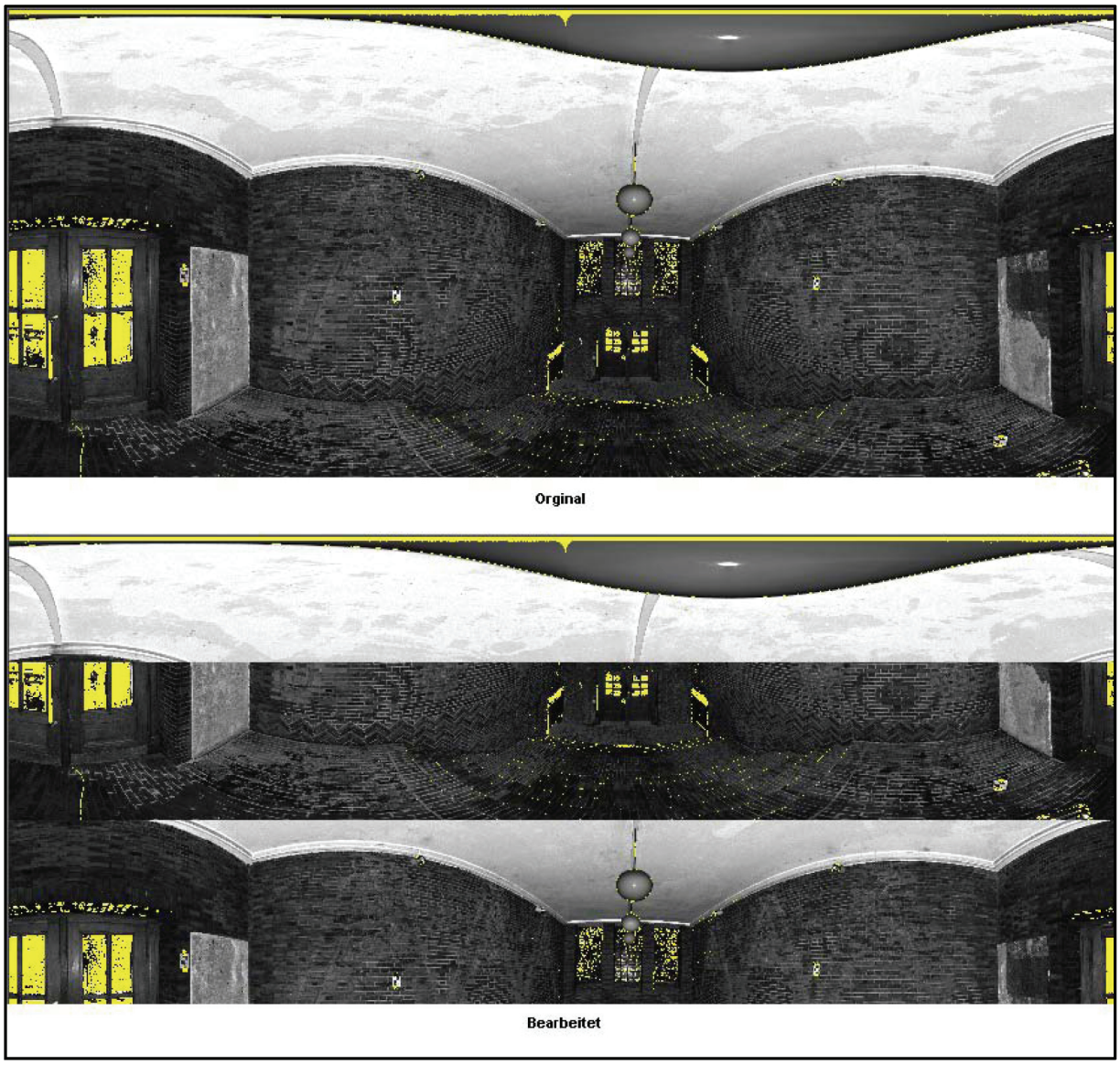

Abbildung 5-17: 2D-Ansicht eines Laserscans

5.4.1 Beschreibung der Teilaufgaben

Die folgenden Kapitel dienen der genaueren Beschreibung der Tätigkeiten der einzelnen Teilaufgaben und der Klärung einer sequenziellen Implementierung. Auf die Implementierung einer Parallelisierung wird in späteren Kapiteln eingegangen.

5.4.1.1 Innere Orientierung

Die Parameter der inneren Orientierung sind, wie schon im Kapitel 2.3 beschrieben, Eigenschaften der Kamera, mit der ein Bild aufgenommen wurde. Sie sind für die Bestimmung der Lage des Projektionszentrums im Bildkoordinatensystem entscheidend.

Die Parameter stehen bei der jetzigen Implementation der Teilaufgabe schon zur Verfügung. Die Bestimmung der inneren Orientierung hatte Lupos3D schon im Vorfeld vorgenommen und in einer Textdatei bereitgestellt. Der Zweck der Teilaufgabe ist es, die Textdatei zu öffnen, die entsprechenden Informationen zu extrahieren und im Programm auf geeignete Variablen zu speichern.

Zum Auslesen der Informationen aus der Textdatei werden verschiedene Techniken benötigt. Zum einen ist das ein „FileStream“, mit dessen Hilfe sich verschiedene Operationen über der Textdatei ausführen lassen. Zum anderen ist das die „String“-Arbeit[46], welche es ermöglicht eine bestimmte Folge von Zeichen zu suchen und zu extrahieren. Um einen „FileStream“ zu erzeugen, wird der Dateipfad benötigt. Dieser muss vom Nutzer beim Start des Programmes angegeben werden. Der „FileStream“ ermöglicht es zu prüfen, wo sich der Zeiger[47] zurzeit befindet und Zeilen aus der Datei einzulesen. In der Abbildung 5-18 ist der Code zu einer Implementierung abgebildet, alle folgenden Zeilenangaben beziehen sich auf diese Abbildung. Für eine String-Suche ist es von entschiedener Wichtigkeit das Format des zu durchsuchenden Strings zu kennen. In diesem Fall ist das Format bekannt und kann auch leicht durch Öffnen der Textdatei überprüft werden. Das Format für die Werte der inneren Orientierung in der Datei besteht aus zwei Komponenten. Einem Teil zur Identifikation eines Wertes und einem Teil mit dem Wert. Getrennt werden die beiden Teile von einem Gleichheitszeichen. Ein Beispiel wäre „meanx_mm=123“. Es werden für die String-Arbeit somit mehrere Variablen benötigt, in der Zeile 2 sind diese zu erkennen. Zunächst wird eine Variable benötigt, die den kompletten String enthält (s), sowie Variablen für den ersten (s_first) und zweiten Teil (s_second) des Strings nach einer Trennung. Die letzte Variable ist die Position (pos) des Trennzeichens, diese wird benötigt um eine korrekte Trennung des Strings durchzuführen. Das eigentliche Extrahieren beginnt mit einer Prüfung der Position des Zeigers, Zeile 5. Wichtig ist, dass der Zeiger nicht am Ende der Datei steht (eof[48]) und dass die Datei geöffnet werden konnte. Würde der Zeiger auf „eof“ stehen, könnte keine Zeile eingelesen werden. Nach der Prüfung kann nun Zeile für Zeile eingelesen und durchsucht werden. In der Zeile 6 wird aus der Datei, die sich hinter der Variablen „file“ befindet, eine Zeile gelesen und auf die Variable „s“ gespeichert. In den Zeilen 7-9 wird die Trennung des Strings in zwei Teile vorgenommen. Zunächst muss dafür die Position des Trennzeichens im String gesucht werden. Mithilfe der Position kann die Trennung vorgenommen werden, Zeile 8-9. Die eigentliche Suche nach dem Wert funktioniert über die Suche nach der Identifikation des Wertes (seinem Namen). Zeile 11-14 zeigt die String-Suche. Der erste Teil des Strings wird nach der Identifikation durchsucht, und wenn der String mit der Suchanfrage übereinstimmt, muss sich der Wert nach Aussage der Formatbeschreibung im zweiten Teil des

[46] Bei einem String handelt es sich um eine Zeichenkette, welche eine Folge von unterschiedlichsten Zeichen enthält.
[47] Der Zeiger gibt an, an welcher Stelle einer Datei aktuell gelesen bzw. geschrieben wird.
[48] eof – **e**nd **o**f **f**ile (deu.: Ende der Datei)

Strings befinden. Der gefundene Wert muss nun auf eine entsprechende Ergebnisvariable gespeichert werden, damit er später zur Verfügung steht.

```
1   fstream file(filename);                          // FileStream erzeugen
2   string s, s_first, s_second; int pos;            // Variablen für die String-Suche anlegen
3   double Result;                                   // Ergebnisvariable anlegen
4
5   if(!file.eof() && !file.fail()){                 // Prüfen wo der Zeiger steht
6       getline(file, s);                            // Eine Zeile lesen
7       pos = s.find("=");                           // Position eines Trennzeichen ermitteln
8       s_first = s.substr(0, pos);                  // Zeile aufteilen in zwei Teile
9       s_second = s.substr(pos+1);                  // Vor und nach dem Trennzeichen
10
11      if(s_first.compare("meanx_mm") == 0){        // Prüfen ob erster Teilstring Wert entspricht
12              istringstream is(s_second);          // Wenn ja, zweiten Teilstring umspeichern
13              is >> Result;                        // auf eine Ergebnisvariable
14       }
```

Abbildung 5-18: Code zum Lesen aus einer Textdatei

5.4.1.2 Äußere Orientierung

Diese Teilaufgabe beruht auf dem gleichen Prinzip wie die innere Orientierung aus Kapitel 5.4.1.1. Eine Erklärung der Vorgehensweise bei der Implementierung erübrigt sich somit. Das Einzige, was die äußere Orientierung von der Inneren unterscheidet, sind die Namen der Parameter für die Identifikation.

Die Teilaufgabe muss für jedes Bild, das benutzt werden soll erneut ausgeführt werden. Jedes Bild verfügt über seine individuelle äußere Orientierung. Die Beispieldaten von Lupos3D umfassen jedoch nur ein Bild und somit auch nur eine Orientierung im Textformat.

Ein Punkt, in dem die Aufgabe der äußeren Orientierung nicht mit der der inneren Orientierung übereinstimmt, ist das bei der Aufgabe der äußeren Orientierung noch die Berechnung einer Rotationsmatrix durchgeführt wird. Die Rotationsmatrix setzt sich drei Drehungen (ω, ϕ, κ) zusammen und beschreibt somit die Ausrichtung des Bildkoordinatensystems zum globalen Koordinatensystem. Die Abbildung 5-19[49] zeigt grafisch die Orientierung eines Bildkoordinatensystems zu einem übergeordneten Koordinatensystem. Die Drehungen finden dabei um die drei Achsen (X, Y, Z) des Koordinatensystems statt. Die eigentliche Rotationsmatrix berechnet sich nach dem Standardfall der Senkrechtaufnahme und ist eine Thematik der Nahbereichsphotgrammetrie. Die Formel 16 zeigt die Berechnung des Standardfalls. Zur Anwendung kommt die Formel 16 später beim Suchen von gleichen Punkten im Laserscan und im Bild.

Formel 16: Berechnung des Standardfalls der Senkrechtaufnahme

$$Rotationsmatrix: R = R_\omega \cdot R_\phi \cdot R_\kappa$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

$$R = \begin{bmatrix} \cos\phi\cos\kappa & -\cos\phi\sin\kappa & \sin\phi \\ \cos\omega\sin\kappa + \sin\omega\sin\phi\cos\kappa & \cos\omega\cos\kappa - \sin\omega\sin\phi\sin\kappa & -\sin\omega\cos\phi \\ \sin\omega\sin\kappa - \cos\omega\sin\phi\cos\kappa & \sin\omega\cos\kappa + \cos\omega\sin\phi\sin\kappa & \cos\omega\cos\phi \end{bmatrix}$$

[49] Nach (2 S. 235)

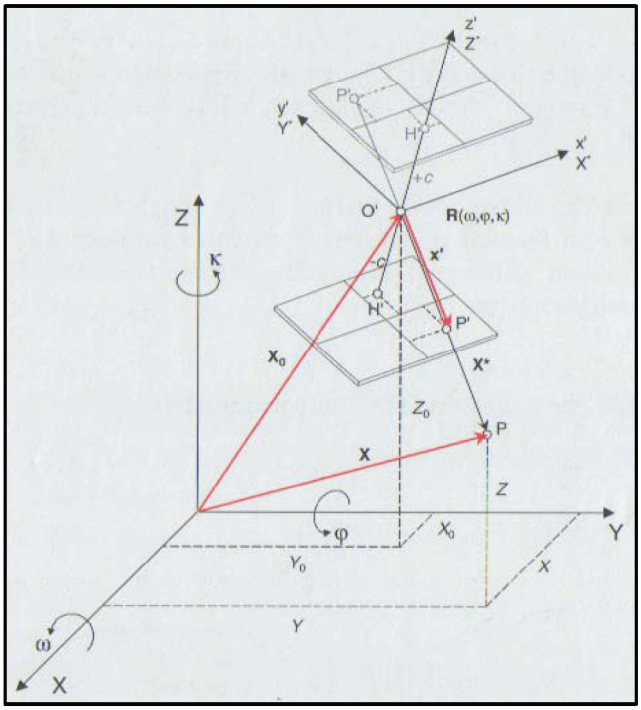

Abbildung 5-19: grafische Darstellung des Standardfalls der Senkrechtaufnahme

Bei einer Implementierung der Berechnung muss die Standardbibliothek „math.h“ eingebunden werden. Diese ermöglicht es verschiedenste mathematische Funktionen auszuführen, Unteranderem auch die benötigten Sinus- und Kosinus-Funktionen. Bei der Implementierung werden die „double“-Funktionen von Sinus und Kosinus angewendet werden, da diese in der Genauigkeit ausreichend sind. Obwohl die Eingangswerte als „float“ vorliegen, ist es notwendig die Berechnungen so genau wie benötigt durchzuführen. Es wird nach der einmaligen Typenkonvertierungen von „float“ nach „double“ jedoch etwas mehr Speicher benötigt. Die Abbildung 5-20 zeigt in zwei Beispielen, wie eine geeignete Ergebnisdatenstruktur aussehen kann und wie auf diese zugegriffen wird. Ebenfalls abgebildet ist die Anwendung der Sinus- und Kosinus-Funktionen im „double“-Format. Die Anzahl der Elemente, die eine Ergebnisstruktur erfassen muss ergibt sich aus der Formel 16. (Kapitel nach (2 S. 37, 235-236))

```
1  /* Beispiel 1 */
2  typedef struct{                                          // Beispiel einer Datenstruktur
3          double r11, r12, r13, r21, r22, r23, r31, r32, r33;     // Implementiert als neuer Datentyp
4  }RotationsMatrix;
5
6  RotationsMatrix RM;                                      // Anlegen der Ergebnisstruktur
7  RM.r11 = sin((double)Drehung);                           // Zugriff auf die Struktur
8      ...                                                  // und Verwendung der Sinus-Funktion
9  RM.r33 = ...;
10
11 /* Beispiel 2 */
12 double[9] RM;                                           // Anlegen der Ergebnisstruktur
13 RM[0] = cos((double)Drehung);                           // Zugriff auf die Struktur
14     ...                                                  // und Verwendung der Kosinus-Funktion
15 RM[8] = ...;
```

Abbildung 5-20: Implementierung einer Ergebnisstruktur und Anwendung der Sinus/Kosinus-Funktionen

5.4.1.3 Farbinformationen

Der Zweck der Teilaufgabe ist die Extraktion der RGB-Farbinformationen aus den Pixeln eines Bildes. Für jedes Pixel sind in einem Farbbild drei Intensitätswerte, jeweils einer für die Farbe Rot, Grün und Blau, hinterlegt. Diese Informationen aus einem Bild herauszulesen ist sehr aufwendig, da jedes Pixel einzeln angefasst werden muss. Genauso umfangreich ist die Datenstruktur, die zum Speichern der

Farbinformationen benötigt wird. Im Internet existiert eine Vielzahl von Bibliotheken, die die Arbeit mit Bildern ermöglichen und vereinfachen. Die ausgesuchte Bibliothek für diese Teilaufgabe nennt sich „bitmap.h“. Es handelt sich dabei um ein Open Source Headerfile, in dem alle von der Aufgabe benötigten Eigenschaften schon implementiert sind. Dadurch, dass die Bibliothek nur als ein Headerfile implementiert ist, lassen sich gegebenenfalls auch Änderungen an den Funktionen vornehmen. Ein kleiner Nachteil der Bibliothek ist, dass sie lediglich Bilder vom Format Bitmap verarbeiten kann. Allerdings liegen die Testdaten im Bitmap-Format vor, weswegen über den Nachteil hinweggesehen werden kann. Ein Vorteil ist, dass es sich bei der Bibliothek um eine sehr leichtgewichtige Implementation handelt.

In der Abbildung 5-21 ist eine Implementation zum korrekten Auslesen der Farbinformationen aus einer Bitmap-Datei, alle folgenden Zeilenangaben beziehen sich auf diese Abbildung. Um mit dem Auslesen der Farbinformationen beginnen zu können, muss zunächst der Pfad zur Bilddatei verfügbar sein. Mit dem Pfad wird ein Objekt vom Typ „CBitmap“ erzeugt (Zeile 1), über diesem Objekt lassen sich im weiteren Verlauf die Funktionen der Bitmap-Bibliothek ausführen. In den nächsten Schritten ist es notwendig bestimmte Informationen aus dem Bild zu extrahieren und Initialisierungen vorzunehmen (Zeile 2-6). Informationen, die aus dem Bild geladen werden müssen, sind die Anzahl der Pixel in der Breite und der Länge des Bildes sowie die Anzahl der Bits pro Pixel. Mit den Informationen über die Breite und Höhe des Bildes lässt sich die Dimension des Bildes bestimmen. In den Zeilen 8-10 werden Felder (Arrays) ausgefasst, die die Ergebnisse der Farbinformationsextraktion aufnehmen. Es existiert jeweils ein Feld pro Farbkanal, um die Verarbeitung zu erleichtern. Die Größe des Feldes ist identisch mit der Dimension des Bildes, da jedes Pixel jeweils für jeden Farbkanal einen Wert enthält. In der Zeile 15 wird ein temporäres Feld angelegt, welches alle Farbinformationen aller Pixel enthält. Mit der Funktion in Zeile 16 wird die Bibliothek angewiesen, die Bits des Bildes in das Feld „Bits“ zu speichern. Der Parameter „Size“ gibt an wie viele Pixel in dem Bild vorkommen und der Parameter „PhotoBitCount“ steht für die Anzahl der Bits pro Pixel. In der doppelten „For“-Schleife (Zeile 18-25) werden den die gelesenen Bits umgespeichert auf die Ergebnisfelder des jeweiligen Farbkanals. Bei der Umspeicherung wird die zweidimensionale Struktur des Bildes (Zeilen und Spalten), konvertiert in eine eindimensionale Struktur. Der Grund für die Konvertierung ist, dass in C++ keine mehrdimensionalen Felder existieren. Es muss also ein Trick angewendet werden, um diese Strukturen trotzdem abbilden zu können.

Um zweidimensionale Felder in C++ zu erstellen, existieren zwei Möglichkeiten. Erstens ist das, das Abbilden des Feldes mit Pointern. Dabei wird ein Pointer Feld angelegt und hinter jedem der Elemente dieses Feldes verbirgt sich wiederum ein Pointer. Diese Methode mag die einfachste sein, aber sie ist bei Weitem nicht die Effizienteste. Der Nachtteil bei Abbildung in ein Pointer Feld ist, das es eine sehr schlechte Speicherausnutzung hat. Beispielsweise wird immer ein symmetrischer Speicher ausgefasst das heißt, wenn ein Pointer in einer Zeile nicht den kompletten Speicher nutzt, ist dieser trotzdem reserviert worden. Bei einem Bild ist das nicht so schlimm, da die Breite des Bildes und somit die Elemente im Pointer immer gleich sind. Ein weiterer Punkt zu Speicherineffizienz ist, dass Pointer neben dem eigentlichen Speicher für die Daten auch Speicher für die Verwaltung benötigen. Bei vielen Pointern steigt somit die Anzahl des Verwaltungsaufwands enorm an. Ein weiterer Nachteil ist die Geschwindigkeit, denn mehrere Pointer zu dereferenzieren lässt sich vom Compiler nicht optimieren und führt somit zu einer Laufzeitverlängerung. Die Abbildung 5-22-a zeigt grafisch die Abbildung einer solchen Speicherstruktur. Die zweite Möglichkeit der Abbildung ist das Umstrukturieren, dabei die zweidimensionale Struktur des Bildes in einem einzeiligen Feld abgebildet. Mit dieser Methode wird auch sequenzielle Allokation genannt. Durch die linear im Speicher abgelegten Elemente kann durch eine einfache Addition und Multiplikation auf ein

gesuchtes Element zu gegriffen werden. Der Index eines gesuchten Elements ergibt sich aus der Formel 17. Die Abbildung 5-23 zeigt die Ermittlung des Index eines Elementes in grafischer Form. Die Vorteile der sequenziellen Allokation liegen in dem geringeren Verwaltungsaufwand, da nur ein Pointer benötigt wird. Ein kleiner Nachteil ist jedoch der Zugriff auf ein gesuchtes Element, dieser wirkt recht kompliziert. Bietet jedoch auch den Vorteil, dass der Compiler bei der Berechnung des Index gegebenenfalls noch Optimierungen durchführen kann. (Abschnitt nach (31))

Für die Abbildung des Bildes im Speicher war die sequenzielle Allokation die erste Wahl. Die Daten (Laserscan) die verarbeitet werden müssen benötigen schon genug Speicher, die kleinste Einsparung ist somit wichtig.

Formel 17: Index eines Elementes im 2D Feld

$$Index = (Anzahl\ der\ Elemente\ pro\ Zeile \cdot gesuchte\ Zeile) + gesuchtes\ Element$$

```
1  CBitmap Photo(filename);                                          // Anlegen eines Bitmap-Objekts
2  unsigned int Rwidth = Photo.GetWidth();                           // Auslesen der Breite des Bildes
3  unsigned int Rheight = Photo.GetHeight();                         // Auslesen der Höhe des Bildes
4  unsigned int PhotoBitCount = Photo.GetBitCount();                 // Auslesen der Bits pro Pixel
5  unsigned int Size = (Rwidth*Rheight);                             // Berechen der Dimension des Bildes
6  unsigned char* Bits;                                              // Variable für die Farbinformation
7
8  int* Rred = new int[Size];                                        // Ergebnis Arrays
9  int* Rgreen = new int[Size];                                      // Eins pro Farbe
10 int* Rblue = new int[Size];                                       // Von der Größe der Dimension des Bildse
11
12 /* Definieren des Arrays für die Farbinformationen */
13 /* in der Dimension des Bildes mal */
14 /* der Anzahl der Werte pro Pixel */
15 Bits = new unsigned char[Size*(PhotoBitCount/8)];
16 Photo.GetBits((void*)Bits, Size, PhotoBitCount);                  // Farbinformation aus dem Bild laden
17
18 for (unsigned int i(0), m(0); i < Rheight; i++){                  // Umspeichern der gesamten Farbinformationen
19   for(unsigned int k(0); k < Rwidth; k++){                       // Auf die Farbarrays
20        Rblue[Rwidth*((Rheight-1)-i)+k] = 255-((int)Bits[m*(PhotoBitCount/8)]);     // Abbilden mehrere Zeilen im Bild
21        Rgreen[Rwidth*((Rheight-1)-i)+k] = 255-((int)Bits[m*(PhotoBitCount/8)+1]);  // auf eine Zeile im Array
22        Rred[Rwidth*((Rheight-1)-i)+k] = 255-((int)Bits[m*(PhotoBitCount/8)+2]);    // Farben invertieren um korrekten Wert
23        m++;                                                      // zu erhalten
24   }
25 }
```

Abbildung 5-21: Code zum Auslesen von Bildinformationen

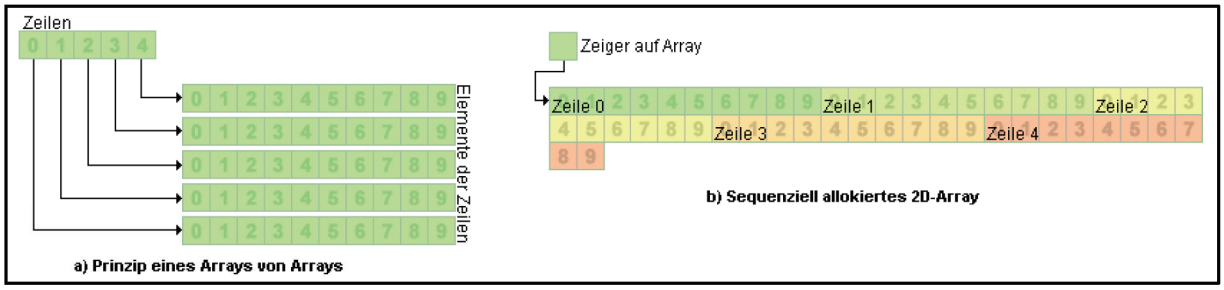

Abbildung 5-22: Bildung 2 dimensionaler Felder in C++

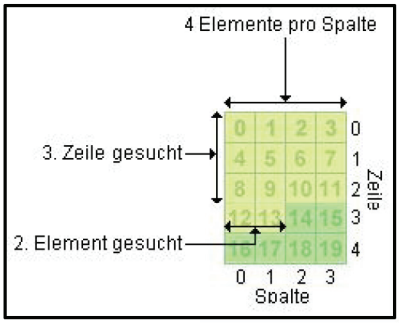

Abbildung 5-23: Adressierung eines Elementes

5.4.1.4 Speicher

Der Existenzsinn der Aufgabe „Speicher ausfassen umkopieren“ liegt in der Datengrundlage, siehe Kapitel 2.1. Der Eingangslaserscan ist vom Typ1, während der Ergebnislaserscan von Typ3 ist. Im Typ1 besteht eine Zeile des Scans lediglich aus vier Werten, X, Y, Z und einem Intensitätswert. Der Typ3 hingegen besteht aus sieben Werten, X, Y, Z, Intensitätswert, R, G, B. Das bedeutet, damit eine Zeile in der Berechnung korrekt verarbeitet werden kann, muss der Speicher den eine Zeile benötigt erweitert werden. Dafür kann jedoch nicht einfach mehr Speicher zugewiesen werden, da sonst die Struktur der Zeile zerstört werden würde. Es wird daher neuer Speicher ausgefasst und jedes Element der originalen Zeile an die korrekte Stelle der neuen Zeile kopiert. Der neue „leere“ Speicher kann mit Defaultwert „-1“ aufgefüllt, damit sich mögliche Fehler leichter erkennen lassen. Der neue „leere“ Speicher lässt sich ebenso für die temporäre Speicherung von Ergebnissen nutzen, da bei späteren Transformationen immer zu den drei original Werten drei transformierte Werte mitgeführt werden müssen.

5.4.1.5 Transformation in globales System

Die Aufgabe dient der Anpassung der, vom Laserscanner, aufgenommenen Punkte an ein globales/übergeordnetes Koordinatensystem. Der Zweck ist, dass so die Koordinaten leichter in den Zusammenhang mit anderen Scans, Karten oder Fotos gebracht werden können.
Bei dieser Teilaufgabe wird auf die Beschreibung in Kapitel 5.3.1.3 verwiesen. Beide Teilaufgaben sind für beide Algorithmen notwendig und gleichen sich zu einhundert Prozent.

5.4.1.6 Ermitteln der Bildkoordinaten

Die Aufgabe beschäftigt sich mit der Suche nach identischen Punkten im transformierten Laserscan und im Bild. Um festzustellen ob der, vom Laserscanner aufgenommene und ins globale System transformierte, Punkt auch im Bild auffindbar ist, ist es notwendig eine photogrammetrische Formel anzuwenden. Bei der Formel handelt es sich um die sogenannte Kollinearitätsbeziehung. Sie ermöglicht es, einen Zusammenhang zwischen dem Bild- und dem globalen Koordinatensystem herzustellen. Mit der Kollinearitätsbeziehung ist möglich, den transformierten Objektpunkt in das Koordinatensystem des Bildes umzurechnen. Dazu werden neben den Koordinaten des Objektpunktes auch noch die Parameter der inneren und äußeren Orientierung sowie die in Kapitel 5.4.1.2 berechnete Rotationsmatrix benötigt. Somit ergibt sich für die Kollinearitätsbeziehung die Formel 18:

Formel 18: Kollinearitätsgleichung

$$Koordinaten\ im\ Bild: x';\ y'$$
$$Koordinaten\ des\ Objektpunktes: X; Y; Z$$
$$Rotationsmatrix\ R: \text{Formel 16}\ auf\ Seite\ 51$$
$$Parameter\ der\ äußeren\ Orientierung: X_0;\ Y_0;\ Z_0;\ \omega;\ \phi;\ \kappa$$
$$Parameter\ der\ inneren\ Orientierung: x'_0;\ y'_0;\ c;\ \Delta x';\ \Delta y'$$

$$x' = x'_0 - c \cdot \frac{r_{11} \cdot (X - X_0) + r_{21} \cdot (Y - Y_0) + r_{31} \cdot (Z - Z_0)}{r_{13} \cdot (X - X_0) + r_{23} \cdot (Y - Y_0) + r_{33} \cdot (Z - Z_0)}$$

$$y' = y'_0 - c \cdot \frac{r_{12} \cdot (X - X_0) + r_{22} \cdot (Y - Y_0) + r_{32} \cdot (Z - Z_0)}{r_{13} \cdot (X - X_0) + r_{23} \cdot (Y - Y_0) + r_{33} \cdot (Z - Z_0)}$$

Diese Berechnungen werden für jeden Punkt im Laserscan durchgeführt und es ergibt sich auch für jeden Punkt eine Koordinate im Bildkoordinatensystem. Da ein Bild jedoch nur einen Ausschnitt des gesamten Laserscans abdeckt, ist es also unmöglich das alle berechneten Koordinaten korrekt sind. Ein Teil der berechneten Koordinaten liegt außerhalb des vom Bild abgedeckten Bereichs und ist somit ungültig. An dieser Stelle kommt die letzte Aufgabe des Berechnungsteils zum Einsatz. (Kapitel nach (2 S. 237-238; 32 S. 270-272))

5.4.1.7 Ermitteln der Pixelkoordinaten

Die letzte Teilaufgabe, der Aufgaben des Berechnungsteils des Algorithmus, beschäftigt sich mit der Prüfung der in Kapitel 5.4.1.6 errechneten Bildkoordinaten und dem Auslesen der entsprechenden „RGB“-Farbinformationen aus den Ergebnisfeldern, siehe Kapitel 5.4.1.3.

Für jede mittels der Kollinearitätsgleichung ermittelte Bildkoordinate wird eine Indexposition in den Farbkanalfeldern berechnet. Die Berechnung basiert auf einer einfachen geometrischen Tatsache, die in Abbildung 5-24 dargestellt ist. Aus der grafischen Darstellung lässt sich eine Formel ableiten, mit der die Indexposition einer Bildkoordinate in den Farbkanalfeldern berechnet werden kann, siehe Formel 20. Der erste Schritt der Berechnung ist die Bildung des Verhältnisses zwischen dem Bildkoordinatensystem und dem Index der Farbkanalfelder, siehe Formel 19. Mit Auflösung des Verhältnisses nach der gesuchten Indexposition ergibt sich ein Wert, der noch in den korrekten Ursprung verschoben werden muss. Um auf eine Indexposition zu ermitteln, muss also beachtet werden, dass der Ursprung des Bildkoordinatensystems nicht mit dem Ursprung der Farbkanalfelder übereinstimmt. Bei den Farbkanalfeldern befindet sich der Ursprung an der Position nullte Zeile/nullte Spalte. Der Ursprung im Bildkoordinatensystem liegt genau in der Mitte der Matrix der Farbkanalfelder. Es muss also im zweiten Schritt die Verschiebung des Ursprungs berücksichtigt

werden, das ist jeweils der erste Teil der Formel 20. Mit der Verbindung der Formel 19 und der Verschiebung des Ursprungs kann die korrekte Indexposition berechnet werden.

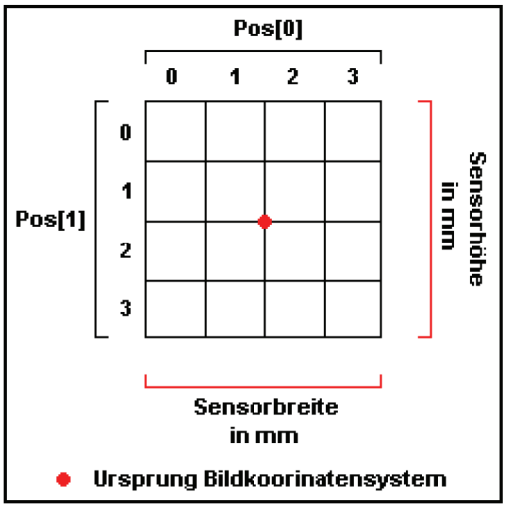

Abbildung 5-24: grafische Darstellung der Ermittlung der Pixelkoordinate

Formel 19: Verhältnis zwischen Bildkoordinatensystem und Indexposition

$$\frac{Bildkoordinate_X}{SensorBreite_{mm}} = \frac{Pos[0]}{BreiteBild_{Pixel}} \rightarrow Pos[0] = \frac{Bildkoordinate_X \cdot BreiteBild_{Pixel}}{SensorBreite_{mm}}$$

$$\frac{Bildkoordinate_Y}{SensorHöhe_{mm}} = \frac{Pos[1]}{HöheBild_{Pixel}} \rightarrow Pos[1] = \frac{Bildkoordinate_Y \cdot HöheBild_{Pixel}}{SensorHöhe_{mm}}$$

Formel 20: Berechnung der Indexposition

$$Spaltenindex: Pos[0] = \frac{BreiteBild_{Pixel}}{2} + \left(\frac{Bildkoordinate_X \cdot BreiteBild_{Pixel}}{SensorBreite_{mm}}\right)$$

$$Zeilenindex: Pos[1] = \frac{HöheBild_{Pixel}}{2} - \left(\frac{Bildkoordinate_Y \cdot HöheBild_{Pixel}}{SensorHöhe_{mm}}\right)$$

Das anschließende Prüfen ist notwendig, da jeder Punkt des Laserscans mittels der Kollinearitätsgleichung eine Koordinate im Bildkoordinatensystem zugewiesen bekommt und für jede Koordinate auch eine Indexposition bestimmt werden kann. Die Indexpositionen, die außerhalb der Dimension der Farbkanalfelder liegen, sind ungültig. Mit ungültigen Positionen darf nicht auf die Farbkanalfelder zugegriffen werden, das würde einen schweren Fehler im Programm erzeugen und dieses zum Absturz führen. Daher werden alle Indexpositionen nicht berücksichtigt, die negativ sind oder deren Dimension die maximale Breite bzw. Höhe des Bildes in Pixel übersteigt. Die Formel 21 zeigt die eben erwähnten Bedingungen noch einmal in mathematischer Form. War die Prüfung erfolgreich, wird auf die drei Farbkanalfelder mit der berechneten Indexposition zugegriffen und die entsprechen Farbintensitäten, auf dem in Kapitel 5.4.1.4 ausgefassten Speicher, zu dem Laserscanpunkt gespeichert. Sollte die Prüfung nicht erfolgreich gewesen sein, wird der mit Defaultwerten beschrieben und im Kapitel 5.4.1.4 ausgefasst Speicher mit dem Intensitätswert des Laserscanpunktes beschrieben. Dadurch ergibt sich in der Darstellung ein graues Bild, in dem ein Farbbild auftaucht. Eine Ansicht des Ergebnisses ist in der Abbildung 5-25 dargestellt.

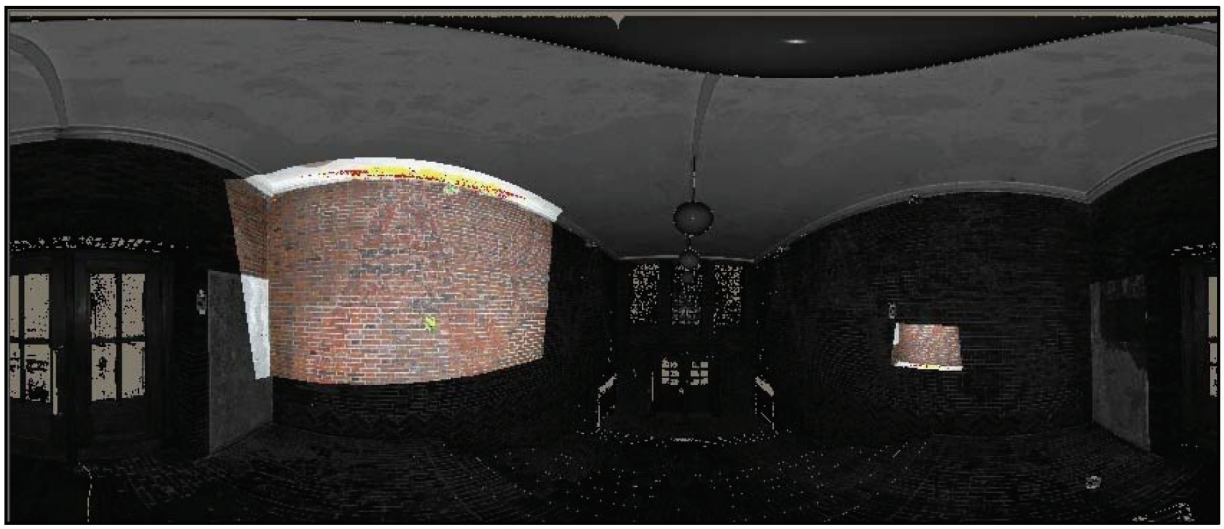

Abbildung 5-25: Beispielergebnis des Berechnungsteils

Formel 21: Prüfbedingungen für die Indexpositionen

$$Spaltenindex: 0 \leq Pos[0] < BreiteBild_{Pixel}$$

$$Zeilenindex: 0 \leq Pos[1] < HöheBild_{Pixel}$$

5.4.1.8 Lesen / Schreiben

Beim Lesen von Daten aus einer Laserscandatei werden die Funktionen verwendet, die Lupos3D zur Verfügung gestellt hat. Das Lesen erfolgt dabei auf die gleiche Art und Weise, wie es in Kapitel 5.3.1.4 schon beschrieben wurde.

Zum Schreiben der Daten werden bei dieser Teilaufgabe ebenfalls die Funktionen von Lupos3D verwendet. Der Grund ist, dass die Aufgabe eine Veränderung der originalen Laserscandatei vornimmt. Damit die Originaldatei nicht beschädigt wird, werden Veränderungen in einer neuen Laserscandatei gespeichert. Die Schritte, die notwendig sind, um so eine bearbeitete Kopie zu erstellen, sind relativ simpel und in der Abbildung 5-26 dargestellt, alle folgenden Zeilenangaben beziehen sich auf diese Abbildung. Das Einbinden der Headerdatei von Lupos3D, stellt die benötigten Funktionen für die Arbeit mit Dateien im „ptb“-Format zur Verfügung (Zeile 1). Der zweite Schritt ist das Öffnen einer Laserscandatei zum Lesen, wie es schon bekannt ist (Zeile2-8). Der eigentliche Schreibteil beginnt in der Zeile 12, dort wird ein Objekt vom Typ „MFFPTB“ angelegt. Dieses Mal wird jedoch ein anderer Konstruktor als beim Lesen verwendet. Der neue Konstruktor legt eine neue Laserscandatei an. Als Parameter werden Konstruktor die Eigenschaften der originalen Laserscandatei übergeben (Anlegen einer Kopie). Die einzige Ausnahme bildet das sogenannte „colorFlag“, der vierte Parameter des Konstruktors. Wie schon in Kapitel 5.4.1.4 und 5.4.1.7 beschrieben, wird die neue Scandatei um „RGB“-Farbinformationen erweitert. Das bedeutet, dass das „colorFlag“ aus der originalen Datei nicht übernommen werden darf. Zum originalen „colorFlag“ wird einfach eine „2“ hinzuaddiert, so ergibt sich das neue benötigte „colorFlag“. Der Wert „2“ ergibt sich aus den Eigenschaften der Typen der Laserscans, wie sie in Kapitel 2.1 beschrieben sind. Beispielweise ist der originale Laserscan vom Typ1, er besitzt also die Werte X, Y, Z, Intensität, durch Hinzufügen von „RGB“-Farbinformationen ergibt sich der Typ3 (X, Y, Z, Intensität, R, G, B). Das eigentliche Schreiben einer Zeile, von Punkten (X, Y, Z) mit den Zusatzinformationen, findet in der Zeile 14 der Abbildung 5-26 statt. Der Parameter der dort verwendeten Funktion ist ein Pointer auf ein Feld (Array) von „float“-Werten. Die Werte des Feldes werden im binären Format in die

Laserscandatei geschrieben. Deswegen ist es notwendig, dass die Struktur des Feldes mit der, bei der Erstellung der Laserscandatei angegebenen, Struktur der Scandatei („colorFlag“) übereinstimmt. Bei einem Fehler in der Struktur wird die Laserscandatei beschädigt.

```
#include "mffptb.h"                                                    // Einbinden Lupos3D Funktionen
MFFPTB ptb_org_file(In_File_Path);                                     // Scandatei zum Lesen öffnen
if(ptb_org_file.isValid()){                                            // Gültigkeit prüfen
    unsigned int height      = ptb_org_file.height();                  // Header-Informationen lesen
    unsigned int width       = ptb_org_file.width();                   //          ...
    unsigned int valPerPixel = ptb_org_file.valuesPerPixel();          //          ...
    unsigned int type        = ptb_org_file.colorFlag();               //          ...
    double* ptb_ori = ptb_org_file.ori();                              //          ...

    /* Durchführen der Aufgaben "Lesen" und des Berechnungsteils */

    MFFPTB ptb_cpy_file(CPY_File_Path, width, height, type+2, ptb_ori);    // Kopie von der Scandatei anlegen
    if(ptb_cpy_file.isValid()){                                        // Gültigkeit der neuen Datei prüfen
        ptb_cpy_file.writeRow(ErgebnisZeile);                          // Eine Zeile in die Datei schreiben
    }
}
```

Abbildung 5-26: Schreiben einer Scandatei mit Lupos3D-Funktionen

5.4.2 Parallelisierung

Neben einer sequenziellen Implementierung des Photo2Scan-Algotrithmus, wurde auch eine Möglichkeit zur Parallelisierung (mit zwei Ansätzen) implementiert. Während bei der sequenziellen Implementation die Operationen der Reihe nach ablaufen, wie in der Abbildung 5-16 dargestellt, verwenden die Versuche der Parallelisierung das Entwurfsmuster „Chain of Responsibility“ oder auch Pipeline genannt. Wie schon erwähnt wurde die Implementierung der Pipeline auf zwei Arten vorgenommen zum einen, eine sequenzielle Pipeline und zum Anderen, eine parallele Pipeline. Das Prinzip einer Pipeline wurde schon im Kapitel 3.2.2.4 auf Seite 17 genauer erläutert. An dieser Stelle sei nur noch einmal erwähnt, dass alle Aufgaben des Berechnungsteils nacheinander ablaufen und die nachfolgende Aufgabe auf dem Ergebnis der vorherigen beruht.

5.4.2.1 Sequenzielle Pipeline

Bei dieser Art der Implementierung wird lediglich das Entwurfsmuster der Pipeline implementiert, es werden keine weiteren Parallelisierungsmethoden eingesetzt. Der Sinn in der Vorgehensweise liegt darin, dass das Betriebssystem vielleicht die Verwaltung des Programmes so organisiert, dass schon die Verwendung eines parallelen Entwurfsmusters die Bearbeitung beschleunigen kann.

Für die Erstellung einer sequenziellen Pipeline musste zunächst untersucht werden, wie eine solche Implementierung aussehen könnte. Dazu wurde gerade der Berechnungsteil immer wieder betrachtet. Während der Entwicklung einer geeigneten Pipeline wurden unterschiedliche UML-Diagramme erstellt. Die Pipeline wurde von einer sehr speziellen Lösung, die genau auf den Algorithmus zugeschnitten war, immer weiter verallgemeinert und optimiert. Das Kapitel stellt die letzte Version (Abbildung 5-27) der entwickelten Pipeline genauer vor und geht an geeigneten Stellen auf vorherige Versionen ein.

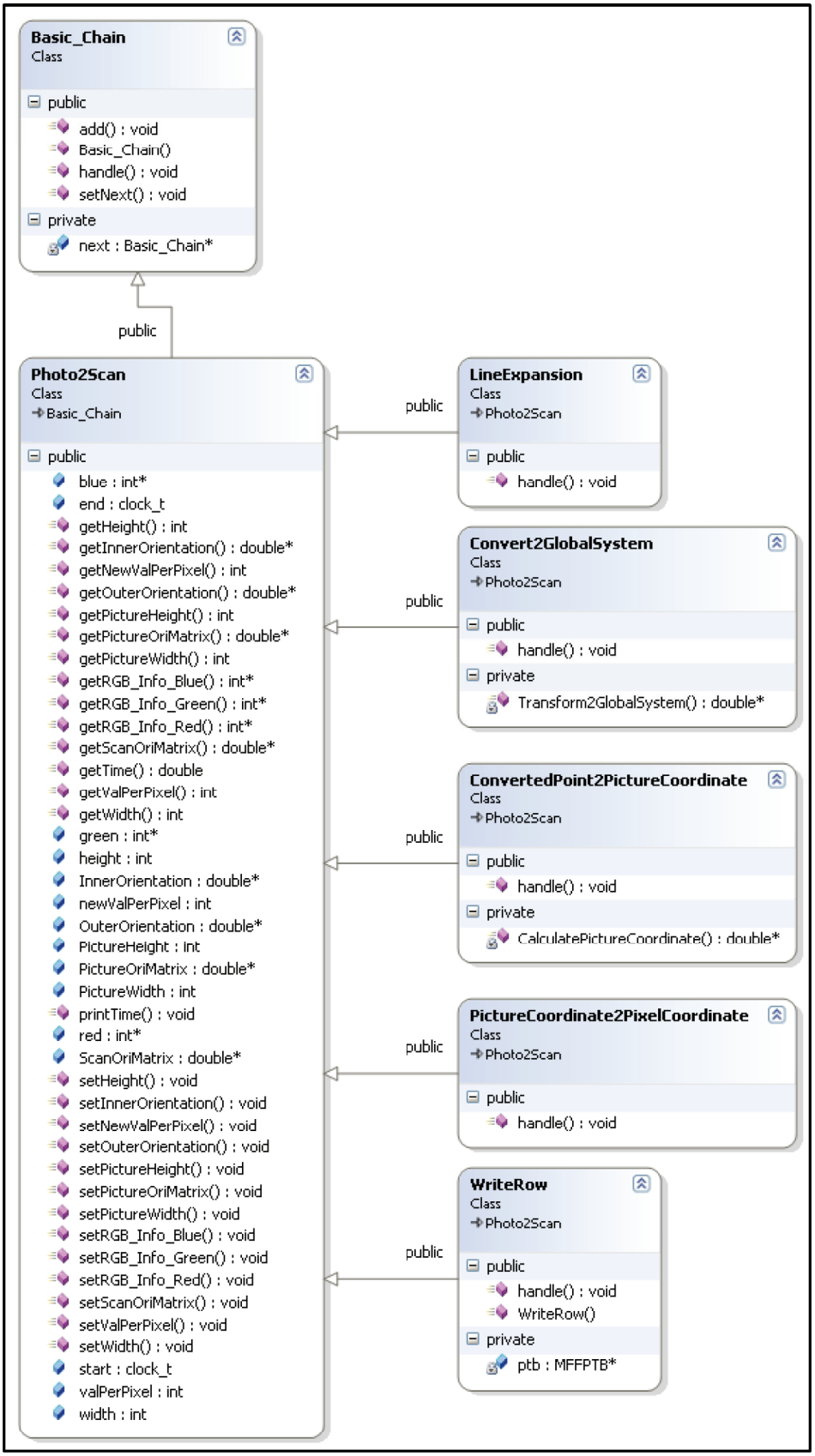

Abbildung 5-27: aktuelle Pipelineversion

Zu erkennen sind in der Abbildung 5-27 sieben unterschiedliche Klassen, die alle aufeinander aufbauen. Die Erläuterung der Struktur der Pipeline beginnt mit der allgemeinsten Klasse und endet bei den spezialisiertesten Klassen. Die Klasse „Basic_Chain“ ist für die Infrastruktur der Pipeline

verantwortlich, sie bildet das Interface, mit dem Pipelines erstellt werden können. In dieser Klasse sind Funktionen und Variablen definiert, die für den Aufbau und die Steuerung der Pipeline notwendig sind. Mit dem Aufruf des Konstruktors der Klasse wird eine neue Instanz einer Aufgabe in der Pipeline angelegt. Diese neue Aufgabe hat bei dem Zeitpunkt der Erstellung allerdings noch keinen Nachfolger. Somit können mehrere Instanzen der Klasse „Basic_Chain“ existieren, ohne miteinander verknüpft zu sein. Es würden dann mehrere Pipelines mit nur einer einzigen Aufgabe vorhanden sein. Die Verknüpfung der einzelnen Aufgaben miteinander übernehmen die Funktionen „add()“ und „setNext()“. Sie sind zuständig für das Festlegen der nachfolgenden Aufgabe in die Pipeline. Beim Aufruf der „add()“-Funktion wird die neue hinzuzufügende Aufgabe automatisch an das Ende der Pipeline gehängt, siehe Abbildung 5-28. Während mit der „setNext()“-Funktion direkt die Aufgabe an eine bestimmte Stelle der Pipeline platziert werden kann. Falls dort schon eine Verbindung zu einer nachfolgenden Aufgabe besteht, wird diese getrennt und durch die neue Verbindung ersetzt.

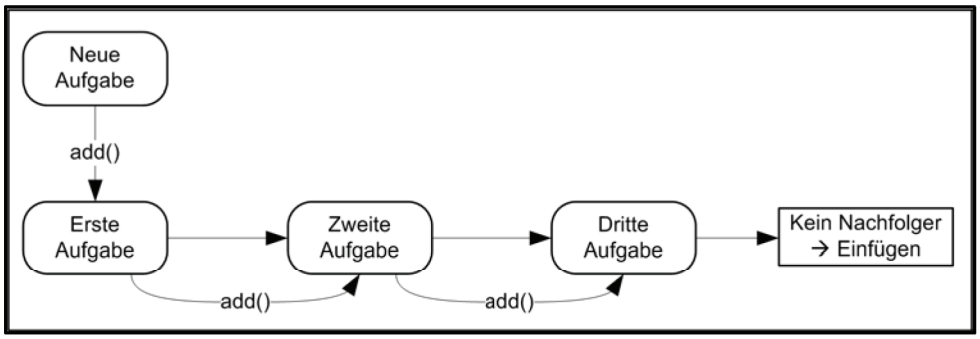

Abbildung 5-28: Einfügen einer neuen Aufgabe in die Pipeline

Die Funktion „handle()“ ist eine virtuelle Funktion das heißt, sie ist abstrakt und kann überschrieben werden. Der Zweck der Funktion ist die Weiterleitung eines Wertes an die nachfolgende Aufgabe. Gibt es keinen Nachfolger, ist die Pipeline beendet. Die „handle()“-Funktion ist auch die Stelle, an der die eigentlichen Operationen der jeweiligen Aufgabe implementiert werden. Der Vorteil des Interface „Basic_Chain“ ist die starke Verallgemeinerung. Es spielt keine Rolle, welchen Algorithmus eine Pipeline implementieren soll, alle Applikationen können die Eigenschaften des Interfaces erben und damit arbeiten.

Die nun folgenden Klassen erben allesamt die Eigenschaften der Klasse „Basic_Chain“ und sind somit die Kinder der Klasse. Auch sind die Kinderklassen speziell für den Photo2Scan-Algorithmus entworfen und in eine einzige Headerdatei geschrieben. Zwischen den eigentlichen Implementationen der Aufgaben der Pipeline befindet sich noch eine weitere Klasse mit dem Namen „Photo2Scan“. Im Gegensatz zu vorherigen Versionen der Pipeline, implementiert diese Klasse alle Funktion und Variablen, die für alle einzelnen Aufgaben gleich sind. Beispielsweise werden alle notwendigen Variablen und Funktionen für die Zeitmessung in dieser Klasse definiert. So ist es möglich, die Implementation der einzelnen Aufgaben relativ kompakt und übersichtlich zu halten.

Die eigentlichen Implementationen der Aufgaben der Pipeline erben über die Klasse „Photo2Scan“ auch die Eigenschaften des Interfaces „Basic_Chain“. Die Aufgaben, die in der Pipeline durchgeführt werden, sind schon in den Kapiteln 5.4.1.4 bis 5.4.1.8 beschrieben. An dieser Stelle bleiben somit nur noch Ergänzungen vorzunehmen, die sich direkt auf die Implementation dieser Pipeline beziehen. Die „main“-Methode der Applikation übernimmt gewisse Initialarbeiten für die Pipeline. Dazu zählen das Erstellen, Konfigurieren und Verknüpfen der einzelnen Aufgaben sowie das Öffnen der Eingangsscandatei und das Anlegen der Ausgangsscandatei. Ebenfalls übernimmt die

„main“-Methode das Lesen der Zeilen aus der Eingangsscandatei und das Einreichen der Zeile in die Pipeline. Um in der letzten Aufgabe der Pipeline („WriteRow“) korrekt eine Zeile schreiben zu können, wird ein Zeiger auf die in der „main“-Methode erstellte Ausgangsscandatei durchgereicht. Die Abbildung 5-29 zeigt das Erstellen, Konfigurieren und Verknüpfen der Pipelineaufgaben durch die „main“-Methode und die Abbildung 5-30 zeigt, wie eine korrekte Implementation einer Pipelineaufgabe vorzunehmen ist. Zu erkennen ist dabei die Spezialisierung der Klasse gegenüber der Elternklasse (Zeile 1). Die eigentliche Implementierung der Operationen der Aufgabe geschieht durch das Überschreiben der „handle()-Funktion des Interfaces „Basic_Chain“ (Zeile 3-17). In der Zeile 16 wird das Ergebnis der Aufgabe durch einen direkten Aufruf der „handle()“-Funktion im Interface an die nachfolgende Aufgabe weitergeleitet.

```
1  LineExpansion FirstTask;                        // Anlegen der Pipelineaufgaben
2  Convert2GlobalSystem SecondTask;                //          ...
3  ConvertedPoint2PictureCoordinate ThirdTask;     //          ...
4  PictureCoordinate2PixelCoordinate FourthTask;   //          ...
5  MFFPTB ptb(File, width, height, 3, ptb_ori);    // Erstellen der Ausgangsscandatei
6  WriteRow FifthTask(&ptb);                       // Anlegen der letzten Aufgaben & weiterleiten des Zeigers
7
8  FirstTask.setWidth(width);                      // Konfiguration der einzelnen Aufgaben
9  FirstTask.setValPerPixel(valPerPixel);          //          ...
10 FirstTask.setNewValPerPixel(7);                 //          ...
11
12 SecondTask.setWidth(width);                     //          ...
13 SecondTask.setNewValPerPixel(7);                //          ...
14 SecondTask.setScanOriMatrix(ptb_ori);           //          ...
15
16 ThirdTask.setWidth(width);                      //          ...
17 ThirdTask.setNewValPerPixel(7);                 //          ...
18 ThirdTask.setInnerOrientation(IOO);             //          ...
19 ThirdTask.setOuterOrientation(OO);              //          ...
20 ThirdTask.setPictureOriMatrix(PRM);             //          ...
21
22 FourthTask.setWidth(width);                     //          ...
23 FourthTask.setNewValPerPixel(7);                //          ...
24 FourthTask.setInnerOrientation(IOO);            //          ...
25 FourthTask.setRGB_Info_Red(red);                //          ...
26 FourthTask.setRGB_Info_Green(green);            //          ...
27 FourthTask.setRGB_Info_Blue(blue);              //          ...
28 FourthTask.setPictureHeight(picture_height);    //          ...
29 FourthTask.setPictureWidth(picture_width);      //          ...
30
31 FirstTask.add(&SecondTask);                     // Verknüpfen der Aufgaben
32 FirstTask.add(&ThirdTask);                      //          ...
33 FirstTask.add(&FourthTask);                     //          ...
34 FirstTask.add(&FifthTask);                      //          ...
35
36 /* Beginn des Lesens von Zeilen */
37 float* PointsIn = new float[width * valPerPixel];
38 Time_Init_End = (clock()-Time_Init_Start);
39 for(int i(0); i<height; ++i){                   // Solange lesen bis alle Zeile behandelt wurden
40     TimeRead_Start = clock();
41     ptb_file.readRow(PointsIn, (unsigned int)i, 0, (unsigned int)width);
42     TimeRead_End += (clock()-TimeRead_Start);
43     FirstTask.handle(PointsIn);                 // Pipeline mit der ersten Aufgabe starten
44 }
```

Abbildung 5-29: Erstellen, Konfigurieren und Starten der sequenziellen Pipeline

```
class LineExpansion : public Photo2Scan{ // First Task
    public:
        /*virtual*/ void handle(float* In_Line){
            start = clock();
            float* Line = new float[width*newValPerPixel];
            for(int point_number(0); point_number<width; point_number++){
                Line[point_number*newValPerPixel] = In_Line[point_number*valPerPixel];   //X
                Line[point_number*newValPerPixel+1] = In_Line[point_number*valPerPixel+1]; //Y
                Line[point_number*newValPerPixel+2] = In_Line[point_number*valPerPixel+2]; //Z
                Line[point_number*newValPerPixel+3] = In_Line[point_number*valPerPixel+3]; //Inten
                //Line[point_number*newValPerPixel+4] = (float)(-1); // provisionally default X
                //Line[point_number*newValPerPixel+5] = (float)(-1); // provisionally default Y
                //Line[point_number*newValPerPixel+6] = (float)(-1); // provisionally default Z
            }
            end += (clock()-start);
            Basic_Chain::handle(Line);
        }
};
```

Abbildung 5-30: Beispiel der Implementation einer Pipelineaufgabe

5.4.2.2 Parallele Pipeline

Bei dieser Art der Implementierung wurde auch das Entwurfsmuster der Pipeline verwendet. Der Unterschied zur Pipeline aus dem Kapitel 5.4.2.1 ist, bei dieser Implementierung werden die einzelnen Aufgaben der Pipeline mit unterschiedlichen Threads ausgeführt werden. Um das zu realisieren, wurde die im Kapitel 3.3.4 vorgestellte TBB Bibliothek von Intel genutzt. Die Bibliothek unterstützt das Erstellen einer parallelisierten Pipeline, dabei übernimmt die Bibliothek die Organisation und die Verteilung der Aufgaben auf die verfügbaren Prozessoren.

Die implementierte parallele Pipeline besteht aus den gleichen Aufgaben wie die sequenzielle Pipeline. Ein Unterschied ist nur, dass das Lesen einer Zeile aus der Eingangsscandatei bei dieser Implementation ebenfalls eine Pipelineaufgabe geworden ist. Das hat den Vorteil, dass das Lesen jetzt möglicherweise mit parallelisiert wird. So kann, während eine Zeile verarbeitet wird, schon eine neue von der Festplatte gelesen werden.

Für die Implementation einer Pipelineaufgabe mittels der TBB-Bibliothek, müssen alle Aufgaben die Basis-Struktur einer entsprechenden TBB-Klasse erben. Diese Klasse nennt sich „filter“ und stammt aus dem Namespace „tbb“. Die Abbildung 5-31 zeigt die neue Struktur der Pipeline bei der Verwendung der TBB-Bibliothek. Die Klasse „filter“ der Bibliothek übernimmt somit die Funktionalität der „Basic_Chain“ Klasse der sequenziellen Pipeline. Die Klassen der einzelnen Aufgaben haben auch die gleiche Bezeichnung, wie bei der sequenziellen Pipeline. Die einzigen Ausnahmen bilden die Klassen für das Lesen und Schreiben. Diese wurden umbenannt in „MyInputFilter“ für das Lesen und „MyOutputFilter“ für das Schreiben. Dadurch verdeutlicht sich welche Aufgabe der Einstiegs- und welche der Endpunkt der Pipeline ist. Wie in der Struktur zu erkennen ist, überschreibt jede Pipelineaufgabe die „operator()()“-Funktion der „filter“-Klasse. Bei der überschriebenen Funktion handelt es sich um die Funktion, die für das Weiterleiten des Ergebnisses einer Aufgabe zur Nächsten zuständig ist. Die „operator()()“-Funktion nimmt in der parallelen Pipeline die gleiche Rolle ein, wie die „handle“-Funktion bei der sequenziellen Pipeline.

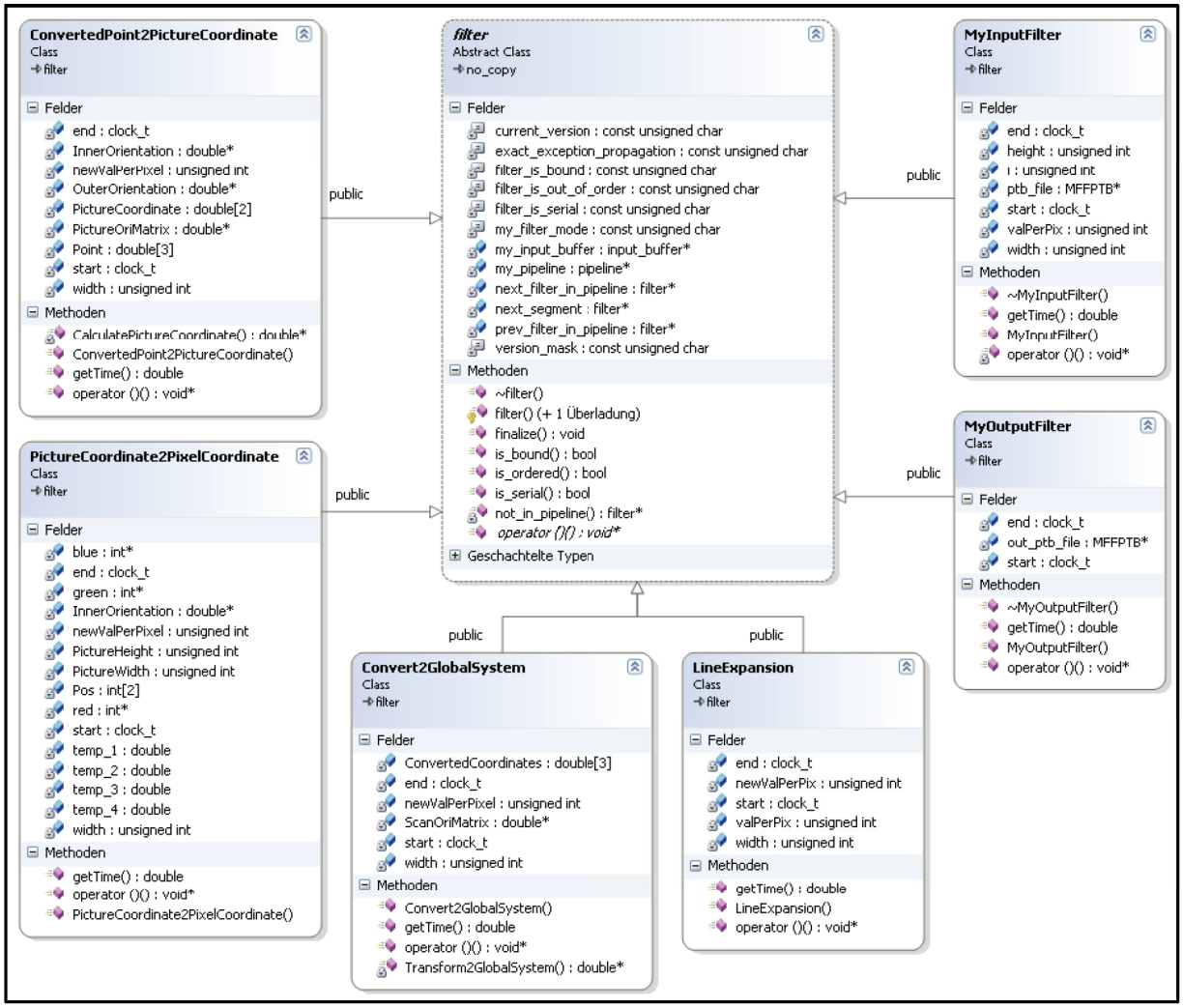

Abbildung 5-31: Struktur der parallelen Pipeline

Grob betrachte ist kein großer Unterschied in der Struktur zwischen der parallelen und der sequenziellen Pipeline zu erkennen. Erste bei einem genaueren Blick auf die Implementation im Quelltext sind gravierende Unterschiede zu erkennen. Die Abbildung 5-32 zeigt die Implementation einer Pipelineaufgabe bei der Verwendung der TBB-Bibliothek, am selben Beispiel wie in der Abbildung 5-30. Die Aufgabe besteht aus zwei Teilen, einem Deklarationsteil (Zeile 1-12) und einem Implementationsteil (Zeile13-34). In der Zeile 1 ist zu erkennen, dass die Klasse die Eigenschaften der „filter“-Klasse erbt. Die Zeile 6 beschreibt den Konstruktor, der bei der Erzeugung einer Instanz der Klasse aufgerufen wird. Dieser verlangt verschiedene Parameter, die die Klasse zum Arbeiten benötigt. Bei der sequenziellen Pipeline wurde das mit sogenannten „setter“-Methoden implementiert. In der Zeile 7 wird schon bei der Deklaration angekündigt, dass die „operator()()“-Funktion der „filter“-Klasse überschrieben wird. Im Implementationsteil wird als Erstes der Konstruktor definiert und die übergeben Werte auf Klassen interne Variablen umgespeichert. Die Zeile 14 zeigt ein besonderes Merkmal der TBB-Bibliothek. Es ist in der Bibliothek möglich Aufgaben mit einem unterschiedlichen Status zu versehen. Beispielsweise kann durch das Angeben des Status „serial_in_order“ der Pipeline gesagt werden, das Aufgaben in einer bestimmten Reihenfolge ausgeführt werden müssen. Wie diese Reihenfolge genau aussieht, wird später bei der Konfiguration der Pipeline in der „main“-Funktion des Programmes festgelegt. Das überschreiben der „operator()()“-Funktion ist ähnlich wie beim Überschreiben der „handle“-Funktion der sequenziellen

Pipeline. Als Ergebnis wird die bearbeitete Zeile an die nachfolgende Aufgabe weitergeleitet. Sollte die Pipelineaufgabe die letzte Aufgabe gewesen sein, gibt sie den Wert „0“ oder auch „NULL“ zurück.

```
1  class LineExpansion : public tbb::filter{
2  private:
3      unsigned int width, valPerPix, newValPerPix;
4      clock_t start, end;
5  public:
6      LineExpansion(unsigned int In_Width, unsigned int In_ValuesPerPixel, unsigned int In_NewValuesPerPixel);
7      /*overwrite*/ void* operator()(void* item);
8      double getTime(void){
9          double temp = (double) end;
10         return (temp*1000)/CLOCKS_PER_SEC;
11     }
12 };
13 LineExpansion::LineExpansion(unsigned int In_Width, unsigned int In_ValuesPerPixel, unsigned int In_NewValuesPerPixel) :
14     tbb::filter(serial_in_order),
15         width(In_Width), valPerPix(In_ValuesPerPixel), newValPerPix(In_NewValuesPerPixel)
16 {end=0;}
17 void* LineExpansion::operator ()(void *item){
18     start=clock();
19     float* Cast_Item = (float*) item;
20     //delete[] item;
21     float* Line = new float[width*newValPerPix];
22     for(unsigned int i(0); i<width; ++i){
23         Line[i*newValPerPix] = Cast_Item[i*valPerPix]; //X
24         Line[i*newValPerPix+1] = Cast_Item[i*valPerPix+1]; //Y
25         Line[i*newValPerPix+2] = Cast_Item[i*valPerPix+2]; //Z
26         Line[i*newValPerPix+3] = Cast_Item[i*valPerPix+3]; //Inten
27         Line[i*newValPerPix+4] = (float)(-1); // provisionally default X
28         Line[i*newValPerPix+5] = (float)(-1); // provisionally default Y
29         Line[i*newValPerPix+6] = (float)(-1); // provisionally default Z
30     }
31     delete[] item;
32     end+=(clock()-start);
33     return Line;
34 }
```

Abbildung 5-32: Beispiel der Implementation einer Pipelineaufgabe mit TBB

Die Implementation der Pipeline in der „main“-Funktion des Programmes gestaltet sich etwas übersichtlicher als bei der sequenziellen Pipeline, da die TBB-Bibliothek dem Entwickler die Arbeit erleichtert. Die Abbildung 5-33 zeigt die Erstellung, Konfiguration und den Start der parallelen Pipeline. Die Implementation ist rech ähnlich zu der, der sequenziellen Pipeline. Ein Unterschied ist jedoch, dass direkt über einem Pipeline-Objekt gearbeitet werden kann (Zeile 2), was Verwechslungen fast unmöglich macht. Bei der sequenziellen Pipeline musste über der ersten Pipelineaufgabe gearbeitet werden, diese könnte bei einer ungeschickten Namensgebung auch mit einer anderen Pipelineaufgabe verwechselt werden. Die Aufgaben müssen auch bei dieser Pipeline in der Reihenfolge hinzugefügt werden, in der sie später ausgeführt werden sollen. Das geschieht mit dem Aufruf der Funktion „add_filter()“ über dem Pipeline-Objekt. Für den Start der Pipeline wird die Funktion „run()“ benutzt (Zeile 32 & 34). Der Parameter der Funktion ist, auf wie vielen Prozessoren die Pipelineaufgaben verteilt werden sollen. An dieser existieren zwei Möglichkeiten, die manuelle Festlegung der Anzahl der Prozessoren (Zeile 32) (können auch aus den Systemeigenschaften ausgelesen werden) und die automatische Erkennung und Verteilung durch die TBB-Bibliothek (Zeile 34). Nach dem Abschluss der Implementierung und der Erstellung der ausführbaren Datei des Programmes (*.exe), muss immer darauf geachtet werden das die DLLs[50] der TBB-Bibliothek verfügbar sein müssen. Ohne diese DLLs funktioniert die Anwendung nicht.

[50] DLL –**d**ynamic **l**inked **l**ibrary

```
1  // Anlegen eines Pipeline-Objektes
2  tbb::pipeline pipeline;
3
4  // Erstellen und Verknüpfen der Aufgabe "Lesen"
5  MyInputFilter input_filter(&InPut, height, width, valuesPerPixel);
6  pipeline.add_filter(input_filter);
7
8  // Erstellen und Verknüpfen der Aufgabe "LineExpansion"
9  LineExpansion FirstTask_Filter(width, valuesPerPixel, newValPerPix);
10 pipeline.add_filter(FirstTask_Filter);
11
12 // Erstellen und Verknüpfen der Aufgabe "Convert2GlobalSystem"
13 Convert2GlobalSystem SecondTask_Filter(width, newValPerPix, Orientation);
14 pipeline.add_filter(SecondTask_Filter);
15
16 // Erstellen und Verknüpfen der Aufgabe "ConvertedPoint2PictureCoordinate"
17 ConvertedPoint2PictureCoordinate ThridTask_Filter(width, newValPerPix, PRM, IOO, OO);
18 pipeline.add_filter(ThridTask_Filter);
19
20 // Erstellen und Verknüpfen der Aufgabe "PictureCoordinate2PixelCoordinate"
21 PictureCoordinate2PixelCoordinate FourthTask_Filter(width, newValPerPix, IOO,
22                                                     picture_height, picture_width,
23                                                     red, blue, green);
24 pipeline.add_filter(FourthTask_Filter);
25
26 // Erstellen und Verknüpfen der Aufgabe "Schreiben"
27 MyOutputFilter output_filter(&OutPut);
28 pipeline.add_filter(output_filter);
29
30 //Start Pipeline
31 if(NumberOfProcesses>0){
32     pipeline.run(NumberOfProcesses);
33 } else{
34     pipeline.run(tbb::task_scheduler_init::automatic);
35 }
```

Abbildung 5-33: Erstellen, Konfigurieren und Starten der parallelen Pipeline

6 Ergebnisse der Zeitmessungen

In den folgenden Abschnitten werden die Messungen der Laufzeit, in der sequenziellen und parallelen Variante der Implementation gegenübergestellt. Die Messungen wurden auf den in Kapitel 5.1 beschriebenen Computer-Systemen durchgeführt. Verglichen wird nicht die Laufzeitmessung zwischen den Systemen, sondern die Ausführung der sequenziellen und parallelen Implementation auf dem gleichen System. Aus dem Unterschied der gemessenen Werte lassen sich Aussagen über die Effizienz und den Nutzen der jeweiligen Implementation ziehen. Die Datengrundlagen der Programme bildeten immer die zwei gleichen Laserscandatensätze. Dabei handelte es sich um einen kleinen Datensatz mit ca. 74MB und einen großen Datensatz mit ca. 644MB. Der kleine Datensatz umfasst insgesamt 1432 Datenzeilen mit 3378 Laserscanpunkten pro Zeile, somit enthält die Datei insgesamt 4.837.296 Laserscanpunkte. Der große Datensatz besteht aus 4297 Datenzeilen mit jeweils 10136 Punkten und somit aus insgesamt 43.554.392 Laserscanpunkten. Beide Testdateien verwenden Punkte des Typs1[51], das bedeutet ein Punkt besteht aus vier Float-Werten. Ein Laserscanpunkt hat somit eine Größe von 16 Byte.

Für den Vergleich der Messzeiten wurden Diagramme aus den Originalmessdaten erstellt, um eine schnelle Übersicht zu vermitteln. Die Darstellung in den Diagrammen ist immer unterteilt in die Gesamtlaufzeit des Programmes und die Zeit, die das Programm für die reine Berechnung der Daten benötigt. Im Anhang dieser Arbeit sind auch die kompletten Messdaten enthalten.

6.1 Schnitte-Algorithmus

Für die Umsetzung dieses Algorithmus wurden drei Implementationen vorgenommen, eine komplett sequenzielle und zwei parallelisierte. Aufgrund der Einfachheit des Algorithmus fällt dieser in Kategorie „peinlich parallel“ (embarrassing parallel). Die beiden parallelen Implementierungen wurden mittels Threads realisiert, wobei zwei unterschiedliche Thread-Bibliotheken (POSIX- und Windows-Threads (MFC)) zum Einsatz kamen. Eine genauere Beschreibung des Algorithmus und seiner Implementation ist in den Kapiteln 2.2, 4, 5.3 nachlesbar.

6.1.1 Testcomputer TC1

Beim TC1 handelt es sich um einen Computer mit zwei Kernen, die genauen Informationen zum System können der Tabelle 5-2 entnommen werden. In der Abbildung 6-1 und Abbildung 6-2, sowie der Tabelle 6-1 und Tabelle 6-2, sind die Resultate der Messungen dargestellt. Die Datengrundlage für die Messungen bilden die im Kapitel 6 beschriebenen Laserscandateien. Die Abbildungen zeigen von links die Ergebnisse für den sequenziellen Algorithmus (Rot), den parallelen Algorithmus mittels Windows-Threads (Grün) und den parallelen Algorithmus mittels POSIX-Threads (Blau). In den Diagrammen sind zwei Blöcke dargestellt, der linke zeigt die gesamte benötigte Zeit und der rechte die benötigte Zeit für die reine Berechnung.

Aufgrund des im Kapitel 4 auf Seite 25 bestimmten 90-prozentigen Anteils der Berechnungen an der Gesamtzeit, war eine Beschleunigung des Programmes zu erwarten. Diese Beschleunigung spiegelt sich auch in den Abbildungen wieder. Die parallelisierten Programme benötigen im Durchschnitt nur 70% der Zeit, die das sequenzielle Programm benötigte. Die Aussage gilt sowohl für den kleinen, als auch den großen Scan. Die Ergebnisse sind damit bei einer Messreihe mit insgesamt 100 Durchläufen und unterschiedlich großen Eingangsdaten konstant. Des Weiteren lässt sich eine Aussage über die verwendeten Thread-Bibliotheken treffen. Dieses scheint nahezu mit gleicher Effizienz zu arbeiten. Die Messungen ergeben eine nur sehr kleine Differenz zwischen den

[51] Beschrieben in Kapitel 2.1 auf der Seite 2

Bibliotheken. Die POSIX-Threads scheinen einen kleinen Vorteil gegenüber den Windows-Threads zu haben. Der Geschwindigkeitsvorteil der POSIX-Threads liegt bei dem kleinen Laserscan bei rund 137ms und bei dem großen bei rund 655ms. Die Messresultate der kleinen und auch der großen Laserscandatei verhalten sich sehr ähnlich und auch so wie erwartet.

Die Werte für den realen SpeedUp der Messungen, Tabelle 6-1 und Tabelle 6-2, ergeben ebenfalls ein einheitliches Bild. Beide Thread-Bibliotheken verhalten sich nahezu identisch. Dennoch kommen die Ergebnisse für den realen SpeedUp nicht an die Sollwerte des theoretischen SpeedUp heran. Dieser liegt nach der Formel 9 für den TC1 bei einem Wert von 1,81[52]. Erreicht wurden jedoch nur Ergebnisse um den Faktor 1,42.

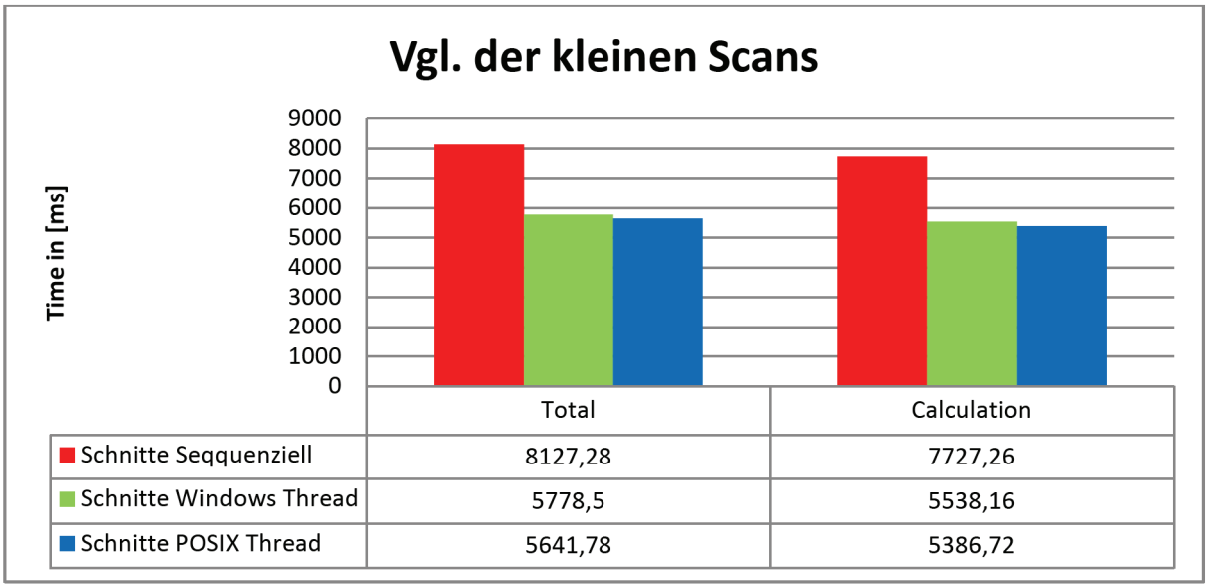

Abbildung 6-1: Messergebnisse des Schnitte-Algorithmus (kl. / TC1)

SpeedUp	*Total*	*Calculation*
Schnitte Sequenziell	1,000	1,000
Schnitte Windows Thread	1,406	1,395
Schnitte POSIX Thread	1,441	1,435

Tabelle 6-1: SpeedUp des Schnitte-Algorithmus (kleiner Datensatz / TC1)

[52] Siehe Tabelle 6-5 auf Seite 74

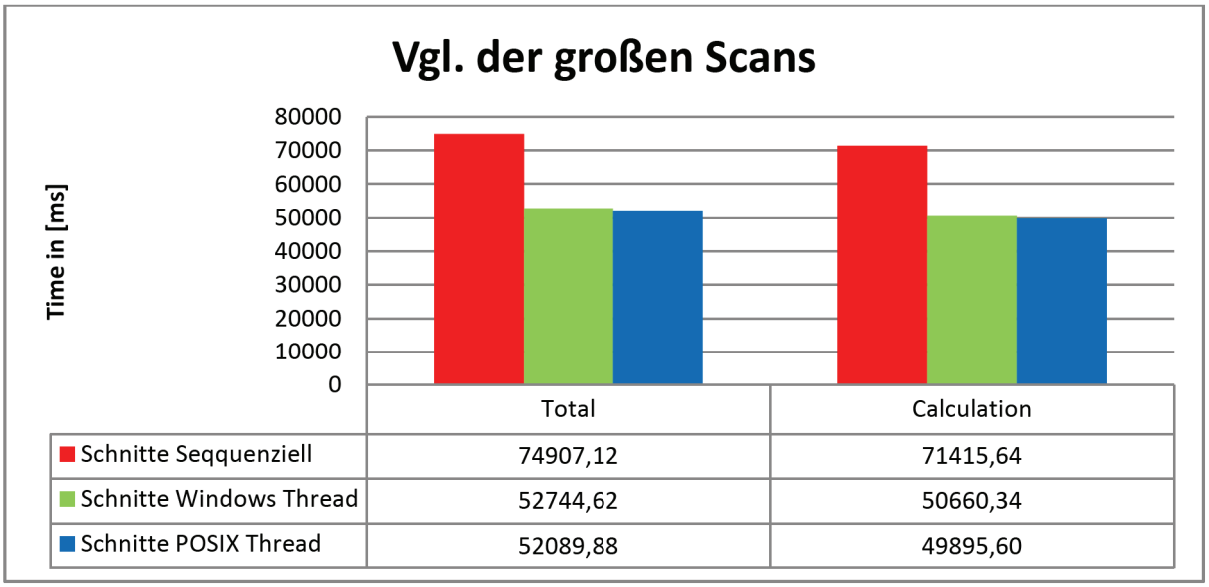

Abbildung 6-2: Messergebnisse des Schnitte-Algorithmus (gr. / TC1)

SpeedUp	*Total*	*Calculation*
Schnitte Sequenziell	1,000	1,000
Schnitte Windows Thread	1,420	1,410
Schnitte POSIX Thread	1,438	1,431

Tabelle 6-2: SpeedUp des Schnitte-Algorithmus (großer Datensatz / TC1)

6.1.2 Testcomputer TC2

Beim TC2 handelt es sich um einen Computer mit vier Kernen, die genauen Informationen zum System können der Tabelle 5-3 entnommen werden.

Die Abbildung 6-3 zeigt die Ergebnisse der Verarbeitung der kleinen Laserscandatei mit ca. 74MB. In der Y-Achse des Diagramms stellt die benötigte Zeit der Ausführung in Millisekunden dar, während auf der X-Achse die ausgeführten Programme farblich codiert dargestellt sind. Die sequenzielle Implementierung ist ganz links in Rot, die Implementierung mit Windows-Threads ist in der Mitte und grün und die POSIX-Threads Implementierung ist rechts und Blau dargestellt. Erkennbar ist eindeutig, dass die beiden parallelen Implementierungen deutlich mehr Zeit benötigten als die sequenzielle Variante. Die Festplatte als ein limitierender Faktor bei der Ausführung konnte durch die Vorbetrachtungen im Kapitel 4 auf der Seite 25 so gut wie ausgeschlossen werden. Das Ergebnis der Voruntersuchung war, dass die Festplatten Aktionen des Programmes im Durchschnitt bei lediglich 10% der gesamten Laufzeit liegen. Somit konnte nach Amdahls-Gesetz ein maximaler Geschwindigkeitsgewinn um den Faktor 10 ermittelt werden. Diese Annahme bestätigten auch die Messung auf dem Testcomputer TC1, bei dem genau das entgegengesetzte Verhalten bestimmt werden konnte. Eine erste Theorie wäre sicherlich, dass die Testcomputer eine unterschiedliche Laufzeitumgebung (runtime environment) für C++ besitzen. Dem ist jedoch nicht so. Das unterschiedlich Ergebnisse zustande kommen können, wenn unterschiedlich Laufzeitumgebungen verwendet werden, war schon vor dem Durchführen der Laufzeitmessungen klar. Es wurde somit von Vorhinein dafür gesorgt, dass auf den Testcomputern gleiche Bedingungen herrschen, soweit das möglich war. Die Systeme verwenden sicherlich andere Hardwarebausteine und somit auch unterschiedliche Treiber.

Die zweite Theorie leitet sich aus einem Beispiel von Bauke und Mertens ab und bezieht sich auf den Flaschenhals der Von-Neumann-Architektur. Dieses Beispiel beschreibt, wie die Höchstleistung eines Systems durch den Hauptspeicher begrenzt werden kann. Zwar bezieht sich das Beispiel

aufgrund des Alters der Literatur auf, für die heutigen Tage, sehr langsame Speichertakte (100Mhz), aber das Verhältnis zwischen dem Takt des Hauptspeichers und der CPU stimmt noch immer. Sicherlich sind die Speichertakte des Hauptspeichers heutzutage standardmäßig mindestens dreimal so schnell, aber auch die Taktfrequenz der CPU hat sich erhöht. Das Beispiel beschreibt auch, dass ein eingesetzter Cache das Flaschenhalsproblem verringern kann. Nun teilen sich bei der parallelen Verarbeitung von Daten auf Shared-Memory-System unter Umständen mehrere Kerne einen Cache. Dadurch kann es dazu kommen, dass der Cache bei der Verarbeitung von großen Datenmengen einfach zu klein ist, um die Daten von mehreren Kernen gleichzeitig zu beinhalten. Das hat wiederum zur Folge, dass sich der Anteil von „cache misses“ stark erhöht. Wie genau ein Cache-Speicher funktioniert, beschreibt das Kapitel 5.1. An dieser Stelle reicht es zu wissen das ein „cache miss“ gleichbedeutend ist mit einem Zugriff auf den langsamen Hauptspeicher des Systems. Wenn die Testcomputer TC1 und TC2 genauer betrachtet werden fällt auf, dass der TC1 mehr Cache-Speicher besitzt als der TC2, genauer gesagt doppelt so viel pro verfügbaren Kern. Jeder Kern bearbeitet in dem parallelisierten Programm des Schnitt-Algorithmus immer eine Datenzeile aus der Laserscandatei bevor er eine neue aus der Datei liest und bearbeitet. Die Frage, die sich stellt, ist der Cache bei vier gleichzeitig verarbeiteten Datenzeilen mit einem gewissen Overhead (sonstige benötigte Werte und andere Programme wie zum Beispiel dem Betriebssystem) schon voll und müssen für die parallelisierte Arbeit ständig Daten aus dem Hauptspeicher nachgeladen werden. Das lässt sich im Groben leicht nachrechnen, siehe Formel 22. Bei einem Cache von „2x2MB“ teilen sich zwei Kerne einen Cache mit 2MB, was theoretisch vollkommen ausreichen sollte. Nach der Rechnung in der Formel 22 belegen zwei Datenzeilen, die auf unterschiedlichen Kernen bearbeitet werden, lediglich 106kB Cache bei der kleinen Laserscandatei und 318kB bei der großen. Somit stehen dem System in jedem der zwei Cache-Speicher 1942kB bzw. 1706kB Speicher für den Overhead zur Verfügung, was mehr als ausreichend ist.

Formel 22: Benötigter Cache-Speicher für die parallele Verarbeitung

$$benötigter\ Speicher\ pro\ Punkt := 4Byte \cdot 4Werte = 16Byte$$
$$benötigter\ Speicher\ pro\ Zeile_{klein} := 16Byte \cdot 3378Punkte \approx 53kB$$
$$benötigter\ Speicher\ pro\ Zeile_{groß} := 16Byte \cdot 10136Punte \approx 159kB$$

Aufgrund der obigen Rechnungen kann die zweite Theorie ausgeschlossen werden, dennoch bleibt diese ein nicht zu verachtendes Risiko bei parallelen Anwendungen. (Abschnitt nach (8 S. 7-8))

Eine dritte Theorie war, dass der Zugriff auf die verwendete DLL von Lupos3D lediglich exklusiv erfolgen konnte. Das bedeutet für die parallelen Implementationen, dass jeweils nur ein Thread eine Berechnung mit den Funktionen der DLL durchführen konnte, während alle anderen Threads warten mussten. Um das zu untersuchen, wurde eine Funktion implementiert, die die gleichen Parameter wie die Schnitte-Funktion aus der DLL akzeptierte. Ziel der Funktion war es Zeit durch Rechenoperationen zu verbrauchen. Der Unterschied zur in der DLL implementierten Funktion war die direkte Implementierung der Testfunktion im Quellcode des Programms. Somit konnte ein lediglich exklusiver Zugriff ausgeschlossen werden. Die Ausführung der Testfunktion ergab, dass anscheinend wirklich nur exklusive auf die verwendete DLL zugegriffen werden konnte. Die Testfunktion lieferte ähnliche Ergebnisse bei der parallelen Implementation wie bei Testcomputer TC1 und wie es auch erwartet wurde. Mit einer Steigerung der genutzten Prozessoren sank die benötigte Zeit. Bis zu einem gewissen Grad, an dem das System mit der Anzahl der gestarteten Threads an sein Grenzen stößt und wieder mehr Zeit für die Berechnung benötigt. Die einfachste Erklärung warum das Problem nicht beim TC1 auftrat ist, es trat auf wirkte sich nur nicht so stark aus. Wenn die DLL nur einen exklusiven Zugriff erlaubt, kommen sich zwei Prozesse (wie beim TC1) weniger häufig beim Zugriff in die Quere als es vier Prozesse (wie beim TC2) tun.

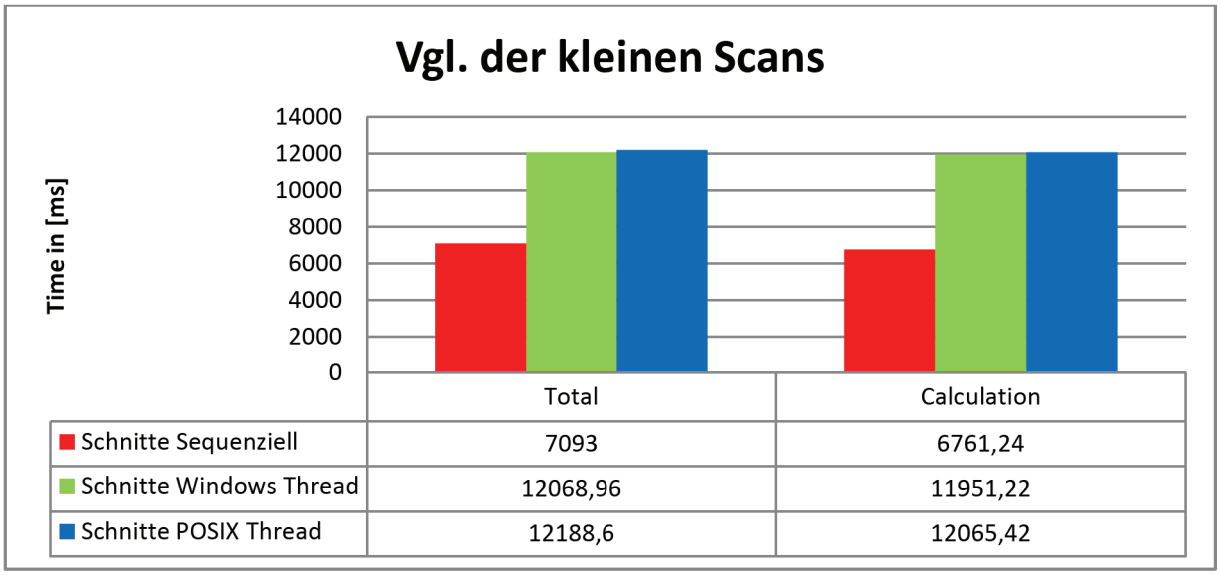

Abbildung 6-3: Messergebnisse des Schnitte-Algorithmus (kl. / TC2)

Die Tabelle 6-3 zeigt den ermittelten realen SpeedUp, dieser wurde nach der Formel 10 berechnet. Die Werte verdeutlichen noch einmal, dass die parallele Verarbeitung fast doppelt so viel Zeit benötigte wie die sequenzielle. Es lässt sich aber beim Vergleich der beiden Thread-Bibliotheken erkennen, dass die Windows Bibliothek minimal schneller ist als die Bibliothek der POSIX-Threads.

SpeedUp	*Total*	*Calculation*
Schnitte Sequenziell	1,000	1,000
Schnitte Windows Thread	0,588	0,566
Schnitte POSIX Thread	0,582	0,560

Tabelle 6-3: SpeedUp des Schnitte-Algorithmus (kleiner Datensatz / TC2)

Ein Schluss, der aus diesen Messungen gezogen werden kann, ist das die Windows-Thread- und die POSIX-Thread-Bibliothek sich nahezu identisch verhalten. Es ist nicht möglich bei der geringen Menge der zu verarbeitenden Daten eine bessere oder schlechtere Bibliothek zu identifizieren. Die Laufzeitunterschiede, die sich in der Berechnung ergeben und sich auf die Gesamtlaufzeit übertragen, liegen gerade einmal bei knapp über 100ms. Somit noch nicht einmal im Sekundenbereich, daher sind die Unterschiede zu vernachlässigen.

Bei der Verarbeitung des großen Laserscans mit ca. 644MB ergibt sich ein ähnliches Bild der Messergebnisse wie bei dem kleinen Scan. Auch hier scheint der exklusive Zugriff die Funktionen der DLL wieder ausschlaggebend für das unerwartete Auftreten längerer Laufzeiten bei den parallelen Implementationen zu sein. Allerdings ist in der Abbildung 6-4 ein deutlicherer Unterschied in der Laufzeit und der Berechnung der beiden verendeten Thread-Bibliotheken zu erkennen. Der Geschwindigkeitsvorteil der POSIX-Threads gegenüber den Windows-Threads ergibt sich ziemlich genau aus der Berechnungszeit. Das lässt den Rückschluss zu, dass die Bibliothek der POSIX-Threads in der Lage ist, die gestarteten Threads effektiver zu verwalten, als es bei der Windows-Bibliothek der Fall ist.

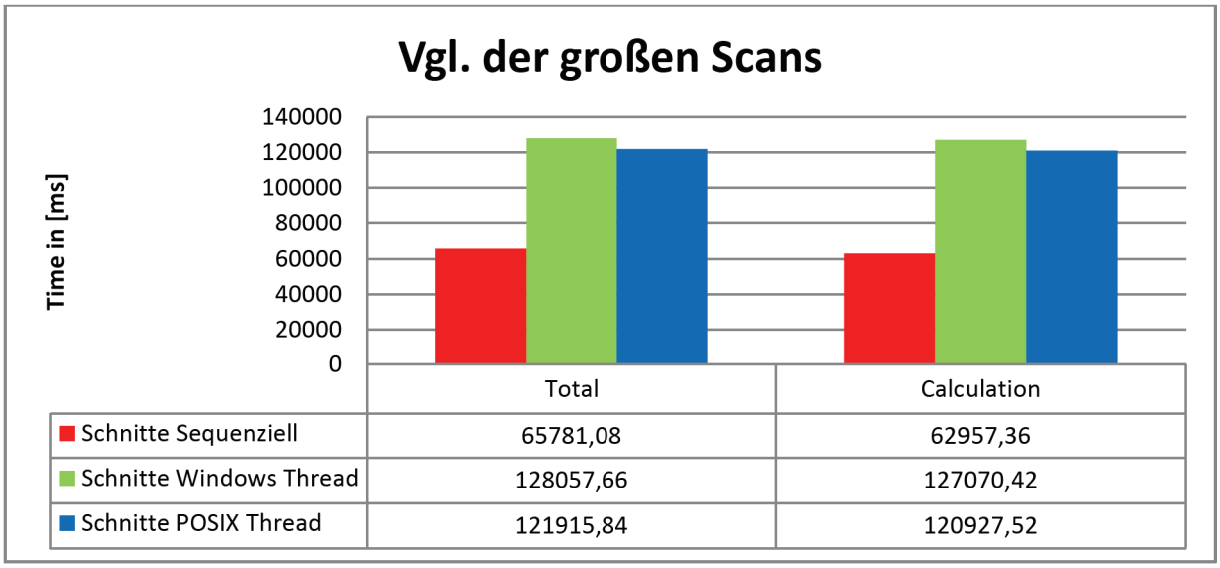

Abbildung 6-4: Messergebnisse Schnitte-Algorithmus (gr. / TC2)

Auch an den berechneten Werten für den realen SpeedUp (siehe Tabelle 6-4) spiegeln sich die oben genannten Erkenntnisse wieder. Denn der SpeedUp der POSIX-Threads ist höher als der bei den Windows-Threads, sowohl bei der Zeit für die reine Berechnung als auch bei der Gesamtzeit.

SpeedUp	*Total*	*Calculation*
Schnitte Sequenziell	1,000	1,000
Schnitte Windows Thread	0,514	0,495
Schnitte POSIX Thread	0,540	0,521

Tabelle 6-4: SpeedUp des Schnitte-Algorithmus (kleiner Datensatz / TC2)

6.1.3 Abschlussbetrachtung

In einer letzten Betrachtung für beide Testcomputer spielt der Vergleich zwischen dem theoretischem und dem realen SpeedUp eine Rolle. Die Tabelle 6-5 zeigt den, mit den Formeln und Erkenntnissen aus Kapitel 4, berechneten theoretischen SpeedUp, in drei Spalten. Die erste Spalte steht für den Grad der Parallelisierbarkeit nach Amdahl. Dieser Grad besagte, dass wenn 90% des Programmes parallelisierbar sind, ein maximaler Geschwindigkeitsvorteil des Faktors 10 bei unendlich vielen verwendeten Prozessoren existiert[53]. Den verwendeten Testcomputern standen jedoch nicht unendlich viele Prozessoren zur Verfügung, sondern lediglich zwei beim TC1 und vier beim TC2. Die zweite und die dritte Spalte der Tabelle 6-5 zeigen darum den theoretischen SpeedUp auf diesem System, unter der Berücksichtigung der Anzahl der verfügbaren Prozessoren, an. Der theoretische SpeedUp muss nicht dem realen SpeedUp entsprechen, denn es existieren eine Reihe von Einflussfaktoren. Beispielsweise können Hintergrundprogramme wichtige Prozessorzeit kosten oder den Zugriff auf die Festplatte kurzzeitig verhindern.

Bei einem Vergleich der Werte der Tabelle 6-1, Tabelle 6-2, Tabelle 6-3 und Tabelle 6-4 steht fest, dass keiner der realen SpeedUp's mit den Werten des ermittelten theoretischen SpeedUp übereinstimmt. Der Vergleich muss kritisch betrachtet werden, denn Probleme, die bei der Verarbeitung auftraten[54], spiegeln sich im realen SpeedUp wieder. Der reale SpeedUp des TC2 war von dem DLL-Problem besonders stark betroffen und kann eigentlich schon gar nicht mehr mit dem theoretischen SpeedUp verglichen werden, da zwischen den beiden Ergebnissen Welten liegen.

[53] Siehe Abbildung 4-2 auf Seite 27
[54] Exklusiver Zugriff auf die verwendete DLL (Kapitel 6.1.2)

Interessanter wird es bei der Betrachtung des realen und theoretischen SpeedUp des TC1. Der reale SpeedUp scheint ist zwar auch vom DLL-Problem betroffen, aber anscheinend nicht so dramatisch wie der TC2. Die Unterschiede der SpeedUp's ergeben sich somit aus dem Problem bei der Verarbeitung und unbestimmten Hintergrundaktivitäten.

Ein letzter Schluss ist somit, dass sich der TC1 zwar nicht besonders gut geschlagen hat. Aber zumindest zeigte der TC1 eine Tendenz zu dem, was ermittelt werden sollte, nämlich ob sich der Algorithmus parallelisieren lässt.

SpeedUp max	*SpeedUp p=2 (TC1)*	*SpeedUp p=4 (TC2)*
10	1,818182	3,076923

Tabelle 6-5: Theoretischer SpeedUp des Schnitte-Algorithmus

6.2 Photo2Scan

Bei diesem Algorithmus wurden drei verschiedene Implementationen getestet. Zum einen eine sequenzielle Variante zu Vergleichszwecken und zum anderen zwei parallele Implementierungen. Die parallelen Implementierungen nutzen das Entwurfsmuster der Pipeline, welches schon im Kapitel 3.2.2.4 in seiner Theorie erläutert wurde. Eine detaillierte Beschreibung der Implementation kann in dem Kapitel 5.4 nachgelesen werden. An dieser Stelle sei nur kurz erwähnt, dass die Implementierung des Entwurfsmusters der Pipeline eine sequenzielle und eine parallele Variante enthält. Die sequenzielle Pipeline arbeitet nur mit einem Prozessor, während die parallele Pipeline mit mehreren Prozessoren arbeiten kann. Die folgenden Abschnitte beschreiben die Ergebnisse der durchgeführten Messungen.

6.2.1 Testcomputer TC1

Beim TC1 handelt es sich um einen Computer mit zwei Kernen, die genauen Informationen zum System können der Tabelle 5-2 entnommen werden.

In der Abbildung 6-6 und Abbildung 6-7 sind die Messergebnisse des Photo2Scan-Algorithmus auf dem Testcomputer TC1 dargestellt. Die Diagramme zeigen auf der X-Achse die ausgeführten Programme und auf der Y-Achse kann die Laufzeit eines jeweiligen Programmes in Millisekunden abgelesen werden. In den Diagrammen existieren des Weiteren zwei Ansichten, die Linke (Total) zeigt die benötigte Gesamtlaufzeit eines Programmes und die Rechte (Calculation) zeigt die benötigte Zeit für die reinen Berechnungen (ohne Dateiarbeit). Zur besseren Unterscheidung sind die verschiedenen Programme farblich gekennzeichnet. Die Farbe Rot (links) steht für die sequenzielle Implementation des Algorithmus, Grün (Mitte) steht für die Implementation der sequenziellen Pipeline und die Farbe Blau (rechts) kennzeichnet Variante des Algorithmus mit der parallelen Pipeline.

Wie im Kapitel 4 beschrieben, ist der Algorithmus stark von den Festplattenoperationen abhängig. Diese lassen sich leider bei der auf dem Testcomputer verwendeten Hardware nicht beschleunigen bzw. umgehen. Mit nur einem verfügbaren Lese-/Schreibkopf ist es unmögliche Daten parallel zu lesen oder zu schreiben. Daher ist für die Messungen eine große Verringerung der Gesamtlaufzeit eher nicht zu erwarten. Allerdings sollten die Ergebnisse der reinen Berechnung deutlich schneller werden, da die Berechnung nicht von der Festplatte abhängt und die einzelnen Schritte auch Daten unabhängig sind.

Ein Problem, das bei der Nutzung einer Pipeline immer besteht, lässt sich nicht umgehen. Eine Pipeline ist erst nach einem kompletten Durchlauf einer Aufgabe durch alle Teilbereiche voll leistungsfähig. Wenn eine Pipeline beispielsweise aus fünf Teilbereichen besteht, ist die Pipeline erst

dann voll leistungsfähig, wenn mit der Bearbeitung des fünften Problems (oder auch Aufgabenstellung) in dem ersten Teilbereich der Pipeline begonnen wurde. Von der anderen Seite betrachtet, macht eine Pipeline also auch nur Sinn, wenn die Summe der Probleme oder Aufgabenstellungen größer ist als die Summe der Teilbereiche der Pipeline. Die Abbildung 6-5 zeigt das grafisch an dem Aufbau der parallelen Pipeline. Die Pipeline besteht aus sechs Teilbereichen, die für ein Problem der Reihe nach ausgeführt werden müssen. In der Abbildung 6-5 kommen drei Abkürzungen vor GS, BK und PK. Diese stehen für globales System (GS), Bildkoordinate (BK) und Pixelkoordinate (PK). Die Verarbeitung der Probleme beginnt jeweils mit dem Einlesen der Daten einer Zeile der Scandatei. Der zweite Schritt ist das Ausfassen des benötigten Speichers, währenddessen kann schon das nächste Problem eingelesen werden und so weiter. Die Pipeline erlangt ihre volle Auslastung erst, wenn mit der Bearbeitung des sechsten Problems begonnen wird, dass ist in der Abbildung 6-5 hervorgehoben (Grün). Bei der Anzahl der Probleme in den beiden Testdatensätzen kann das jedoch vernachlässigt werden.

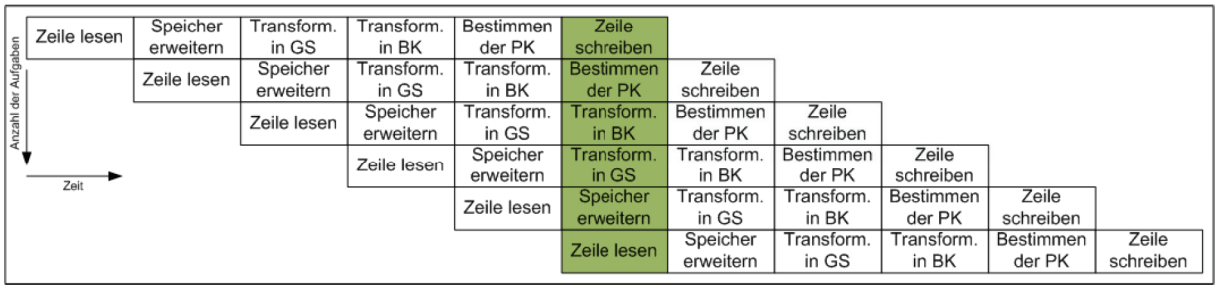

Abbildung 6-5: Auslastung der parallelen Pipeline

Die Ergebnisse der drei unterschiedlichen Pipeline-Implementationen, werden immer zwei gleiche Eigenschaften aufweisen. Aufgrund der starken Festplatten Abhängigkeit des Algorithmus werden die benötigten Zeiten für das Lesen und Schreiben von Daten von beziehungsweise auf die Festplatte sich erhöhen. Das bedeutet bei der gleichen Pipeline Implementation, wird der Anteil der Lese- und Schreiboperationen mit der Anzahl der zu bearbeitenden Aufgaben ansteigen. Die Formel 23 zeigt, wie die Verhältnisse der Operationen des Algorithmus mit den gemittelten Messwerten bestimmt werden kann.

Formel 23: Bestimmung des Anteils von Festplattenoperationen

$$Anteil_{Berechnung} : \frac{Zeit_{Gesamt}}{100\%} = \frac{Zeit_{Berechnung}}{X}$$

$$X = \frac{(100\% \cdot Zeit_{Berechnung})}{Zeit_{Gesamt}}$$

$$Anteil_{Festplatte} = 100\% - Anteil_{Berechnung}$$

Die Tabelle 6-6 zeigt, welche Ergebnisse die Berechnungen der Anteile der Gesamtlaufzeit ergaben. An diesen Werten lässt sich die erste Vermutung bestätigen. Die Ergebnisse legen auch die Vermutung nahe, dass die Berechnung der Teilschritte des Algorithmus mittels einer Pipeline wesentliche effizienter sind, als bei einer rein sequenziellen Implementation des Algorithmus. Die Abbildung 6-6 und die Abbildung 6-7 belegen das auch noch einmal grafisch.

Die zweite Eigenschaft ist, dass die Berechnung mit steigender Parallelität schneller wird. Die sequenzielle Implementation kommt komplett ohne parallele Techniken aus und ist daher auch die langsamste Variante. Das trifft sowohl auf die Festplattenoperationen, als auch auf die

Berechnungen zu. Die sequenzielle Pipeline besitzt zumindest schon den parallelen Ansatz der Pipeline, ist aber immer noch für nur einen Thread entwickelt. Diese Variante ist daher von der Geschwindigkeit auch als Mittelmaß der drei Implementationen einzuordnen. Die dritte Variante, die parallele Pipeline, besitzt den höchsten Grad der Parallelität der drei Implementationen. Sie besitzt sowohl den parallelen Ansatz der Pipeline, als auch die Verteilung der Teilschritte der Abarbeitung auf mehrere Threads (je nach Bedarf). Somit ist diese als schnellste Implementation einzuordnen. Auch diese Theorie lässt sich an der Tabelle 6-6 und der Abbildung 6-6 sowie der Abbildung 6-7 bestätigen. Denn auch unter den Gesichtspunkten zeigen die Werte, dass ein höherer Grad an Parallelität schnellere Programme hervorbringt. Die Werte zeigen aber auch, dass die Größe der zu bearbeitenden Aufgabenstellung (Vergleich zwischen dem kl. Scan und dem gr. Scan) kaum noch Auswirkungen auf eine Steigerung der Berechnungsgeschwindigkeit haben. Die Pipeline-Implementationen konnten das Verhältnis der benötigten Zeiten für die Berechnung und die Festplattenoperationen nicht mehr so stark beeinflussen. Das bestärkt wiederum die Aussage aus Kapitel 4 auf Seite 25, dass die Festplatte der limitierende Faktor des Algorithmus ist.

	Anteil Festplattenoperationen		**Anteil Berechnung**	
	kl. Scan	gr. Scan	kl. Scan	gr. Scan
Photo2Scan sequenziell	≈ 37%	≈ 63%	≈ 63%	≈ 37%
Photo2Scan Pipe Sequenziell	≈ 90%	≈ 97%	≈ 10%	≈ 3%
Photo2Scan Pipe Parallel	≈ 96%	≈ 97%	≈ 4%	≈ 3%

Tabelle 6-6: Anteile der Operationen am Photo2Scan-Algorithmus (TC1)

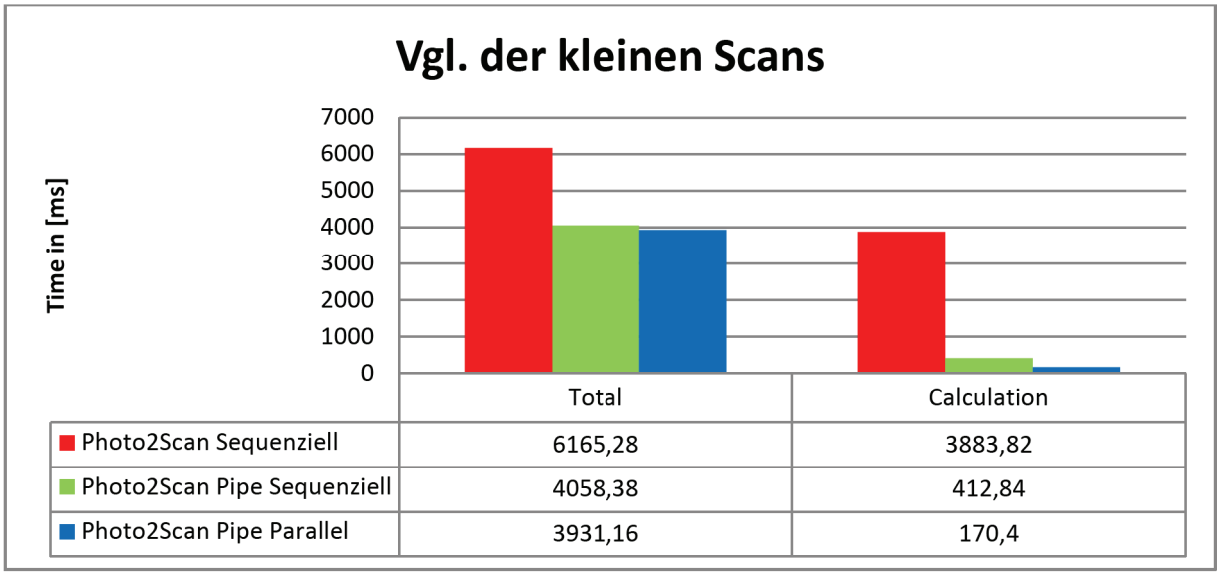

Abbildung 6-6: Messergebnisse Photo2Scan-Algorithmus (kl. / TC1)

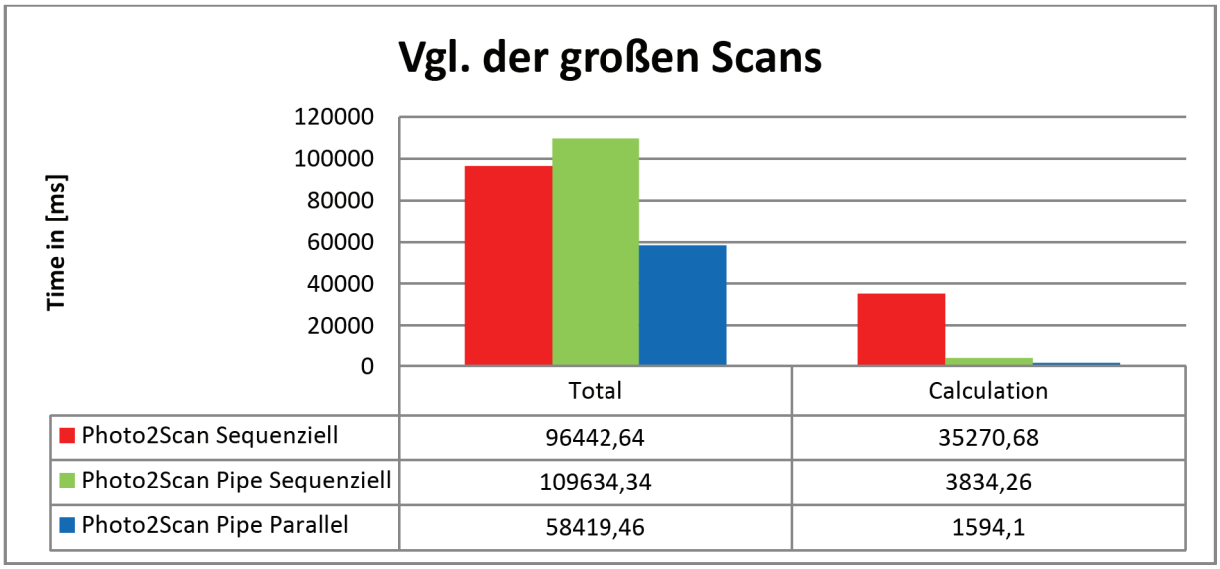

Abbildung 6-7: Messergebnisse Photo2Scan-Algorithmus (gr. / TC1)

Der Geschwindigkeitsgewinn dank der vorgenommenen Parallelisierungen lässt sich am besten an dem berechneten SpeedUp erkennen. Wie der SpeedUp berechnet wird, ist durch die Formel 10 auf der Seite 28 beschrieben. Als Referenz für die Berechnung gilt die rein sequenzielle Implementation des Algorithmus, darum ist der SpeedUp für diese Variante gleich Eins. In der Tabelle 6-7 und der Tabelle 6-8 kann der berechnete SpeedUp für den Testcomputer 1 jeweils für die kleine und große Scandatei entnommen werden. In diesen Werten spiegeln sich auch die Vermutungen der vorherigen Abschnitte wieder. Der Geschwindigkeitsvorteil steigt mit zunehmender Parallelität der Implementation, dabei kommt der eigentliche Gewinn jedoch aus der effektiveren Berechnung. Werden die Werte für die Berechnungen separat betrachtet und die Festplattenoperationen vernachlässigt, wird ein enormer Geschwindigkeitsvorteil erreicht. Werden die Werte jedoch in ihrer Gesamtheit betrachtet, steht fest, dass nahezu der gesamte Gewinn aus der Berechnung kommt und stark von den Festplattenoperationen ausgebremst wird.

SpeedUp	*Total*	*Calculation*
Photo2Scan Sequenziell	1,000	1,000
Photo2Scan Pipe Sequenziell	1,519	9,408
Photo2Scan Pipe Parallel	1,568	22,792

Tabelle 6-7: Photo2Scan - SpeedUp (kl. / TC1)

SpeedUp	*Total*	*Calculation*
Photo2Scan Sequenziell	1,000	1,000
Photo2Scan Pipe Sequenziell	0,880	9,199
Photo2Scan Pipe Parallel	1,651	22,126

Tabelle 6-8: Photo2Scan - SpeedUp (gr. / TC1)

6.2.2 Testcomputer TC2

Beim TC2 handelt es sich um einen Computer mit vier Kernen, die genauen Informationen zum System können der Tabelle 5-3 entnommen werden.

Auch hier gelten in erster Linie die Voraussetzungen und Vermutungen aus dem vorherigen Kapitel 6.2.1. Die Festplatte ist auch bei dem TC2 der limitierende Faktor. Des Weiteren wird bei den Messungen erwartet, dass mit zunehmender Parallelität ein Geschwindigkeitsvorteil entsteht und sich das Verhältnis der benötigen Zeiten (Festplattenoperationen und Berechnung) im Vergleich zur

sequenziellen Implementierung stark verschiebt. Die grafische Aufteilung der Abbildung 6-8 und Abbildung 6-9 sind mit denen im Kapitel 6.2.1 identisch und bedürfen keiner weiteren Erläuterung.

Zunächst steht das Verhältnis von Festplattenoperationen und Berechnungszeit im Fokus. Die Vermutung, dass mit steigender Parallelität sich die benötigten Zeiten stark in Richtung Festplattenoperationen verlagern werden, wurde schon in Kapitel 6.2.1 eingeführt und soll mittels der Messwerte des TC2 weiter bestätigt werden. Die aus den Messwerten, mittels der Formel 23, ermittelten Ergebnisse sind in der Tabelle 6-9 dargestellt. Die Tabelle bestätigt, wie auch schon Tabelle 6-6 die Verlagerung der benötigten Zeiten bei steigender Parallelität. Aufgrund der unterschiedlichen Hardwarekomponenten, die im TC1 und TC2 verbaut wurden, weichen die ermittelten Werte des TC1 und des TC2 natürlich voneinander ab. Die Tendenz lässt sich dennoch erkennen. Die Abbildung 6-8und Abbildung 6-9 zeigen noch einmal grafisch, wie sich das Verhältnis der Festplatten- zu den Berechnungsoperationen verschoben hat. An dieser Abbildung fällt auch auf, dass die Verwendung von mehr Prozessoren nur minimale Vorteile für die Berechnung bringt, im Vergleich zu der Pipeline Implementierung mit nur einem Prozessor.

	Anteil Festplattenoperationen		**Anteil Berechnung**	
	kl. Scan	gr. Scan	kl. Scan	gr. Scan
Photo2Scan sequenziell	≈ 36%	≈ 34%	≈ 64%	≈ 66%
Photo2Scan Pipe Sequenziell	≈ 79%	≈ 89%	≈ 21%	≈ 11%
Photo2Scan Pipe Parallel	≈ 82%	≈ 84%	≈ 18%	≈ 16%

Tabelle 6-9: Anteile der Operationen am Photo2Scan-Algorithmus (TC2)

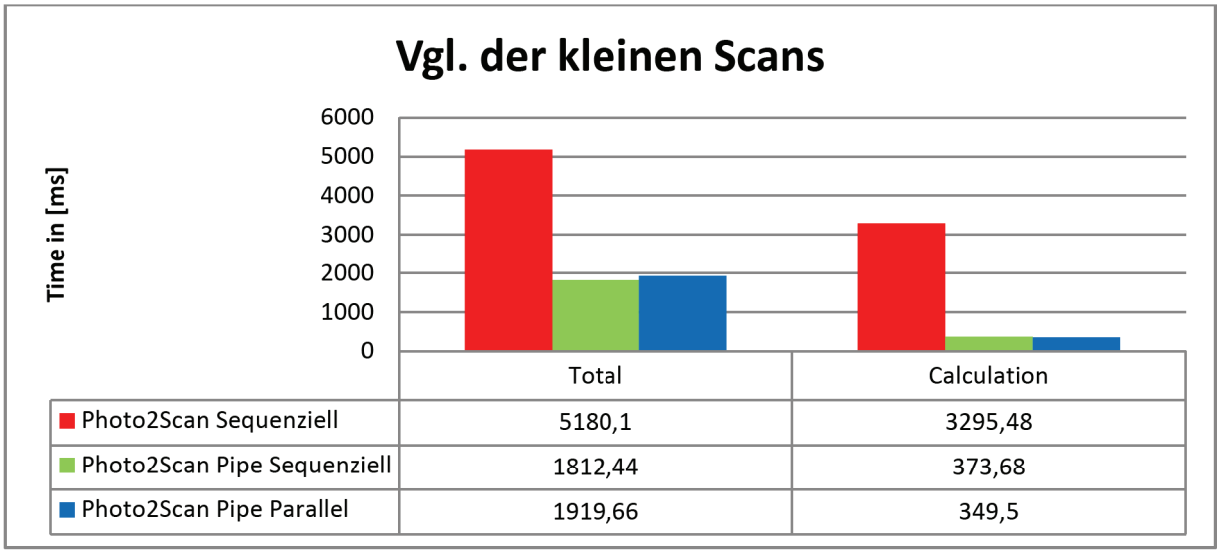

Abbildung 6-8: Messergebnisse Photo2Scan-Algorithmus (kl. / TC2)

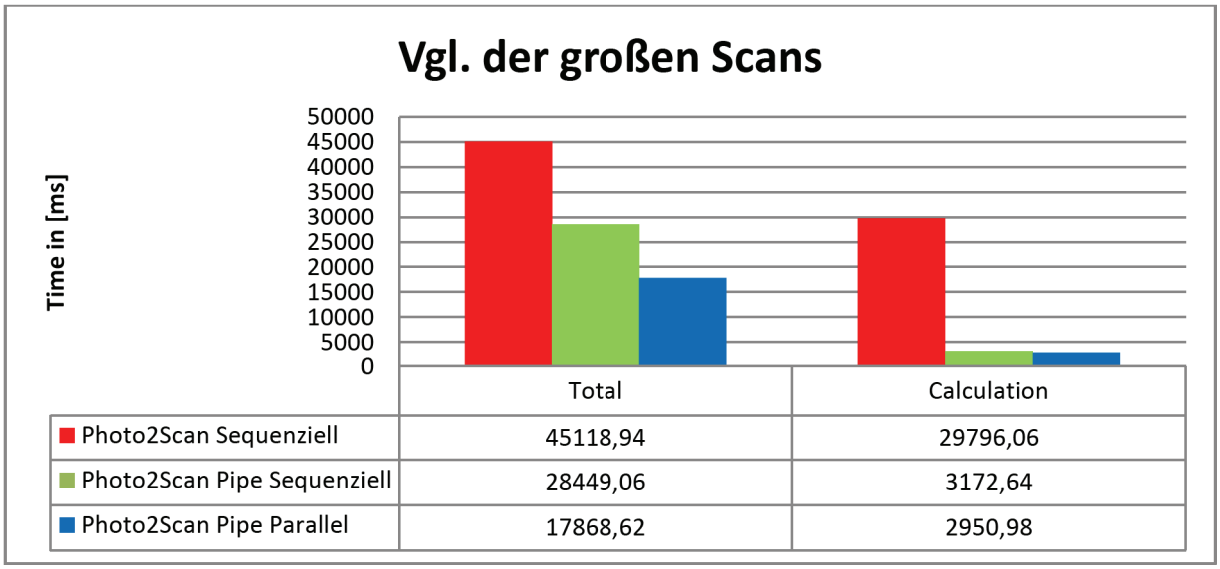

Abbildung 6-9: Messergebnisse Photo2Scan-Algorithmus (gr. / TC2)

Um den tatsächlichen Geschwindigkeitsgewinn bei dem TC2 beurteilen zu können, ist es sinnvoll auch hier den SpeedUp zu berechnen. Zum Vergleich der unterschiedlichen Implementationen und ihres SpeedUp's dienen die Tabelle 6-10 und die Tabelle 6-11. Diese Tabelle ist genauso strukturiert wie die Tabelle 6-7, und zeigt somit in der ersten Zeile die Referenzzeit der sequenziellen Implementierung des Photo2Scan-Algorithmus. Die Ergebnisse des realen SpeedUp's zeigen sowohl beim kleinen als auch beim großen Datensatz eindeutig einen Geschwindigkeitsgewinn. Bei beiden Datensätzen scheint der Vorteil aus den Berechnungen zukommen. Dies scheint auch logisch, da die Berechnungen unabhängig voneinander durchgeführt werden können und mehrere Prozessoren somit nahezu wirklich parallel arbeiten können. Des Weiteren zeigen die Werte einen interessanten unterschied zwischen den parallelen Implementationen. So scheint die sequenzielle Pipeline für den kleinen Datensatz in ihrer Gesamtheit effizienter zu arbeiten als die parallele Pipeline, obwohl die parallele Pipeline als die schnellste eingestuft werden kann. Beim großen Datensatz lässt sich jedoch ein eindeutig umgekehrter Sachverhalt erkennen. Anscheinend existiert bei der parallelen Pipeline bedingt durch die komplexe Struktur ein gewisser Overhead, der sich erst durch die Summe der bearbeiteten Aufgabe in der Pipeline relativiert.

SpeedUp	*Total*	*Calculation*
Photo2Scan Sequenziell	1,000	1,000
Photo2Scan Pipe Sequenziell	2,858	8,819
Photo2Scan Pipe Parallel	2,698	9,429

Tabelle 6-10: Photo2Scan - SpeedUp (kl. / TC2)

SpeedUp	*Total*	*Calculation*
Photo2Scan Sequenziell	1,000	1,000
Photo2Scan Pipe Sequenziell	1,586	9,392
Photo2Scan Pipe Parallel	2,525	10,097

Tabelle 6-11: Photo2Scan - SpeedUp (gr. / TC2)

Es zeigt sich auch bei diesem Testcomputer, dass mit steigender Parallelität die Geschwindigkeiten der Implementierungen für diesen Algorithmus zunehmen und dass die Festplatte der limitierende Faktor ist. Denn der Geschwindigkeitsgewinn der Berechnungen ist weitaus größer als der der gesamten Implementation.

6.2.3 Abschlussbetrachtung

Bei einem Vergleich der verwendeten Testcomputer lassen sich noch ein paar weitere Schlussfolgerungen der Implementationen ziehen.

So scheint der reale SpeedUp, der jeweils in den Kapiteln 6.2.1 und 6.2.2 dargestellt ist, beim TC2 nicht mehr so eindeutig von den Berechnungen abzuhängen, wie beim TC1. Das lässt sich daran erkennen, dass die ermittelte SpeedUp für die Berechnungen der parallelen Pipeline nur noch halb so hoch ist wie bei dem TC1. Ein Grund dafür könnt die komplexe Struktur der verwendeten Pipeline sein. Diese stammt aus der Bibliothek TBB und verwendet alle zur Verfügung stehenden Prozessoren des Systems. Diese bekommen dann wiederum je nach dem Anfallen von Aufgaben neue hinzu oder es werden dem Prozessor Aufgaben entzogen und umverteilt. Dieses erfordert möglicherweise einen hohen Verwaltungsaufwand, der anscheint stark von der Anzahl der Prozessoren im System abhängt. Beim realen SpeedUp des TC1 (Tabelle 6-7 und Tabelle 6-8) war kein Overhead im Vergleich zur Gesamtlaufzeit der sequenziellen Pipeline erkennbar. Bei TC2 (Tabelle 6-10 und Tabelle 6-11) hingehen fiel ein gewisser Overhead zur Gesamtlaufzeit der sequenziellen Pipeline auf. Es scheint also, dass die Implementation der parallelen Pipeline durch die Verwendung der TBB einen höheren Verwaltungsaufwand für die Umverteilung von Aufgaben an die Prozessoren aufweist, je größer die Anzahl der Prozessoren ist, die dem System zur Verfügung stehen. Gleichzeitig scheint es jedoch auch so, dass gerade durch die Umverteilung der Aufgaben der Prozessoren ein nahezu konstanter SpeedUp auf dem gleichen System bei einer steigenden Anzahl von Aufgaben erreicht wird. So scheint es fast so als würde es die TBB-Pipeline auch schaffen das Einlesen und Schreiben von Daten zu parallelisieren. Diese beiden Teilaufgaben stecken bei der parallelen Pipeline mit in der durch die TBB parallelisierten Pipeline. Eine Parallelisierung dieser beiden Aufgaben ist jedoch unlogisch, es handelt sich dabei sicher nur um einen sehr viel effizienter organisierten Zugriff auf das Speichermedium Festplatte.

Der theoretische SpeedUp für diesen Algorithmus ist in der Tabelle 6-12 dargestellt. Die Tabelle zeigt zwei Datenzeilen, eine unter Berücksichtigung der Festplattenoperationen und eine ohne diese. Die ermittelten Werte stammen aus dem Kapitel 4 auf Seite 25 und wurden mithilfe der Formel 9 bestimmt. Die Spalte „SpeedUp max“ zeigt an, welcher SpeedUp maximal bei der Verwendung von unendlich vielen Prozessoren zu erwarten ist. Die weiteren Spalten zeigen den erwarteten SpeedUp bei zwei bzw. vier verwendeten Prozessoren. Bei einem Vergleich der Tabelle 6-7, Tabelle 6-8, Tabelle 6-10 und Tabelle 6-11 mit der Tabelle 6-12 ist festzustellen, dass die Pipeline-Implementierungen den erwarteten SpeedUp für ihr System Größenteils erfüllt haben und sogar nahe an den erwarteten SpeedUp, ohne Berücksichtigung der Festplattenoperationen, herankommen.

Festplatten-operationen	*SpeedUp max*	*SpeedUp p=2 (TC1)*	*SpeedUp p=4 (TC2)*
ohne	10	1,818182	3,076923
mit	2,631579	1,449275	1,869159

Tabelle 6-12: Theoretischer SpeedUp des Photo2Scan-Algorithmus

Eine Problematik, die sich auch wiederum aus dem ermittelten Werten für das Verhältnis der benötigten Zeiten erkennen lässt, ist, dass schon alleine die Verwendung des Entwurfsmusters der Pipeline zu einer deutlichen Verschiebung der Verhältnisse der benötigten Zeiten führt. Mit einer Steigerung der Anzahl der Prozessoren, die an den Berechnungen teilnehmen, ließ sich die Berechnung an sich nur noch minimal steigern.

Abschließend bleibt nur noch festzustellen, dass die Erwartungen an die parallelisierten Implementationen erfüllt wurden und zum Teil die Ergebnisse die Erwartungen deutlich positiv übertroffen haben.

7 Schlussfolgerung

Während dieser Masterthesis war es das Ziel, Algorithmen, die auch in der Praxis eingesetzt werden, hinsichtlich ihrer Parallelisierbarkeit zu untersuchen. Die zwei dafür ausgewählten Algorithmen wurden in einer komplett sequenziellen Implementation von der Firma Lupos3D eingesetzt. Die Algorithmen waren dabei sehr unterschiedlich. Während der Schnitte-Algorithmus in seiner Komplexität sehr einfach und rechenintensiv war, in Bezug auf den Anteil der Berechnungen an der Gesamtlaufzeit des Algorithmus. Handelte es sich bei Photo2Scan-Algorithmus um einen komplexeren Algorithmus, der jedoch stark von den Lese- und Schreiboperationen auf der Festplatte beeinflusst war. Die Voruntersuchungen im Kapitel 4 ließen darauf schließen, dass eigentlich nur der Schnitte-Algorithmus für eine Parallelisierung geeignet war. Der Schnitte-Algorithmus benötigte 90 Prozent seiner Gesamtlaufzeit nur für Rechenoperationen (parallelisierbarer Anteil) und war daher prädestiniert für eine Parallelisierung. Das Gesetz von Gene Amdahl, über die Beschleunigung von Computerprogrammen durch eine parallele Ausführung, bestätigte dem Algorithmus eine maximale Beschleunigung um den Faktor 10 bei unendlich vielen verwendeten Prozessoren. Durch die Berechnung des theoretischen SpeedUp des Algorithmus, für einen Zwei-Kern und einen Vier-Kern Computer, zeigte sich ein lohnenswertes Potenzial des Algorithmus. Die theoretische mögliche Beschleunigung des Algorithmus lag bei dem Faktor 1,81 für den Zwei-Kern Computer und dem Faktor 3,07 für den Vier-Kern Computer. Der zweite Algorithmus, Photo2Scan, der vor den Voruntersuchungen, aufgrund seiner im Vergleich zum Schnitte-Algorithmus langen Laufzeit als sehr vielversprechend galt, enttäuschte leider im Kapitel 4. Der Anteil der parallelisierbaren Operationen an der Gesamtlaufzeit des Algorithmus betrug lediglich 62 Prozent. Aus diesem Anteil ergibt sich, nach Amdahl, wiederum nur ein kleiner Faktor (2,64) für die theoretisch mögliche maximale Beschleunigung. Bei einem so geringen Faktor ist generell nicht von einem großen Nutzen der Parallelisierung auszugehen. Zumal der Faktor nur mit der theoretischen Anzahl von unendlich vielen Prozessoren erreicht werden kann. Die reale Beschleunigung bei Computern mit einer handelsüblichen Anzahl von Prozessoren (2, 4, 6-Kerne) fällt wesentlich geringer aus. Werden die Lese- und Schreiboperationen jedoch aus der Laufzeit des Photo2Scan-Algorithmus ausgeklammert, ergibt sich eine andere Verteilung der parallelisierbaren Operationen. Dann beträgt der Anteil wie beim Schnitte-Algorithmus 90 Prozent. Die Ausklammerung der Lese- und Schreiboperationen aus der Messung kann unter der Voraussetzung vorgenommen werden, da wesentlich schnellere Medien zur Speicherung von Daten existieren als magnetische Festplatten. Durch die Annahme, dass die Massenspeichermedien in der Zukunft wesentlich schneller werden (z. B.: SSD[55]), ist eine Untersuchung der Parallelisierbarkeit durchaus interessant.

Die Implementierung der verwendeten Algorithmen geschah auf unterschiedlichen Arten (siehe Kapitel 5 auf Seite 29). Je nachdem, welche Möglichkeit der Parallelisierung der jeweilige Algorithmus bot. Der Schnitte-Algorithmus war so gestaltet, dass viele kleine Rechenoperationen ausgeführt werden mussten. Auch die Reihenfolge der Berechnungen spielte für das Ergebnis keine Rolle. Aufgrund dieser Struktur bot sich eine Parallelisierung mittels Threads an. Um Vergleiche vornehmen zu können, wurden eine sequenzielle und zwei parallele Varianten des Algorithmus entwickelt. Die parallelen Implementierungen nutzen dabei einmal die Windows-Threads aus den Microsoft Foundation Classes (MFC) und die POSIX-Threads für Windows. Der zweite Algorithmus aus der Praxis (Photo2Scan) hatte eine andere Struktur als der Schnitte-Algorithmus. Die Berechnungen waren wesentlich Komplexer und die Reihenfolge musste exakt eingehalten werden. Die Berechnung eines Datensatzes setzte sich aus unterschiedlichen Teilschritten zusammen, die jeweils auf der

[55] Solid State Drive

vorherigen Berechnung beruhten. Daher wurde die Parallelisierung dieses Algorithmus mittels des Entwurfsmusters der Pipeline durchgeführt. Zum Vergleich wurde auch wieder eine rein sequenzielle Implementierung vorgenommen. Das Muster der Pipeline wurde auf zwei unterschiedliche Arten implementiert. Zum einen handelte es sich um eine sequenzielle Pipeline das bedeutet, es wurde lediglich das Entwurfsmuster implementiert. Die Pipeline wurde nur mittels eines Threads auf einem Prozessor durchgeführt. Die zweite parallele Implementation wurde mit der Intel Bibliothek TBB[56] vorgenommen. Das Ergebnis der zweiten Implementation war eine parallel ausgeführte Pipeline. Dabei wurden die unterschiedlichen Berechnungen auf unterschiedliche Prozessoren verteilt. Wenn das Programm bemerkte, das ein Prozessor Zuviel und ein anderer Zuwenig Aufgaben hatte, wurde eine automatische Umverteilung vorgenommen.

Für die Messungen wurden Eingangsdaten von unterschiedlicher Größe verwendet. Bei den Daten handelte es sich um einen kleinen Laserscan mit 4.837.296 Messpunkten und einen großen Laserscan 43.554.392 Messpunkten. Jeder implementierte Algorithmus, sequenziell und parallel, wurde mit beiden Datensätzen getestet. Für jeden Algorithmus wurde mit jedem der beiden Eingangsdatensätze eine Messreihe durchgeführt. Eine Messreihe bestand aus 51 Durchläufen des jeweiligen Algorithmus. Für die Auswertung wurde später der erste Durchlauf aus der Messreihe gestrichen, da bei diesem das entsprechende Programm zuerst geladen wurde. Bei den folgenden Durchläufen befand sich das Programm schon im Hauptspeicher. Der erste Durchlauf war somit ein Ausreißer, der durch betriebssystembedingte Faktoren so stark beeinflusst wurde, dass keine repräsentativen Aussagen aus der Messung abgeleitet werden konnten.

Für den Schnitte-Algorithmus ergaben die Messungen auf dem TC1[57] die erwarteten Ergebnisse. Der Ermittelte reale SpeedUp für die Windows-Threads (MFC) und auch die POSIX-Threads, lag gerundet bei dem Faktor 1,4. Erwartet wurde bei zwei Prozessor-Kernen ein theoretischer Faktor von 1,8. Aufgrund von Hintergrundprozessen anderer Programm und Verwaltungsaufgaben des Betriebssystems konnte der theoretische Wert nicht ganz erreicht werden. Aus den Ergebnissen der Messungen konnte ebenfalls abgeleitet werden, dass die Windows-Threads nahezu genauso effizient arbeiten wie die POSIX-Threads. Der Unterschied zwischen den Laufzeiten der parallelisierten Programme war minimal, gerade einmal rund 750ms bei einer Gesamtlaufzeit von rund 52sek. Dennoch waren die POSIX diesen Sekundenbruchteil schneller als die Windows-Threads. Für den TC2[58] verhielt sich der Schnitte-Algorithmus gänzlich unerwartet. Die Ergebnisse der Messungen zeigten, eine Verlangsamung des Algorithmus durch die Parallelisierung. Die parallelisierten Implementationen benötigen fast doppelt so lange wie die sequenzielle Implementation. Eine erste Theorie, warum dieses Phänomen auftrat, war eine unterschiedliche Laufzeitumgebung. Dies konnte jedoch ausgeschlossen werden, da es sich bei beiden Testcomputern um x86-System mit Windows XP handelte. Vor den Messungen wurden zudem alle von Microsoft zur Verfügung gestellten Updates auf beiden Rechnern installiert und auch die gleiche Run-Time-Engine für C++ benutzt. Die zweite Theorie war, dass die Datenmengen und das Programm, bei einer parallelen Verarbeitung nicht in den von den Prozessor-Kernen gemeinsam genutzten Cache passten. Aber die im Kapitel 6.1.2 mittels der Formel 22 berechneten Größen des Programmes, mit Daten, zeigte, dass auch diese Theorie eher ausgeschlossen werden konnte. Die letzte Theorie war, dass die von Lupos3D zur Verfügung gestellte DLL-Datei schlecht erstellt worden ist. Bei der Ergebnisauswertung ergab sich der Eindruck als erlaube die DLL nur einen exklusiven Zugriff das bedeutet, nur ein Thread kann

[56] Threading Building Blocks
[57] Dual-Core (2GHz), 2048MB RAM; 1x4MB L2-Cache (siehe auch Tabelle 5-2 auf Seite 30)
[58] Quad-Core (2,3GHz); 4096MB RAM; 2x2MB L2-Cache (siehe auch Tabelle 5-3 auf Seite 31)

gleichzeitig die Funktionen der DLL nutzen. Dennoch konnte aufgezeigt werden, dass eine Parallelisierung des Algorithmus nicht nur theoretisch möglich ist, sondern auch praktisch die realisierbar ist.

Die Ergebnisse des zweiten Algorithmus, Photo2Scan, verhielten sich wie erwartet (siehe Kapitel 6.2). Es wurde deutlich, dass der Geschwindigkeitsvorteil mit einer Zunahme des Grades an Parallelität zunahm. Die ermittelten Werte für den SpeedUp, der beider Testcomputer, kamen bei der Berücksichtigung der Festplattenoperationen nicht an die ermittelten Werte des theoretischen SpeedUp heran. Wenn die Festplattenoperationen, wie in Kapitel 4 erläutert, außen vorgelassen werden, ergaben sich reale SpeedUp Werte, die jenseits aller Erwartungen lagen. Alleine durch die Umstrukturierung der Berechnung in das Entwurfsmuster der Pipeline, konnte die Berechnung beim TC1 um den Faktor 9,3 und beim TC2 um den Faktor 9,1 beschleunigt werden. Die Beschleunigung der Berechnung konnte jedoch, durch die Verwendung der Bibliothek TBB, noch weiter gesteigert werden. Die Berechnung im Entwurfsmuster der Pipeline, verteilt auf mehrere Prozessoren bewirkte eine Beschleunigung um den Faktor 22,4 beim TC1 und den Faktor 9,7 beim TC2, im Vergleich zur komplett sequenziellen Implementation. Die Parallelisierung des Photo2Scan-Algorithmus ist somit sinnvoll, wenn die entsprechend schnelle Hardware zum Lesen und Schreiben von Daten auf ein Massenspeichermedium zur Verfügung steht. Ist eine solche Hardware nicht vorhanden, können dennoch Beschleunigungen durch die Parallelisierung erreicht werden. Jedoch ist in diesem Fall mit einem sehr starken Rückgang des Geschwindigkeitsvorteiles zu rechnen.

7.1 Bewertung

In diesem Abschnitt möchte ich meine Eindrücke zu den verwendeten Techniken und Bibliotheken darstellen und auch etwas Kritik üben.

Die beim Schnitte-Algorithmus eingesetzten Bibliotheken[59] für die Parallelisierung waren in der Implementation recht ähnlich. Beide boten ausreichende Funktionen zur Steuerung, Verwaltung und vor allem zur Synchronisierung von Threads. Beide eignen sich daher sowohl für kleine als auch große Parallelisierungsprojekte. Die Windows-Threads sind jedoch nur ein kleiner Bestandteil einer großen Klassensammlung von Microsoft. Die Bibliothek ist daher für einen Einsteiger in die Parallelisierung nicht der beste Anfang, da sie doch recht Komplex ist. Die Dokumentation der Funktionen ist, wenn die richtige Stelle gefunden wurde, umfangreich, informativ, gut verständlich und zum Teil auch mit Beispielen versehen. Ein sehr großer Nachteil ist jedoch, dass die MFC-Klassensammlung nicht Plattform unabhängig ist und des Weiteren auch nur in der kostenpflichtigen Version des Microsoft Visual Studio erhältlich. Die POSIX-Threads sind dahingegen ein Open Source Produkt, welches auch Plattform unabhängig ist. Bei dieser Bibliothek fanden sich auch wesentlich mehr Beispiel, Vorträge und Foren-Beiträge, welche einen Einstieg in die Bibliothek wesentlich einfacher gestalten als bei den Windows-Threads.

Die bei dem Photo2Scan eingesetzte Technik der Entwurfsmuster ist in ihrer Gesamtheit sehr Komplex und nicht immer leicht verständlich. In unterschiedlicher Literatur sind jedoch immer wieder Erläuterungen und Implementationsbeispiele zu finden, die sich auch auf die eigenen Implementationsprobleme anwenden lassen. Eine Strukturierung des Programmcodes mittels Entwurfsmuster ist immer sehr sinnvoll und erleichtert die spätere Weiterentwicklung von Software enorm. Das bei dem Algorithmus verwendete Muster der Pipeline bot zu dem sogar in der sequenziellen Implementation eine Beschleunigung der Berechnungen. Die auch bei diesem Algorithmus verwendete Bibliothek TBB, war für mich das persönliche Highlight der Masterthesis. Die

[59] POSIX-Threads und Windows-Threads (MFC)

Bibliothek bietet dem Entwickler eine Unsumme von parallelen Möglichkeiten bei nur geringen Implementationsaufwand. Die Bibliothek existiert in zwei Varianten, einer frei verfügbaren und einer kostenpflichtigen. Die kostenpflichtige Variante bietet die gleichen Funktionen, wie die kostenlose Variante, aber es existiert ein besserer Support. Gerade dieser Support ist auch die Kritik an der Bibliothek, denn diese ist sehr kompliziert in ihrer Anwendung und daher für einen schnellen Einstieg gar nicht geeignet.

7.2 Aussichten

Die Entwicklung von Techniken und Werkzeugen zur Parallelisierung bleibt nicht stehen. Daher ist es unvermeidbar, dass nicht alle Möglichkeiten in dieser Masterthesis behandelt werden können. Während der Recherche zu der Thematik und auch während des Schreibens der Thesis, stieß ich auf weitere Möglichkeiten, denen ich mich gerne auch noch gewidmet hätte.

Für die beiden parallelisierten Algorithmen können noch weitere Bibliotheken getestet werden, die ebenfalls eine Parallelisierung mittels Threads erlauben. Dadurch könnten, vor allem auf der Basis des Schnitte-Algorithmus, Vergleiche hinsichtlich der Effizienz der verschiedenen Bibliotheken durchgeführt werden. Insbesondere die Boost-Bibliothek scheint eine sehr vielversprechende Multithreading-Bibliothek zu sein. Boost ist ein plattformunabhängiges Open Source Produkt, dessen Teilbibliotheken sogar in den C++-Standard übernommen wurden[60]. Des Weiteren bietet sich ein weiteres Entwurfsmuster an, das Master/Worker Prinzip[61]. Scheint sowohl für den Schnitte-Algorithmus, als auch für den Photo2Scan-Algorithmus interessant. Mittels mehreren Threads ließen sich die einzelnen Zeilen der Laserscans optimaler auf die Prozessoren verteilen. Beim Photo2Scan-Algorithmus ließe sich das Master/Worker-Prinzip sogar noch mit dem Entwurfsmuster der Pipeline kombinieren.

Ein weiterer Punkt, welcher einer Nachbesserung bedarf, ist die DLL von Lupos3D. So simpel die Anwendung der Funktionen der DLL auch scheint, ist sie dennoch nicht gut programmiert. Die Ebene, die für die Berechnung des Schnitte-Algorithmus notwendig ist, wird für jede Datenzeile des Laserscans neu berechnet, obwohl sie sich nicht ändert. Die Technik der Laserscanner ermöglicht die Erfassung von Millionen von Messpunkten in sehr kurzer Zeit. Bei dieser Funktion werden unzählige Berechnungen, für die mehrfache Berechnung einer Ebene verschwendet, die sich während der gesamten Berechnung des kompletten Laserscans nicht ändert. Es bietet sich daher an die Berechnung der Ebene aus drei Koordinaten, wie in Kapitel 2.2 beschrieben, neu zu implementieren. Dadurch ließen sich höchstwahrscheinlich auch die Problematik bei der Berechnung[62] eliminieren und somit auch bessere Ergebnisse bei der Beschleunigung erreichen.

[60] Vgl. (34) – Kapitel 1 und Kapitel 6
[61] Siehe Kapitel 3.2.2.3 auf Seite 16
[62] Siehe Kapitel 6.1 auf Seite68

Literatur

1. **Wikipedia.** Hessesche Normalform. *Wikipedia.* [Online] Wikimedia Foundation Inc., 19. November 2010. [Zitat vom: 01. Januar 2011.] http://de.wikipedia.org/wiki/Hessesche_Normalform.

2. **Luhmann, Thomas.** *Nahbereichsphotogrammetrie.* 2 Ausg. Heidelberg : Wichmann, 2003. ISBN: 3-87907-398-3.

3. **Wehrenpfennig, Prof. Dr.-Ing. Andreas.** *Grundlagen der Informatik: Rechnerarchitektur.* Fachbereich LGGB. Neubrandenburg : Hochschule Neubrandenburg, 2004. [Vorlesung].

4. —. *Grundlagen der Informatik.* Fachbereich LGGB. Neubrandenburg : Hochschule Neubrandenburg, 2004. [Vorlesung].

5. **Wikipedia.** Prozessor (Hardware). *Wikipedia.* [Online] Wikimedia Foundation Inc., 11. August 2010. [Zitat vom: 12. August 2010.] http://de.wikipedia.org/wiki/Prozessor_%28Hardware%29.

6. —. Mehrkernprozessor. *Wikipedia.* [Online] Wikimedia Foundation Inc., 2. August 2010. [Zitat vom: 12. August 2010.] http://de.wikipedia.org/wiki/Mehrkern-Prozessor.

7. **Roloff, Sascha.** Multicore-Architekturen. *Kurs 1: Programmierkonzepte für Multi-Core Rechner.* [PowerPoint]. Reinswald : Technische Universität München, 29. September 2009.

8. **Bauke, Heiko und Mertens, Stephan.** *Cluster Computing.* Berlin : Springer, 2006. ISBN: 3-54042-299-4.

9. **Abi-Chahla, Fedy.** Nvidia Cuda: Das Ende der CPU? *Tom's Hardware.* [Online] TG Publishing AG, 18. Juni 2008. [Zitat vom: 17. August 2010.] http://www.tomshardware.de/CUDA-Nvidia-CPU-GPU,testberichte-240065.html.

10. **Gamma, Erich.** *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Bonn : Addison-Wesley, 1996. ISBN: 3-89319-950-0.

11. **Müller, Udo.** *C++ - Implementierungstechniken.* Bonn : International Thomson Publishing, 1997. ISBN: 3-8266-0254-4.

12. **Herold, Helmut, Klar, Michael und Klar, Susanne.** *C++, UML und Design Patterns: Grundlagen und Praxis der Objektorientierung.* München : Addison-Wesley, 2005. ISBN: 3-8273-2267-7.

13. **Siu, S., et al.** *Design Patterns for Parallel Programming.* Dept. of Electrical & Computer Engineering, University of Waterloo. Waterloo, Ontario, Kanada : s.n.

14. **Siu, Stephen und Singh, Ajit.** *Composing Parallel Applications Using Design Patterns.* Dept. of Electrical & Computer Engineering, University of Waterloo. Waterloo, Ontario, Kanada : s.n.

15. **Kim, Eun-Gyn.** *Parallel Programming Patterns.* [PowerPoint] Urbana : University of Illinois, 10. Juni 2004. http://www.cs.uiuc.edu/homes/snir/PPP/.

16. **c't.** extra. *Programmieren.* Hannover : Heise Zeitschriften Verlag, Februar 2009.

17. **Georgieva, Antoniya und Kuhlmeyer, Thomas.** *Paradigmen für paralleles Programmieren.* [PowerPoint] Berlin : Zuse Insitut, 26. Mai 2008. www.zib.de/andrzejak/lehre/para08/folien/prog-2.ppt.

18. **Wikipedia.** Zuständigkeitskette. *Wikipedia.* [Online] Wikimedia Foundation Inc., 5. März 2010. [Zitat vom: 21. Juli 2010.] http://de.wikipedia.org/wiki/Zust%C3%A4ndigkeitskette.

19. **Quinn, Michael Jay.** *Parallel Programming: in C with MPI and OpenMP.* New York : McGraw-Hill, 2003. ISBN: 007-123265-6.

20. **Wikipedia.** Message Passing Interface. *Wikipedia.* [Online] Wikimedia Foundation Inc., 10. Juli 2010. [Zitat vom: 10. August 2010.] http://de.wikipedia.org/wiki/Message_Passing_Interface.

21. **Pluemicke, Prof. Dr.** Userlevel und Kernel-Threads. [Online] Duale Hochschule Baden-Württemberg, 10. Mai 2003. [Zitat vom: 12. August 2010.] http://www.ba-horb.de/~pl/ISBSMI2001/html/node50.html.

22. **Wikipedia.** Thread (Informatik). *Wikipedia.* [Online] Wikimedia Fundation Inc., 29. Juli 2010. [Zitat vom: 12. August 2010.] http://de.wikipedia.org/wiki/Thread_%28Informatik%29.

23. **Pankratius, Dr. Victor, Tichy, Prof. Walter F. und Otto, Dipl.-Inform. Frank.** *Parallelität in C/C++.* Fakultät für Informatik. Karlsruhe : Universität Karlsruhe.

24. **Wikipedia.** Microsoft Foundation Classes. *Wikipedia.* [Online] Wikimedia Inc., 11. November 2010. [Zitat vom: 30. Januar 2011.] http://de.wikipedia.org/wiki/Microsoft_Foundation_Classes.

25. **Microsoft.** _beginthread, _beginthreadex. *Microsoft Developer Network (MSDN).* [Online] Microsoft Corporation, 2011. [Zitat vom: 30. 01 2011.] http://msdn.microsoft.com/de-de/library/kdzttdcb%28v=VS.90%29.aspx.

26. **Wikipedia.** Intel Threading Building Blocks. *Wikipedia.* [Online] Wikimedia Foundation Inc., 5. Juni 2010. [Zitat vom: 13. Juli 2010.] http://en.wikipedia.org/wiki/Intel_Threading_Building_Blocks.

27. **Riemann, Mario.** Napoleon: Total War - Better performance coming with the Empire add-on? . *PC Games Hardware.* [Online] Computec Media AG, 24. November 2009. [Zitat vom: 13. Juli 2010.] http://www.pcgameshardware.com/aid,700063/Napoleon-Total-War-Better-performance-coming-with-the-Empire-add-on/News/.

28. **Intel.** *Threading Building Blocks - Tutorial.* 2009. Dokumentnummer: 319872-003US.

29. **Wikipedia.** Cache. *Wikipedia.* [Online] Wikimedia Foundation Inc., 20. Juli 2010. [Zitat vom: 03. August 2010.] http://de.wikipedia.org/wiki/Cache.

30. **Microsoft.** MFC Reference. *Microsoft Developer Network (MSDN).* [Online] Microsoft Corporation, 2011. [Zitat vom: 30. 01 2011.] http://msdn.microsoft.com/de-de/library/d06h2x6e%28v=VS.90%29.aspx.

31. **Keßeler, Florian.** Tutorial: Mehrdimensionale Arrays in C++. *Developia.* [Online] SOFTGAMES - Mobile Entertainment Services GmbH , 09. Februar 2005. [Zitat vom: 21. August 2010.] http://www.developia.de/developia/viewarticle.php?cid=19547&page=0.

32. **Kraus, Karl.** *Photogrammetrie - Band 1.* 6 Ausg. Bonn : Dümmler, 1997. ISBN: 3-427-78646-3.

33. **Prümm, Olaf und Pospiš, Michael.** *PTB – Ein Vorschlag für ein einheitliches binäres.* [PowerPoint] Bochum, Deutschland : Lupos3D, 02. September 2008.

34. **Schäling, Boris.** Die Boost C++ Bibliotheken. *Highscore.* [Online] 03. Februar 2010. [Zitat vom: 16. Februar 2011.] http://www.highscore.de/cpp/boost/.

35. **Stokes, Jon.** Understanding CPU caching and performance. *ars technica.* [Online] Ars Technica, 09. Juni 2002. [Zitat vom: 09. Januar 2011.] http://arstechnica.com/gadgets/reviews/2002/07/caching.ars.

Anhang

Messwerte Schnitte-Algorithmus TC1 – sequenziell

Small Scan

Time in [ms]

Total	Initial	Read	Calc	Write
8109	15	77	7685	332
8125	0	125	7721	279
8094	0	93	7626	375
8141	0	79	7796	266
8140	15	155	7705	265
8141	0	125	7771	245
8141	0	109	7841	191
8110	0	187	7736	187
8094	0	62	7717	315
8141	0	48	7751	342
8125	15	155	7675	280
8172	0	109	7747	316
8141	0	95	7731	315
8110	0	140	7703	267
8156	0	31	7764	361
8140	15	79	7858	188
8093	15	93	7704	281
8172	0	79	7827	266
8110	0	78	7827	205
8141	0	110	7654	377
8156	0	174	7748	234
8094	0	159	7542	393
8094	0	140	7751	203
8125	0	110	7750	265
8125	15	31	7765	314
8125	0	189	7593	343
8125	0	64	7812	249
8110	0	62	7663	385
8156	15	46	7781	314
8094	0	80	7672	342
8078	0	187	7597	294
8109	15	108	7689	297
8125	0	155	7516	454
8156	15	128	7719	294
8110	0	63	7766	281
8156	0	77	7799	280
8156	0	110	7811	235
8141	0	47	7861	233

Big Scan

Time in [ms]

Total	Initial	Read	Calc	Write
74703	0	812	70952	2939
74625	0	900	71204	2521
74984	15	781	71219	2969
75031	0	907	71343	2765
74860	0	952	71563	2345
74953	15	952	71581	2405
75031	0	748	71682	2585
75078	0	781	71676	2621
74625	0	764	70957	2904
74750	0	1027	70954	2769
74921	15	831	71527	2548
74765	15	733	71508	2509
74891	16	969	71361	2545
74968	15	797	71589	2567
74797	0	904	71260	2633
74906	16	1000	71115	2775
74953	0	888	71311	2754
74922	0	969	71267	2686
74750	0	878	71362	2510
74860	0	811	71400	2649
74766	0	987	71028	2751
75078	16	934	71414	2714
74813	16	771	71408	2618
75093	15	1035	71349	2694
74968	15	769	71531	2653
74969	0	872	71421	2676
74969	16	1070	71545	2338
75031	0	707	71678	2646
74735	0	893	71157	2669
75016	16	889	71743	2368
74937	15	935	71366	2621
74844	0	900	71056	2872
74860	0	841	71561	2458
74875	15	766	71482	2612
74969	0	855	71549	2565
74781	15	869	71095	2802
75000	15	950	71431	2604
74875	0	602	71565	2708

8125	0	127	7779	219
8141	0	61	7590	490
8172	0	79	7766	327
8140	15	157	7626	342
8125	15	47	7781	282
8063	0	125	7644	294
8188	0	109	7737	342
8109	15	111	7747	236
8110	0	78	7658	374
8172	0	127	7844	201
8110	0	32	7842	236
8078	15	93	7675	295
8127,3	**3,9**	**102,1**	**7727**	**294,02**

74921	15	753	71672	2481
74891	0	809	71489	2593
75109	15	733	71545	2816
75015	15	866	71911	2223
74875	0	961	71514	2400
75063	0	1018	71327	2718
74984	15	688	71685	2580
75015	15	793	71339	2868
74859	15	763	71379	2702
74796	15	735	71533	2513
74985	0	794	71634	2557
74891	0	863	71544	2484
74907	**7,32**	**856,5**	**71416**	**2626,06**

Messwerte Schnitte-Algorithmus TC1 – MFC-Threads – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Tread 1 Lock2Unlock	Tread 1 Write2Unlock	Tread 1 Lock2Write	Tread 2 Total	Tread 2 Initial	Tread 2 Read	Tread 2 Calc	Tread 2 Lock2Unlock	Tread 2 Write2Unlock	Tread 2 Lock2Write
5813	16	5797	0	110	5311	376	361	15	5781	0	16	5658	107	91	16
5813	0	5781	0	16	5687	78	78	0	5813	0	32	5657	124	108	16
5797	0	5781	0	46	5564	171	155	16	5797	0	77	5550	170	140	30
5781	15	5750	0	108	5267	375	375	0	5766	0	79	5608	79	47	32
5719	0	5719	0	15	5501	203	203	0	5703	0	64	5436	203	203	0
5781	0	5781	0	63	5484	234	234	0	5766	0	48	5548	170	139	31
5797	0	5781	0	96	5544	141	125	16	5797	0	63	5515	219	188	31
5797	0	5781	0	78	5451	252	236	16	5797	0	47	5594	156	141	15
5750	0	5750	0	77	5469	204	204	0	5719	0	78	5453	188	172	16
5750	0	5735	0	16	5547	172	172	0	5750	0	48	5577	125	110	15
5797	0	5781	0	15	5591	175	175	0	5797	0	63	5658	76	76	0
5797	0	5797	0	30	5579	188	188	0	5797	0	15	5607	175	143	32
5766	0	5766	0	79	5563	124	108	16	5766	0	48	5609	109	109	0
5766	16	5750	0	32	5640	78	78	0	5750	0	80	5546	124	109	15
5719	0	5719	0	31	5561	127	111	16	5719	0	79	5311	329	298	31
5781	0	5766	0	15	5656	95	95	0	5781	0	110	5515	156	125	31
5750	0	5735	0	31	5565	139	139	0	5750	0	95	5451	204	189	15
5796	15	5781	0	32	5685	64	64	0	5781	0	93	5452	236	236	0
5797	16	5765	0	32	5577	156	141	15	5781	0	46	5596	139	139	0
5766	16	5750	0	47	5296	407	375	32	5750	0	47	5450	253	221	32
5781	0	5781	0	125	5281	375	344	31	5781	0	16	5578	187	187	0

5781	0	5766	0	31	5593	142	142	0	5781	0	31	5421	329	266	63
5813	16	5781	0	0	5594	187	187	0	5797	0	47	5623	127	95	32
5766	0	5766	0	31	5610	125	110	15	5766	0	94	5501	171	156	15
5719	0	5719	0	31	5456	232	232	0	5719	0	32	5486	201	185	16
5766	0	5766	0	110	5469	187	171	16	5766	0	47	5547	172	172	0
5750	0	5750	0	15	5547	188	172	16	5750	0	95	5466	189	173	16
5797	0	5781	0	64	5605	112	112	0	5797	0	47	5488	262	230	32
5766	0	5766	0	61	5517	188	173	15	5750	0	62	5516	172	172	0
5766	0	5766	0	94	5329	343	343	0	5735	0	0	5500	235	219	16
5797	0	5781	0	92	5453	236	236	0	5797	0	47	5579	171	171	0
5797	0	5797	0	63	5533	201	170	31	5797	0	31	5499	267	236	31
5781	15	5766	0	96	5388	282	266	16	5766	0	47	5626	93	93	0
5735	0	5735	0	48	5389	298	298	0	5719	0	47	5406	266	234	32
5781	0	5781	0	0	5532	249	249	0	5766	0	110	5438	218	186	32
5797	0	5797	0	47	5672	78	78	0	5781	0	80	5515	186	186	0
5766	0	5750	0	61	5487	202	202	0	5766	0	32	5563	171	155	16
5781	15	5766	0	46	5564	156	125	31	5750	0	32	5592	126	126	0
5797	0	5797	0	94	5532	171	171	0	5781	0	31	5596	154	139	15
5750	0	5750	0	16	5624	110	110	0	5750	0	64	5547	139	109	30
5813	0	5813	0	79	5500	234	234	0	5813	0	32	5627	154	122	32
5766	0	5766	0	79	5529	158	158	0	5766	0	31	5562	173	141	32
5781	0	5781	0	16	5515	250	235	15	5781	0	0	5640	141	126	15
5796	15	5766	0	47	5611	108	108	0	5781	0	60	5548	173	173	0
5953	15	5938	0	15	5734	189	142	47	5860	0	16	5703	141	94	47
5734	15	5719	0	64	5469	186	171	15	5719	0	76	5537	106	91	15
5781	0	5766	0	62	5596	108	108	0	5781	0	79	5498	204	204	0
5781	0	5781	16	0	5563	202	202	0	5765	0	48	5561	156	125	31
5781	15	5750	0	47	5483	220	220	0	5766	0	30	5438	298	282	16
5719	0	5703	0	48	5469	186	186	0	5719	0	79	5516	124	108	16

5778,5	**4**	**5768,22**	**0,32**	**51,02**	**5523,64**	**193,24**	**185,44**	**7,8**	**5768,54**	**0**	**53,42**	**5538,16**	**176,96**	**158,8**	**18,16**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
5778,5	4	5768,54	0,32	53,42	5538,16	193,24	185,44	18,16

Messwerte Schnitte-Algorithmus TC1 – MFC-Threads – großer Scan

Big Scan

Time in [ms]

Total	Initial	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Tread 1 Lock2Unlock	Tread 1 Write2Unlock	Tread 1 Lock2Write	Tread 2 Total	Tread 2 Initial	Tread 2 Read	Tread 2 Calc	Tread 2 Lock2Unlock	Tread 2 Write2Unlock	Tread 2 Lock2Write
52937	15	52922	0	676	50569	1677	1539	138	52891	0	560	50688	1643	1392	251
52250	0	52250	0	594	50032	1624	1501	123	52203	0	375	49986	1842	1698	144
52703	15	52688	0	565	50430	1693	1598	95	52641	0	485	50309	1847	1625	222
52922	0	52922	0	672	50393	1857	1734	123	52875	0	529	50357	1989	1740	249
53000	15	52969	0	538	50808	1623	1484	139	52969	0	482	50841	1646	1375	271
52703	15	52688	0	424	50866	1398	1350	48	52641	0	535	50676	1430	1306	124
52469	0	52469	0	346	50415	1708	1596	112	52422	0	400	50297	1725	1554	171
53594	0	53594	0	486	51690	1418	1295	123	53594	0	377	51393	1824	1608	216
53000	16	52953	0	424	50832	1697	1601	96	52984	0	374	50836	1774	1600	174
52750	0	52750	0	252	50723	1775	1618	157	52625	0	501	50427	1697	1524	173
52391	0	52391	0	420	50153	1818	1773	45	52344	0	406	50326	1612	1425	187
52343	15	52328	0	435	50586	1307	1262	45	52297	0	477	50196	1624	1438	186
52906	0	52906	0	377	50804	1725	1584	141	52875	0	423	50649	1803	1644	159

53250	0	53250	0	501	51228	1521	1380	141	53203	0	578	51025	1600	1441	159
52438	0	52438	0	457	50345	1636	1514	122	52375	0	455	50084	1836	1696	140
52781	0	52781	0	407	50910	1464	1400	64	52766	0	438	51106	1222	1054	168
52453	0	52453	0	419	50348	1686	1595	91	52422	0	708	49853	1861	1677	184
52500	0	52500	0	313	50484	1703	1593	110	52453	0	548	50380	1525	1323	202
52469	0	52469	0	486	50342	1641	1534	107	52375	0	376	50385	1614	1457	157
53672	0	53641	0	312	51624	1705	1502	203	53672	0	416	51454	1802	1612	190
53140	15	53125	0	424	51217	1484	1328	156	53078	0	466	50999	1613	1364	249
52593	15	52578	0	545	50540	1493	1382	111	52453	0	417	50536	1500	1389	111
52453	15	52438	0	562	50214	1662	1536	126	52391	0	548	50425	1418	1250	168
52625	15	52610	0	328	50819	1463	1399	64	52547	0	546	50334	1667	1482	185
52563	16	52547	0	438	50492	1617	1522	95	52484	0	687	50001	1780	1654	126
52656	0	52656	0	407	50670	1579	1470	109	52594	0	359	50421	1814	1546	268
53063	0	53063	0	466	51051	1546	1420	126	53047	0	496	50801	1750	1591	159
52718	15	52703	0	356	50891	1456	1379	77	52656	0	456	50498	1702	1495	207
52906	0	52906	0	458	51106	1342	1218	124	52875	0	362	50896	1617	1351	266
52296	15	52281	0	409	50184	1688	1593	95	52219	0	422	50190	1591	1392	199
52234	15	52219	0	421	49951	1847	1783	64	52172	0	406	50095	1671	1518	153
52718	15	52703	0	357	50777	1569	1457	112	52656	0	709	50192	1755	1580	175
53156	0	53156	0	387	51127	1642	1534	108	53000	0	378	51279	1343	1200	143
52563	0	52563	0	372	50489	1702	1608	94	52516	0	451	50180	1869	1701	168
53047	16	53031	0	452	50769	1810	1653	157	53000	0	480	50852	1668	1516	152
52875	15	52860	0	501	50755	1604	1540	64	52860	0	391	50611	1858	1603	255
52797	0	52797	0	420	50850	1527	1434	93	52719	0	501	50437	1781	1612	169
52985	0	52985	0	451	50904	1630	1507	123	52953	0	544	50739	1670	1447	223
52547	0	52547	0	376	50645	1526	1420	106	52453	0	513	50084	1856	1671	185
52469	0	52469	0	626	50178	1665	1585	80	52391	0	426	50475	1490	1331	159
52937	15	52906	0	344	51089	1473	1426	47	52922	0	388	51282	1252	955	297
52469	0	52469	0	516	50153	1800	1723	77	52453	0	528	50265	1660	1488	172

52781	0	52781	0	298	50676	1807	1714	93	52735	0	494	50341	1900	1631	269
52469	0	52438	0	500	50352	1586	1477	109	52469	0	294	50540	1635	1475	160
53406	15	53391	0	670	50814	1907	1783	124	53360	0	496	51366	1498	1344	154
52438	0	52438	0	406	50675	1357	1218	139	52391	0	436	50267	1688	1533	155
53046	15	53031	0	436	50895	1700	1512	188	52953	0	405	50670	1878	1706	172
52563	0	52563	0	483	50359	1721	1626	95	52547	0	390	50159	1998	1857	141
52281	0	52281	0	418	50187	1676	1597	79	52250	0	406	50123	1721	1484	237
52906	0	52906	0	656	50606	1644	1553	91	52860	0	423	51030	1407	1199	208
52745	**6,06**	**52736,06**	**0**	**451,74**	**50660,34**	**1623,98**	**1517**	**106,98**	**52692,62**	**0**	**465,22**	**50547,12**	**1679,32**	**1491,08**	**188,24**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
52745	6,06	52736,06	0	465,22	50660,34	1679,32	1517	188,24

Messwerte Schnitte-Algorithmus TC1 – POSIX-Threads – kleiner Scan

Small Scan

Time in [ms]

Total	Inital	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Tread 1 Lock2Unlock	Tread 1 Write2Unlock	Tread 1 Lock2Write	Tread 2 Total	Tread 2 Initial	Tread 2 Read	Tread 2 Calc	Tread 2 Lock2Unlock	Tread 2 Write2Unlock	Tread 2 Lock2Write
5671	0	5671	0	47	5469	155	139	16	5640	0	46	5329	265	234	31
5734	0	5625	0	15	5499	111	111	0	5734	0	46	5485	203	171	32
5593	0	5562	0	15	5328	219	219	0	5593	0	124	5235	234	234	0
5656	0	5656	0	46	5393	217	217	0	5640	0	93	5437	110	95	15
5687	0	5671	0	47	5439	185	185	0	5687	0	0	5548	139	139	0

5609	0	5609	0	109	5249	251	251	0	5593	0	16	5467	110	78	32
5578	0	5578	0	15	5300	263	247	16	5531	0	110	5311	110	95	15
5656	0	5656	0	16	5515	125	125	0	5625	0	64	5344	217	217	0
5656	0	5640	0	78	5407	155	139	16	5625	0	78	5345	202	171	31
5640	0	5640	0	94	5330	216	216	0	5609	0	62	5453	94	78	16
5671	0	5656	0	47	5392	217	217	0	5671	0	64	5405	202	186	16
5656	0	5656	0	16	5406	234	219	15	5625	0	16	5470	139	124	15
5625	0	5609	0	63	5421	125	125	0	5625	0	155	5328	142	142	0
5718	0	5718	0	30	5454	234	218	16	5703	0	47	5529	127	96	31
5734	0	5718	0	80	5480	158	126	32	5734	0	32	5450	252	220	32
5562	0	5562	0	63	5421	78	78	0	5531	0	75	5254	202	186	16
5625	0	5625	0	32	5455	138	138	0	5593	0	80	5339	174	174	0
5687	0	5687	0	63	5468	156	156	0	5656	0	48	5421	187	155	32
5609	0	5609	0	92	5409	108	108	0	5609	0	47	5282	280	250	30
5625	0	5593	0	79	5263	251	235	16	5625	0	77	5377	171	155	16
5687	0	5656	0	78	5309	269	269	0	5687	0	0	5545	142	126	16
5718	0	5671	0	62	5217	392	361	31	5718	0	92	5392	234	219	15
5609	0	5578	0	63	5344	171	171	0	5609	0	16	5484	109	109	0
5640	0	5640	0	45	5392	203	188	15	5593	0	63	5329	201	185	16
5656	0	5656	0	64	5469	123	123	0	5625	0	46	5361	218	202	16
5656	0	5656	0	31	5485	140	124	16	5593	0	77	5374	142	142	0
5656	0	5656	0	46	5533	77	77	0	5656	0	61	5347	248	201	47
5625	0	5625	0	31	5344	250	250	0	5593	0	32	5359	202	171	31
5625	0	5625	0	110	5252	263	232	31	5593	0	0	5343	250	234	16
5687	0	5656	0	80	5388	188	188	0	5687	0	47	5483	157	141	16
5625	0	5593	0	78	5296	219	203	16	5625	0	62	5325	238	206	32
5656	0	5609	0	32	5375	202	186	16	5656	0	16	5421	219	188	31
5609	0	5609	0	77	5299	233	218	15	5578	0	16	5330	232	232	0
5687	0	5656	0	31	5470	155	155	0	5687	0	47	5546	94	63	31

5640	0	5609	0	78	5296	235	220	15	5640	0	123	5422	95	79	16
5546	0	5531	0	111	5295	125	109	16	5500	0	16	5374	110	110	0
5609	0	5578	0	96	5358	124	108	16	5609	0	123	5283	203	203	0
5656	0	5656	0	79	5387	190	190	0	5625	0	62	5391	172	140	32
5656	0	5656	0	63	5386	207	207	0	5609	0	62	5421	126	110	16
5640	0	5593	0	45	5328	220	220	0	5640	0	77	5296	267	251	16
5625	0	5609	0	16	5404	189	189	0	5625	0	32	5422	171	140	31
5609	0	5593	0	32	5359	202	187	15	5562	0	79	5328	155	140	15
5609	0	5578	0	30	5422	126	126	0	5609	0	63	5421	125	109	16
5593	0	5593	0	31	5405	157	141	16	5562	0	94	5359	109	94	15
5656	0	5625	0	63	5343	219	219	0	5656	0	77	5374	205	205	0
5640	0	5640	0	30	5263	347	331	16	5578	0	0	5388	190	158	32
5593	0	5593	0	63	5418	112	112	0	5562	0	62	5330	170	154	16
5640	0	5640	0	48	5374	218	202	16	5625	0	32	5345	248	248	0
5671	0	5671	0	64	5436	171	171	0	5640	0	46	5406	188	157	31
5578	0	5578	0	47	5296	235	235	0	5578	0	47	5328	203	187	16
5641,78	**0,00**	**5627,40**	**0,00**	**55,42**	**5380,82**	**191,16**	**183,62**	**7,54**	**5623,38**	**0,00**	**57,00**	**5386,72**	**179,66**	**162,08**	**17,58**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
5641,78	0,00	5627,40	0,00	57,00	5386,72	191,16	235,00	16,00

Messwerte Schnitte-Algorithmus TC1 – POSIX-Threads – großer Scan

Big Scan

Time in [ms]

Total	Inital	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Tread 1 Lock2Unlock	Tread 1 Write2Unlock	Tread 1 Lock2Write	Tread 2 Total	Tread 2 Initial	Tread 2 Read	Tread 2 Calc	Tread 2 Lock2Unlock	Tread 2 Write2Unlock	Tread 2 Lock2Write
52046	0	52046	0	456	49895	1695	1584	111	51828	0	518	49423	1887	1729	158
51781	15	51766	0	437	49654	1675	1580	95	51531	0	360	49820	1351	1305	46
52656	0	52625	0	436	50595	1594	1453	141	52656	0	424	50609	1623	1530	93
52062	0	51968	0	360	50186	1422	1281	141	52062	0	465	50036	1561	1403	158
52265	0	52265	0	531	49937	1797	1720	77	52109	0	454	50144	1511	1290	221
51578	0	51578	0	404	49400	1774	1711	63	51421	0	749	48992	1680	1618	62
52093	15	52047	0	528	49783	1736	1597	139	52078	0	424	50172	1482	1339	143
52250	0	52250	0	519	50113	1618	1525	93	52078	0	496	50011	1571	1495	76
51812	0	51812	0	438	49933	1441	1410	31	51656	0	376	49654	1626	1500	126
52234	0	52187	0	438	50013	1736	1547	189	52234	0	489	50031	1714	1588	126
51781	0	51781	0	418	49852	1511	1464	47	51609	0	516	49399	1694	1569	125
51843	0	51843	0	452	49980	1411	1379	32	51656	0	391	49461	1804	1711	93
51843	0	51843	0	389	49941	1513	1435	78	51656	0	500	49892	1264	1107	157
52093	0	52093	0	472	50096	1525	1337	188	52046	0	621	49806	1619	1437	182
51968	0	51968	0	427	49786	1755	1676	79	51750	0	370	49471	1909	1800	109
51562	0	51562	0	421	49767	1374	1327	47	51468	0	548	49384	1520	1348	172
52171	0	52171	0	417	50424	1330	1269	61	52062	0	579	49761	1722	1534	188
51718	0	51718	0	469	49649	1600	1507	93	51640	0	486	49703	1451	1340	111
52016	0	52016	0	464	49782	1770	1646	124	51922	0	435	49940	1547	1328	219
51906	0	51906	0	607	49547	1752	1705	47	51796	0	387	50016	1393	1251	142
52171	0	52140	0	722	49951	1467	1309	158	52171	0	432	49913	1826	1577	249

52062	0	52062	0	281	49889	1892	1766	126	52000	0	608	49463	1929	1665	264
51750	0	51750	0	420	49501	1829	1752	77	51703	0	468	49742	1493	1335	158
51718	0	51718	0	562	49548	1608	1482	126	51640	0	436	49340	1864	1696	168
51671	0	51671	0	440	49589	1642	1548	94	51546	0	451	49596	1499	1312	187
51953	0	51953	0	406	50000	1547	1499	48	51890	0	407	49902	1581	1362	219
51921	0	51875	0	529	49450	1896	1816	80	51921	0	547	49948	1426	1252	174
51735	0	51735	0	469	49730	1536	1473	63	51641	0	280	49581	1780	1669	111
52031	0	52031	0	549	49856	1626	1548	78	51937	0	640	49302	1995	1742	253
52046	0	52031	0	433	50107	1491	1366	125	52046	0	482	49946	1618	1460	158
52109	0	52046	0	342	50152	1552	1397	155	52109	0	659	49747	1703	1564	139
51921	0	51921	0	500	49890	1531	1361	170	51671	0	472	49656	1543	1450	93
52031	0	52031	0	470	50129	1432	1402	30	51687	0	548	49511	1628	1565	63
52187	0	52187	0	514	50001	1672	1624	48	51859	0	803	49477	1579	1411	168
51953	0	51953	0	500	49647	1806	1682	124	51593	0	406	49868	1319	1179	140
51781	0	51468	0	412	49088	1968	1907	61	51781	0	641	49472	1668	1510	158
52421	0	52421	0	407	50133	1881	1772	109	52203	0	420	49907	1876	1769	107
53750	0	53750	0	592	51381	1777	1699	78	52671	0	547	50574	1550	1443	107
52156	15	52125	0	423	49750	1952	1827	125	51688	0	499	49389	1800	1692	108
53015	0	52625	0	405	50500	1720	1483	237	53015	0	437	50613	1965	1730	235
52656	46	52610	0	604	49939	2067	1895	172	52219	0	592	49914	1713	1513	200
53453	0	53015	0	470	50528	2017	1859	158	53453	0	560	50470	2423	1890	533
51765	0	51765	0	443	49630	1692	1568	124	51390	0	378	49579	1433	1292	141
52484	0	52218	0	404	50316	1498	1359	139	52484	0	636	50308	1540	1388	152
52000	0	51703	0	408	49732	1563	1437	126	52000	0	592	49789	1619	1542	77
51937	0	51671	0	406	49804	1461	1335	126	51937	0	485	50099	1353	1215	138
52031	0	52031	0	622	49327	2082	2005	77	51718	0	328	49959	1431	1291	140
51937	0	51625	0	464	49317	1844	1750	94	51937	0	501	49866	1554	1494	60
51937	15	51922	0	564	49635	1723	1532	191	51516	0	532	49129	1855	1699	156
52234	0	52234	0	344	49927	1963	1852	111	51859	0	373	49350	2136	1948	188

52089,88	**2,12**	**52034,64**	**0,00**	**463,76**	**49895,60**	**1675,28**	**1569,16**	**106,12**	**51930,86**	**0,00**	**494,96**	**49782,70**	**1652,56**	**1497,54**	**155,02**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
52089,88	2,12	52034,64	0,00	494,96	49895,60	1675,28	235,00	16,00

Messwerte Photo2Scan TC1 – sequenziell – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
4703	218	126	578	0	0	203	3781	173	186	1311	1345
4640	203	108	472	0	0	203	3842	110	187	1356	1453
9906	203	169	5956	0	0	203	3578	32	185	1471	1143
4640	218	77	435	0	0	203	3910	265	327	1553	981
4609	203	159	406	0	0	187	3841	205	236	1482	1172
4578	203	95	327	0	0	203	3953	188	156	1612	1266
10125	203	47	5780	0	0	187	4095	201	216	1516	1346
4984	531	94	549	0	0	203	3810	126	95	1561	1169
4671	281	110	379	0	0	187	3901	139	170	1591	1048
4609	203	156	280	0	0	203	3970	185	107	1659	1157
9250	203	157	4984	0	0	203	3891	205	140	1615	1229
4671	218	112	635	0	0	203	3691	158	156	1488	1185
4609	218	46	481	0	0	203	3864	189	204	1564	1093
4640	203	173	551	0	0	187	3713	140	204	1378	1197
10000	203	124	5851	0	0	203	3822	280	250	1312	1188
4625	218	48	438	0	0	203	3921	187	186	1554	1185
4656	203	173	455	0	0	203	3825	93	204	1561	1311
4656	203	108	407	0	0	203	3938	203	203	1516	1171
9765	203	64	5419	0	0	203	4079	188	110	1619	1290
4656	218	109	504	0	0	203	3825	186	188	1488	1233
4531	203	62	375	0	0	203	3891	173	253	1546	1078
9734	203	79	5531	0	0	203	3905	189	169	1447	1284
5250	718	109	434	0	0	203	3989	203	188	1468	1347
4640	203	79	593	0	16	187	3765	140	172	1830	981
10171	187	95	6123	0	0	187	3766	159	174	1548	907
4968	546	108	371	0	0	203	3943	188	315	1427	1160
4625	203	79	465	0	0	203	3878	80	216	1487	1226
4578	203	123	329	0	0	203	3923	140	248	1499	1234
9828	203	78	5551	0	0	203	3996	220	156	1511	1346
5218	656	77	513	0	0	203	3972	127	239	1637	1061
4609	203	111	452	0	0	203	3843	172	310	1686	973
10609	203	15	6419	0	0	203	3972	126	142	1293	1362
4609	218	172	343	0	0	203	3876	173	78	1784	1029
4625	203	125	471	0	0	187	3826	124	313	1434	1108
4562	203	127	412	0	0	203	3820	188	264	1496	1062
10234	203	108	6214	0	0	203	3709	80	201	1645	1028
4734	203	109	691	0	0	203	3731	159	218	1589	1046
4578	203	107	422	0	0	203	3846	93	232	1516	1264
9687	203	108	5370	0	0	187	4006	158	189	1489	1121
4578	218	109	345	0	0	203	3906	156	219	1483	1199

4656	203	63	402	0	0	203	3988	157	201	1564	1079
10156	203	94	5986	0	0	203	3873	201	296	1410	1130
5125	593	139	716	0	0	203	3677	110	361	1251	1063
4594	188	93	449	0	0	188	3864	187	263	1312	1397
10078	203	77	5747	0	0	203	4051	202	220	1660	1206
4609	203	108	330	0	0	187	3968	139	315	1529	1344
4593	203	79	417	0	0	203	3894	284	158	1419	1300
4609	203	45	295	15	0	188	4066	156	139	1627	1268
5062	203	46	703	0	0	187	4110	219	326	1610	1192
9421	218	139	5177	0	0	203	3887	187	204	1517	1121
6165,28	**247,22**	**102,36**	**1930,66**	**0,3**	**0,32**	**199,2**	**3883,8**	**167**	**209,78**	**1518,42**	**1181,56**

Messwerte Photo2Scan TC1 – sequenziell – großer Scan

Big Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
97750	1031	56988	4996	0	16	969	34735	1607	1851	13626	10705
94875	421	54923	4835	0	15	375	34696	1672	1901	14269	10275
96750	1094	55400	4503	0	62	985	35753	1978	1699	13896	10635
96812	453	56652	4267	0	16	390	35425	1668	1511	13863	10837
96500	1110	54949	4643	0	16	1047	35798	1599	2013	13488	10735
97968	500	56933	4687	0	78	390	35848	1621	1795	14618	10591
95860	1063	54943	4749	0	47	969	35105	1604	1789	13420	10901
94406	406	53989	4568	0	16	359	35443	1480	1488	14372	10903
95515	781	54551	4907	0	16	718	35276	1857	1918	13274	10607
95515	468	55811	4192	0	46	375	35044	1601	1922	13428	10609
95953	1031	55296	4536	0	32	953	35090	1750	1763	13796	10484
98406	453	58094	4504	0	0	406	35355	1671	1785	13648	10683
98719	1563	57451	4856	0	16	1500	34849	1616	1953	13546	10556
95687	421	54608	4594	0	16	359	36064	1768	1591	13555	11242
95813	813	55512	4189	0	31	735	35299	1910	1856	13719	10495
94968	453	54729	4839	0	16	390	34947	1612	1575	13739	10774
95094	813	54594	4520	0	46	735	35167	1848	1458	14212	10535
96281	500	56078	4844	0	32	421	34859	1435	1694	14094	10188
100235	1203	58825	4301	0	31	1141	35906	1854	1917	13280	11049
96359	406	55988	4719	0	32	343	35246	1632	1925	13753	10629
96750	1156	56030	4732	0	63	1047	34817	1387	1939	13912	10634
95843	453	55074	5185	0	47	359	35131	1774	1724	13488	10498
96250	1015	55960	4246	0	47	921	35013	1300	1954	13530	10826
95750	453	55677	4429	0	16	390	35191	1603	1767	13807	10346
98219	906	58229	4158	0	16	844	34926	1359	1761	13917	10352
95593	515	55963	4383	0	62	406	34732	1562	1638	13713	10732
95812	921	54924	4820	15	16	844	35147	1568	2051	13221	10611
96656	421	56218	4809	0	16	359	35208	1441	1720	14002	10929
96141	953	55416	4577	0	47	875	35195	1505	1928	14092	10721

95406	515	54567	4489	0	31	453	35835	1509	1915	13924	10465
99593	843	59350	4770	0	62	734	34630	1657	1610	13574	10903
95187	437	55617	3970	0	15	375	35163	1313	1648	13935	10617
97328	828	56314	4546	0	47	734	35640	1707	1734	13632	11035
97031	500	56764	4820	0	63	390	34947	1812	1873	13962	10135
96593	812	55746	4919	0	31	750	35116	1196	1726	14587	10967
96125	453	55944	4651	0	46	375	35077	1482	1619	13540	10984
98390	812	57010	4812	0	31	734	35741	1689	1842	14353	10338
95109	531	54831	4761	0	16	468	34986	1391	1917	13885	10364
96344	1031	55439	4309	0	47	953	35565	1507	1627	13962	10664
95468	546	55102	3766	0	31	484	36054	1634	2023	14434	10384
95625	968	55011	4003	0	31	906	35643	1623	1881	13491	11330
95562	437	55780	4694	0	15	375	34651	1357	1889	13419	10926
98453	1110	57632	4172	0	16	1047	35539	1572	2089	13850	11052
95125	468	54070	4814	0	31	390	35773	1654	1846	13700	10730
96219	1078	55767	4579	0	62	985	34795	1644	2139	13458	10137
95656	515	55219	4711	0	15	453	35195	1623	2043	14299	10676
96985	1188	55844	4434	0	46	1110	35519	1408	1779	13849	10486
96656	468	55962	5048	0	46	375	35178	1437	1971	13996	10966
95579	985	54571	4302	0	16	922	35721	1432	1773	14442	11095
97218	453	56998	4266	0	0	406	35501	1484	1920	13947	10762
96443	**735,08**	**55867**	**4568,48**	**0,3**	**32,16**	**660,48**	**35271**	**1588**	**1815**	**13830,3**	**10682**

Messwerte Photo2Scan TC1 – sequenzielle Pipeline – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
4000	390	48	3137	0	0	187	425	79	111	140	95
3796	437	94	2905	0	0	203	360	95	45	157	63
4671	515	109	3688	0	0	203	359	77	32	140	110
4250	203	47	3531	0	0	203	469	62	127	155	125
4515	578	111	3310	0	0	203	516	125	95	219	77
4235	594	172	3109	0	0	203	360	63	125	110	62
4031	515	125	3099	15	0	203	292	92	91	77	32
4328	562	126	3232	0	0	218	408	62	95	125	126
4328	515	156	3266	0	0	203	391	94	47	126	124
4500	203	157	3907	0	0	187	233	46	45	78	64
4562	203	126	3873	0	0	187	360	64	46	173	77
4015	203	95	3201	0	0	188	516	107	47	203	159
3937	203	93	3201	0	0	203	440	95	109	127	109
3828	203	78	3187	0	0	203	360	111	32	139	78
3843	203	92	3205	0	0	203	343	78	47	140	78
3843	203	94	3094	0	0	203	452	78	95	139	140
3860	188	142	3152	0	0	188	378	111	31	126	110
3843	203	157	3094	0	0	187	389	125	93	110	61

4234	203	95	3607	0	0	203	329	94	48	140	47
3984	203	46	3251	0	0	203	484	108	110	205	61
3828	203	62	3123	0	0	187	440	31	109	142	158
3843	203	123	3078	0	0	203	439	62	78	204	95
3937	203	124	3156	0	0	203	454	140	77	94	143
3640	203	203	2793	0	0	187	441	158	63	110	110
3937	203	92	3220	0	0	203	422	109	47	141	125
3906	203	138	3158	0	0	203	407	92	64	172	79
4109	203	124	3364	0	0	203	418	64	62	186	106
4828	765	201	3427	0	0	203	435	124	48	186	77
4046	203	47	3374	0	0	203	422	171	62	110	79
4031	203	218	3295	0	0	203	315	127	64	60	64
3968	203	63	3264	0	0	203	438	140	31	109	158
3750	203	64	3108	0	0	187	375	124	62	127	62
3828	203	93	3047	0	0	203	485	154	62	158	111
3828	203	94	3141	0	0	187	390	94	47	171	78
3796	203	107	3159	0	0	203	327	79	31	139	78
4703	203	77	3954	0	0	203	469	189	30	172	78
4125	203	62	3378	0	0	203	482	126	93	124	139
4000	203	93	3217	0	0	203	487	142	32	173	140
4265	468	126	3250	0	15	203	421	156	46	127	92
3640	203	111	2875	0	0	187	451	78	94	154	125
3812	203	47	3094	0	0	187	468	125	76	156	111
3828	203	79	3172	0	0	203	374	108	94	156	16
4312	203	77	3543	0	0	203	489	80	47	204	158
4125	203	78	3327	0	0	188	517	141	108	189	79
4703	203	80	3904	0	0	187	516	141	63	250	62
4015	203	141	3278	0	0	203	393	95	32	174	92
3968	203	62	3265	0	0	187	438	126	31	94	187
3921	203	157	3217	0	0	187	344	94	47	109	94
3781	203	63	3091	0	0	203	424	48	95	188	93
3843	187	94	3265	0	0	187	297	62	63	94	78
4058,38	**268,56**	**105,26**	**3271,72**	**0,3**	**0,3**	**197,92**	**412,84**	**102,9**	**66,58**	**146,04**	**97,3**

Messwerte Photo2Scan TC1 – sequenzielle Pipeline – großer Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
80093	406	53667	22134	0	0	359	3886	1191	609	1414	672
125265	2000	106957	12698	0	15	1938	3610	984	504	1373	749
82281	375	55632	22400	0	0	359	3858	1078	671	1501	608
81250	796	56273	20032	0	15	735	4149	1070	655	1644	780
101734	406	80697	16698	0	0	390	3933	1194	582	1516	641
101235	953	81328	15009	0	15	906	3945	1142	627	1375	801
81375	406	54326	22886	0	0	390	3757	1004	548	1487	718

81218	781	54935	21800	0	16	734	3702	1108	630	1186	778
81046	484	54732	21916	0	0	468	3914	1003	566	1482	863
82093	781	56024	21141	0	31	735	4147	1193	719	1623	612
159484	375	150044	4971	0	0	359	4094	1317	762	1455	560
80812	359	57284	19757	0	0	343	3412	797	581	1265	769
135921	781	122082	8958	0	16	734	4084	1317	560	1577	630
125984	421	107168	14168	0	15	391	4227	1285	513	1457	972
103125	421	80933	17941	0	0	406	3815	1253	532	1282	748
81125	781	54912	21224	0	15	735	4208	1273	815	1370	750
121265	375	103781	13014	0	0	359	4095	1373	625	1534	563
134265	375	120886	9233	0	0	359	3755	1127	593	1363	672
79953	375	54693	21173	0	0	359	3712	1021	733	1301	657
80688	781	54363	21646	0	15	735	3883	1372	389	1343	779
165015	375	157606	3754	0	0	359	3280	926	699	1000	655
104125	484	81058	18476	0	0	468	4107	1407	482	1513	705
101234	984	78371	18209	0	16	906	3670	1035	501	1350	784
98109	375	76020	17886	0	0	359	3812	1175	440	1430	767
167609	890	158870	4323	0	16	844	3511	1047	580	1117	767
80453	375	55414	20866	0	0	360	3798	823	892	1261	822
141281	796	127433	9548	0	15	735	3504	863	486	1405	750
136453	375	122472	9706	0	0	375	3900	1176	559	1366	799
161109	390	151883	4639	0	0	390	4197	1236	702	1354	905
83594	391	57726	21660	0	0	375	3817	1077	692	1209	839
161719	797	151979	5314	0	31	734	3629	1052	560	1202	815
159937	375	151886	3627	0	0	375	4049	1090	748	1432	779
80062	421	53546	22096	0	0	406	3983	1099	626	1465	793
167844	906	158905	4451	0	15	860	3550	1034	401	1426	689
79093	390	53538	21142	0	0	375	4023	1041	751	1465	766
128125	796	110375	12870	0	15	735	4084	1196	455	1480	953
80968	359	55125	21805	0	0	343	3679	1121	529	1564	465
127265	781	109440	13465	0	16	734	3579	1190	501	1158	730
146406	406	136153	6223	0	0	390	3624	1050	546	1281	747
84187	390	58480	21455	0	0	375	3846	1206	533	1399	708
97859	796	75866	17614	0	15	735	3583	1233	530	972	848
81000	406	54632	21644	0	0	390	4318	1322	721	1342	933
122968	781	107608	10791	16	15	735	3788	1345	719	1242	482
137390	406	122443	10595	15	0	375	3930	1175	467	1222	1066
100875	406	79843	16639	0	0	390	3987	1111	622	1520	734
165046	1015	156531	4258	0	15	969	3242	1096	420	999	727
78890	468	54190	20218	0	0	453	4014	1142	735	1494	643
81296	796	56145	20540	0	16	765	3815	907	609	1688	611
80062	375	55808	20288	0	0	375	3591	1068	577	1289	657
81531	891	56588	20419	0	31	828	3617	1017	762	1425	413
109634,3	**596,56**	**89733**	**15466,4**	**0,62**	**7,38**	**566,14**	**3834,26**	**1127**	**601,18**	**1372,36**	**733,48**

Messwerte Photo2Scan TC1 – parallele Pipeline – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
4109	406	140	3610	0	0	187	157	61	157	157	94
4187	468	126	3641	0	0	187	141	141	124	110	124
4390	578	62	3781	0	0	187	172	157	92	172	141
4156	578	186	3532	0	0	187	126	92	96	126	108
4125	500	78	3578	0	0	187	189	77	189	140	124
4171	578	171	3515	0	0	187	156	108	80	156	156
4000	484	93	3501	0	0	203	216	80	216	126	79
4078	421	95	3625	0	0	187	172	140	141	172	78
4265	500	170	3671	0	0	187	172	80	142	172	76
3859	187	77	3641	0	0	187	234	95	93	234	111
3812	187	140	3609	0	0	187	204	32	204	155	63
3968	203	110	3719	0	0	187	156	109	156	127	138
3984	187	156	3765	0	0	187	141	79	126	124	141
3656	187	93	3421	0	0	187	189	172	189	62	94
3843	187	124	3609	0	0	187	281	48	78	281	125
3843	187	173	3624	0	0	187	234	46	124	234	64
3703	187	156	3501	0	0	187	141	111	93	141	107
3703	187	77	3437	0	0	187	201	173	95	201	126
3640	203	123	3375	0	0	187	185	94	158	185	95
3656	188	203	3405	0	0	172	189	75	79	189	78
4046	187	92	3812	0	0	187	205	141	77	205	94
4890	187	154	4671	0	0	187	159	64	139	159	109
3656	187	63	3453	0	0	187	188	31	187	188	141
3812	187	140	3517	0	0	187	141	126	125	124	141
4875	187	95	4658	0	0	187	218	173	218	93	30
3640	187	141	3421	0	0	187	218	95	79	218	92
3796	187	77	3547	0	0	187	234	124	158	234	63
3656	187	124	3344	0	0	187	188	110	188	173	93
3750	187	111	3531	0	0	187	187	171	62	187	94
3703	187	48	3516	0	0	187	187	92	141	187	126
3828	187	126	3594	0	0	187	187	125	94	187	93
3953	187	92	3766	0	0	187	238	46	238	140	62
3875	187	109	3657	0	0	187	171	111	93	171	141
3671	187	77	3468	0	0	187	173	109	77	157	173
3562	187	158	3250	0	0	187	238	76	155	238	92
3703	203	155	3468	0	0	187	189	110	189	78	46
4078	187	141	3843	0	0	187	204	204	48	110	123
3859	203	94	3595	0	0	187	233	95	110	233	92
3671	187	139	3438	0	0	187	203	125	64	203	93
4078	531	159	3483	0	0	203	156	126	125	156	76
3968	203	79	3765	0	0	187	173	63	122	173	141

3968	187	189	3750	0	0	187	156	30	111	156	108
3875	187	61	3672	0	0	187	282	109	79	282	63
3984	187	141	3703	0	0	187	188	62	188	140	172
3812	203	80	3609	0	0	187	154	154	126	108	110
3796	187	76	3609	0	0	187	202	92	111	202	112
3625	187	107	3423	0	0	187	281	31	95	281	95
4656	187	141	4407	0	0	187	172	93	156	172	79
3734	187	156	3470	0	0	187	142	92	125	142	126
3890	203	94	3671	0	0	187	172	15	154	159	172
3931,16	**252,74**	**119,44**	**3633,42**	**0**	**0**	**187,34**	**170,4**	**99,3**	**129,32**	**170,4**	**105,48**

Messwerte Photo2Scan TC1 – parallele Pipeline – großer Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
50281	500	18839	49076	0	0	375	1558	754	1219	1558	863
84953	985	81125	36953	0	31	906	1488	673	1089	1488	795
51640	484	16766	50780	0	0	437	1277	782	1096	1277	876
53610	938	22544	52315	0	15	860	1663	656	1093	1663	699
51547	828	19518	50281	0	15	766	1904	847	934	1904	749
86890	453	84028	36875	0	0	406	1622	817	1051	1622	827
51296	859	17641	49998	0	15	782	1596	653	1017	1596	702
53312	406	21893	52483	0	0	375	1630	936	952	1630	809
51359	843	19297	50171	0	15	750	1685	610	1094	1685	767
51187	828	19181	49954	0	16	765	1633	881	973	1633	674
50359	468	19623	49421	0	0	421	1601	866	986	1601	626
51656	875	20282	50327	0	15	797	1764	823	1086	1764	544
58796	859	34375	57530	0	15	782	1717	795	986	1717	659
85234	437	81995	37970	0	0	390	1634	681	960	1634	545
79516	875	75568	41737	0	78	750	1701	620	997	1701	782
79640	484	76211	43073	0	0	437	1642	665	858	1642	844
50218	1234	16436	48515	16	15	1157	1574	943	1044	1574	675
53468	421	21747	52561	0	0	375	1614	941	953	1614	635
52453	921	19925	51096	0	15	829	1467	910	1092	1467	732
53110	922	19567	51733	0	31	844	1858	684	1033	1858	606
51687	437	19913	50702	0	0	390	1646	835	1186	1646	670
51906	844	20473	50639	0	16	750	1374	999	1121	1374	563
52906	1016	20604	51530	0	15	954	1460	799	1078	1460	639
51171	421	19076	50266	0	0	390	1486	840	1232	1486	740
50328	828	19046	49094	0	15	750	1176	1016	1035	1176	790
52078	828	20328	50827	0	15	766	1497	813	1002	1497	658
56781	437	19899	55861	15	0	375	1372	746	1090	1372	597
86328	890	82527	36941	0	15	797	1710	593	1016	1710	777
50468	437	16642	49623	0	0	406	1688	1076	937	1688	658
53063	828	21036	51874	0	15	766	1632	810	1205	1632	550

51297	938	20151	49924	0	31	859	1486	891	1031	1486	690
52781	437	19613	51936	15	0	360	1624	848	1057	1624	796
52062	844	19583	50766	0	32	765	1778	745	925	1778	891
85875	953	82110	36412	0	15	875	1450	592	1018	1450	848
49234	437	16866	48375	0	0	390	1693	1022	829	1693	605
89891	828	86278	33786	0	31	750	1725	593	984	1725	809
51953	484	17177	51092	0	0	437	1575	814	1156	1575	887
53156	937	22561	51784	0	15	860	1801	832	1163	1801	642
50375	828	19471	49078	0	16	750	1499	951	936	1499	629
50609	625	19564	49295	0	0	531	1678	843	1139	1678	840
51547	828	19141	50359	0	15	766	1562	721	1141	1562	671
89031	828	85408	40116	0	15	766	1514	801	971	1514	638
50250	546	17609	49249	0	0	484	1579	794	1016	1579	691
52391	828	21326	51127	0	15	766	1510	1125	985	1510	711
51625	953	19607	50264	0	31	875	1609	764	988	1609	783
51296	421	19790	50467	0	0	375	1606	952	902	1606	595
52625	860	19472	51314	0	31	781	1641	750	950	1641	764
62906	1671	30923	31700	0	31	1578	1593	845	956	1593	614
61578	437	57108	55545	0	0	390	1829	612	1047	1829	686
53250	891	22917	51876	0	15	829	1284	794	1173	1284	707
58419,46	**743,2**	**32055,6**	**48293,4**	**0,92**	**13,2**	**674,7**	**1594,1**	**807,1**	**1035,64**	**1594,1**	**710,96**

Messwerte Schnitte-Algorithmus TC2 – sequenziell

Small Scan

Time in [ms]

Total	Initial	Read	Calc	Write
7094	0	47	6826	221
7109	0	93	6735	281
7094	0	64	6734	296
7078	0	76	6749	253
7093	15	125	6764	189
7109	0	62	6831	216
7110	0	93	6765	252
7094	0	46	6752	280
7078	0	32	6846	200
7093	15	79	6809	190
7110	0	47	6842	221
7094	0	124	6797	173
7093	15	45	6780	253
7109	15	78	6844	172
7109	0	47	6829	233
7094	0	108	6716	270
7109	0	125	6748	236
7078	0	94	6766	218
7109	15	31	6716	347

Big Scan

Time in [ms]

Total	Initial	Read	Calc	Write
65781	15	890	62843	2033
65718	15	963	62885	1855
65766	0	787	62983	1996
65766	0	798	62947	2021
65781	0	860	62869	2052
65765	15	996	62209	2545
65781	15	806	62644	2316
65735	0	801	62783	2151
65718	15	831	62720	2137
65734	15	765	62923	2031
65781	0	604	63255	1922
65750	15	812	63046	1877
65750	15	1076	62791	1868
65719	0	702	63284	1733
65828	15	752	63103	1958
65796	15	859	62774	2148
65734	0	739	62882	2097
65750	0	815	63367	1568
65781	0	830	62875	2076

7078	0	140	6767	171
7078	0	109	6763	206
7093	0	78	6750	265
7078	0	112	6778	188
7078	0	80	6716	282
7078	0	93	6767	218
7094	0	77	6771	246
7094	0	125	6693	276
7078	0	64	6686	328
7109	0	142	6825	142
7094	0	95	6825	174
7093	15	123	6723	232
7093	0	47	6716	330
7094	0	79	6798	217
7094	0	95	6798	201
7094	0	139	6750	205
7093	15	96	6781	201
7094	0	78	6843	173
7094	0	108	6738	248
7110	0	77	6816	217
7078	0	141	6592	345
7109	15	111	6779	204
7093	0	124	6672	297
7078	0	95	6732	251
7078	0	140	6691	247
7093	0	78	6750	265
7093	0	63	6809	221
7094	0	141	6770	183
7078	0	125	6688	265
7094	0	61	6753	264
7094	0	139	6673	266
7093	**2,4**	**91,82**	**6761**	**236,58**

65750	0	841	62880	2029
65719	0	779	62691	2249
65781	0	799	63078	1904
65813	0	894	62847	2072
65812	15	763	63029	2005
65734	15	658	63114	1947
65781	0	749	63033	1999
65828	15	608	63162	2043
65750	15	860	62815	2060
65781	15	796	63281	1689
65781	15	720	62910	2136
65797	0	543	63183	2055
65812	15	750	63348	1699
65796	0	981	62543	2272
65765	15	940	62902	1908
65781	0	829	62752	2200
65765	15	923	62904	1923
65765	0	929	62740	2065
65734	0	841	63159	1734
65750	0	747	62874	2129
65750	0	703	62793	2254
65766	0	609	63264	1893
65796	0	751	62746	2299
65766	0	735	62936	2095
65781	15	795	62953	2002
65797	0	720	63051	2010
65797	0	776	62757	2264
65781	0	691	63353	1737
65968	15	696	62921	2336
65938	0	799	63054	2085
65985	0	625	63612	1748
65781	**6,3**	**790,72**	**62957**	**2024,5**

Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – kleiner Scan – Thread 1&2

Small Scan

Time in [ms]

Total	Initial	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Thread 1 Lock2Unlock	Thread 1 Write2Unlock	Thread 1 Lock2Write	Thread 2 Total	Thread 2 Initial	Thread 2 Read	Thread 2 Calc	Thread 2 Lock2Unlock	Thread 2 Write2Unlock	Thread 2 Lock2Write
11140	0	11140	0	0	11046	94	94	0	11140	0	47	11030	63	63	0
11140	0	11125	0	15	11017	93	93	0	11093	0	15	11046	32	32	0
10516	0	10422	0	0	10375	32	16	16	10469	0	62	10360	47	47	0
12078	0	12047	0	110	11859	78	78	0	12078	0	31	12000	47	47	0
12266	0	12235	0	61	12079	95	95	0	12250	0	46	12096	108	108	0
10563	0	10531	0	30	10471	30	15	15	10563	0	31	10516	16	16	0
11547	0	11547	0	48	11468	31	31	0	11531	0	0	11516	15	15	0
12031	15	12000	0	16	11936	48	48	0	12016	0	78	11892	46	46	0
11906	0	11906	0	32	11827	47	47	0	11828	0	46	11735	47	47	0
12625	15	12594	0	48	12470	76	76	0	12578	0	47	12500	31	31	0
12734	15	12672	0	78	12515	79	79	0	12719	0	15	12641	63	63	0
11765	15	11735	0	32	11656	47	32	15	11750	0	16	11640	94	79	15
11969	0	11969	0	62	11845	62	62	0	11969	0	31	11844	94	94	0
12875	15	12735	0	15	12658	62	62	0	12844	0	48	12733	63	63	0
10985	0	10922	0	31	10767	124	124	0	10938	0	47	10844	47	47	0
12063	0	12063	0	31	12000	32	32	0	12047	0	16	11999	32	32	0
11360	0	11360	0	0	11329	31	31	0	11360	0	31	11268	61	61	0
11890	15	11860	0	31	11720	109	109	0	11813	0	48	11749	16	16	0
11000	15	10953	0	31	10859	63	63	0	10969	0	30	10908	31	16	15
11328	0	11235	0	31	11174	30	30	0	11328	0	31	11281	16	16	0
12625	0	12625	0	48	12529	48	32	16	12609	0	94	12468	47	47	0

12438	0	12438	0	15	12312	111	95	16	12391	0	31	12299	61	45	16
14703	0	14672	0	31	14549	92	92	0	14703	0	16	14656	31	31	0
10750	0	10750	15	15	10656	64	48	16	10735	0	31	10688	16	16	0
12938	0	12938	0	78	12797	63	63	0	12922	0	32	12827	63	63	0
11812	15	11703	0	46	11563	94	78	16	11766	0	32	11686	48	48	0
11844	0	11828	0	16	11765	47	47	0	11797	0	0	11735	62	62	0
12406	0	12375	0	16	12264	95	63	32	12406	0	15	12282	109	93	16
12968	0	12968	0	48	12842	78	78	0	12968	0	32	12873	63	47	16
12468	0	12437	0	16	12389	32	16	16	12468	0	0	12453	15	15	0
12156	0	12110	0	31	11986	93	93	0	11969	0	31	11877	61	46	15
11828	0	11813	0	0	11734	79	79	0	11735	0	0	11673	62	62	0
12687	15	12656	0	31	12564	61	46	15	12625	0	31	12579	15	15	0
12422	0	12422	0	32	12312	78	62	16	12406	0	80	12294	32	0	32
10640	15	10610	0	32	10453	125	125	0	10578	0	31	10468	79	79	0
12656	0	12531	0	0	12468	63	63	0	12656	0	77	12531	48	48	0
12343	15	12328	0	46	12236	46	46	0	12250	0	15	12155	80	80	0
12375	0	12375	0	46	12155	174	174	0	12359	0	31	12220	108	108	0
11687	15	11672	0	32	11547	93	78	15	11656	0	15	11579	62	62	0
12015	15	12000	0	15	11906	79	79	0	11953	0	77	11845	31	15	16
13235	0	13235	0	16	13141	78	78	0	13235	0	32	13203	0	0	0
12766	0	12750	0	78	12625	47	31	16	12750	0	31	12704	15	15	0
13250	0	13141	0	32	13014	95	95	0	13250	0	47	13155	48	48	0
12516	0	12516	0	47	12392	77	77	0	12500	0	64	12405	31	31	0
12140	0	12093	0	16	11967	110	110	0	12125	0	48	12030	47	47	0
12187	0	12187	0	32	12076	79	79	0	12140	0	31	12062	47	31	16
11671	0	11546	0	63	11403	80	80	0	11484	0	47	11359	78	78	0
12094	0	12094	0	45	12003	46	46	0	12094	0	31	11984	79	48	31
12484	15	12422	0	0	12343	79	79	0	12469	0	32	12342	95	95	0
11563	0	11563	0	32	11406	125	125	0	11547	0	16	11531	0	0	0

12068,96	4,2	12036,98	0,3	33,14	11929,36	73,88	69,48	4,4	12036,58	0	35,32	11951,22	50,04	46,28	3,76

Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – kleiner Scan – Thread 3&4

Small Scan

Time in [ms]

Thread 3 Total	Thread 3 Initial	Thread 3 Read	Thread 3 Calc	Thread 3 Lock2Unlock	Thread 3 Write2Unlock	Thread 3 Lock2Write	Thread 4 Total	Thread 4 Initial	Thread 4 Read	Thread 4 Calc	Thread 4 Lock2Unlock	Thread 4 Write2Unlock	Thread 4 Lock2Write
11125	0	15	11064	46	46	0	11125	0	142	10920	63	63	0
11140	0	63	11000	77	77	0	11109	0	32	11046	31	31	0
10516	0	47	10393	76	76	0	10406	0	62	10298	46	46	0
12063	0	16	11909	138	138	0	11969	0	16	11922	31	31	0
12266	0	62	12076	128	128	0	12250	0	0	12234	16	16	0
10563	0	0	10471	92	76	16	10516	0	30	10438	48	48	0
11547	0	32	11453	62	62	0	11547	0	30	11485	32	32	0
12000	0	32	11827	141	141	0	12016	0	78	11907	31	31	0
11859	15	32	11748	64	64	0	11781	0	79	11640	62	47	15
12610	0	64	12484	62	62	0	12578	0	47	12437	94	94	0
12719	0	0	12626	93	93	0	12656	0	0	12640	16	16	0
11750	0	47	11545	158	158	0	11703	0	0	11640	63	63	0
11938	0	79	11750	109	109	0	11906	0	46	11813	47	47	0
12844	0	48	12686	110	94	16	12860	0	32	12797	31	31	0
10985	0	78	10812	95	95	0	10938	0	16	10892	30	30	0
12063	0	16	11953	94	94	0	12031	0	32	11938	61	61	0
11360	0	32	11219	109	109	0	11313	0	15	11298	0	0	0
11875	0	16	11811	48	48	0	11828	0	15	11734	79	79	0

10985	0	32	10877	76	76	0	10969	0	78	10814	77	77	0
11235	0	47	11112	76	76	0	11235	0	0	11189	46	46	0
12593	0	61	12420	112	112	0	12625	0	48	12530	47	47	0
12422	0	79	12234	109	109	0	12422	0	108	12252	62	62	0
14703	0	32	14671	0	0	0	14672	0	16	14563	93	93	0
10735	0	32	10596	107	107	0	10703	0	63	10578	62	62	0
12906	0	31	12749	126	126	0	12860	0	47	12766	47	47	0
11719	0	62	11563	94	94	0	11797	0	16	11687	94	63	31
11844	0	0	11797	47	47	0	11828	0	30	11752	46	46	0
12406	0	46	12266	94	94	0	12406	0	47	12281	78	78	0
12968	0	0	12809	159	159	0	12937	0	16	12891	30	30	0
12468	0	48	12358	62	62	0	12437	0	31	12345	61	61	0
12156	0	77	11968	111	111	0	12016	0	47	11953	16	16	0
11828	0	48	11704	76	76	0	11797	0	46	11720	31	31	0
12641	0	46	12549	46	46	0	12672	0	15	12642	15	0	15
12406	0	16	12344	46	31	15	12250	0	48	12125	77	61	16
10625	0	0	10547	78	78	0	10594	0	0	10548	46	46	0
12610	0	16	12532	62	62	0	12610	0	62	12486	62	62	0
12328	0	15	12220	93	77	16	12203	0	32	12156	15	15	0
12375	0	15	12282	78	62	16	12343	0	16	12280	47	47	0
11641	0	47	11485	109	109	0	11656	0	0	11593	63	47	16
11969	0	16	11890	63	63	0	12000	0	0	11985	15	15	0
13235	0	61	13067	107	107	0	13156	0	32	13109	15	15	0
12578	0	31	12499	48	48	0	12766	0	45	12689	32	16	16
13203	0	0	13140	63	63	0	13250	0	47	13171	32	32	0
12516	0	16	12421	79	79	0	12453	0	31	12360	62	62	0
12140	0	31	12031	78	78	0	12109	0	0	12094	15	15	0
12187	0	15	12125	47	47	0	12156	0	16	12124	16	16	0
11656	0	31	11561	64	64	0	11578	0	46	11532	0	0	0

12094	0	31	11968	95	95	0	12016	0	63	11890	63	63	0
12453	0	32	12325	96	96	0	12438	0	32	12328	78	78	0
11563	0	15	11469	79	79	0	11516	0	16	11455	45	30	15
12048,22	**0,3**	**34,16**	**11928,12**	**85,64**	**84,06**	**1,58**	**12020,04**	**0**	**35,32**	**11939,34**	**45,38**	**42,9**	**2,48**

Time								
		Thread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
12068,96	4,2	12048,22	0,3	35,32	11951,22	85,64	84,06	4,4

Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – großer Scan – Thread 1&2

Big Scan

Time in [ms]

Total	Initial	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Thread 1 Lock2Unlock	Thread 1 Write2Unlock	Thread 1 Lock2Write	Thread 2 Total	Thread 2 Initial	Thread 2 Read	Thread 2 Calc	Thread 2 Lock2Unlock	Thread 2 Write2Unlock	Thread 2 Lock2Write
129766	0	129672	0	421	128480	771	771	0	129766	0	248	129081	437	375	62
119422	0	119422	0	330	118358	734	718	16	119063	0	359	118327	361	345	16
116656	0	116625	0	265	115626	734	734	0	116422	0	279	115520	623	608	15
135593	15	135438	0	237	134642	559	529	30	135485	0	356	134723	406	406	0
127641	0	127610	0	326	126427	857	841	16	127547	0	247	126881	419	403	16
120125	0	120125	0	219	119358	548	548	0	120110	0	269	119263	578	547	31
123766	0	123766	0	310	122742	714	714	0	123391	0	248	122700	443	427	16
112468	15	112281	0	329	111358	594	594	0	112141	0	234	111361	546	514	32
127812	15	127672	0	298	126642	732	716	16	127797	0	297	126909	591	560	31
118140	0	118140	0	264	117161	715	699	16	117921	0	234	117156	531	453	78

127516	0	127516	0	204	126605	707	691	16	126953	0	232	126239	467	451	16
113687	0	113578	0	329	112601	648	648	0	113500	0	284	112471	745	651	94
132750	15	132688	0	298	131472	918	902	16	132735	0	278	131853	604	527	77
125093	0	125093	0	356	123960	777	715	62	124984	0	252	124312	420	373	47
148531	0	148531	0	312	147411	808	808	0	148250	0	312	147432	506	458	48
129203	0	128969	0	358	127797	814	799	15	128625	0	142	128110	373	358	15
124297	0	124141	0	283	123196	662	615	47	124297	0	203	123674	420	388	32
137265	15	137250	0	217	136350	683	667	16	137016	0	171	136411	434	403	31
109875	0	109594	0	361	108218	1015	984	31	109797	0	219	108953	625	562	63
128234	0	128078	0	285	127095	698	668	30	127593	0	282	126843	468	452	16
122047	0	121781	0	358	120719	704	688	16	122047	0	267	121079	701	701	0
131860	0	131860	0	297	130860	703	687	16	131547	0	314	130703	530	498	32
128515	15	128391	0	362	127199	830	814	16	127875	0	201	127157	517	486	31
136750	0	136703	0	280	135676	747	731	16	136422	0	266	135684	472	441	31
144843	0	144828	0	326	143719	783	737	46	144765	0	315	144011	439	423	16
134531	0	134531	0	375	133390	766	720	46	134500	0	206	133822	472	440	32
140437	15	140406	0	158	139453	795	779	16	140375	0	360	139652	363	347	16
126937	0	126812	0	217	125866	729	698	31	126812	0	221	126173	418	418	0
122860	0	122719	0	372	121599	748	733	15	122844	0	309	122052	483	452	31
117500	15	117485	0	376	116611	498	498	0	117219	0	217	116674	328	328	0
114859	0	114765	0	266	113729	770	754	16	114750	0	297	113833	620	588	32
114735	0	114641	0	330	113637	658	642	16	114438	0	175	113842	421	406	15
116360	0	116344	0	264	115357	723	708	15	116360	0	296	115454	610	547	63
134015	15	134000	0	297	133016	687	672	15	133735	0	343	133030	362	362	0
139047	0	139031	0	346	137971	714	668	46	139000	0	204	138299	497	466	31
155093	0	154859	0	344	153796	719	672	47	154687	0	328	153719	640	592	48
127140	15	126906	0	266	125599	1041	1041	0	127125	0	235	126229	661	598	63
133656	0	133656	0	402	132563	691	660	31	133234	0	95	132514	625	625	0
119031	15	118969	0	269	118029	671	640	31	118797	0	220	118136	441	410	31

111500	0	111469	0	231	110377	861	845	16	111047	0	235	110352	444	429	15
136343	15	136328	0	310	135379	639	624	15	136266	0	282	135612	372	372	0
123718	0	123718	0	267	122882	569	521	48	123640	15	279	122699	647	584	63
141156	0	141000	0	281	140059	660	660	0	140578	0	203	139979	396	364	32
136938	0	136578	0	295	135595	688	688	0	136906	0	297	136153	456	409	47
144750	0	144750	0	356	143644	750	750	0	144609	0	372	143892	345	313	32
144453	0	144375	0	376	143200	799	767	32	143922	0	406	143017	499	467	32
116078	0	116016	0	231	115188	597	581	16	115594	0	264	114690	640	624	16
145375	0	145250	0	247	144267	736	720	16	145375	0	236	144610	529	514	15
122891	0	122891	0	281	121795	815	815	0	122641	0	281	121905	455	423	32
111625	0	111625	0	297	110776	552	536	16	111125	0	281	110330	514	482	32
128057,66	**3,6**	**127977,52**	**0**	**301,58**	**126949**	**726,62**	**708,2**	**18,42**	**127832,56**	**0,3**	**263,02**	**127070,42**	**497,88**	**467,4**	**30,48**

Messwerte Schnitte-Algorithmus TC2 – MFC-Threads – großer Scan – Thread 3&4

Big Scan

Time in [ms]

Thread 3 Total	Thread 3 Initial	Thread 3 Read	Thread 3 Calc	Thread 3 Lock2Unlock	Thread 3 Write2Unlock	Thread 3 Lock2Write	Thread 4 Total	Thread 4 Initial	Thread 4 Read	Thread 4 Calc	Thread 4 Lock2Unlock	Thread 4 Write2Unlock	Thread 4 Lock2Write
129750	0	204	128922	624	593	31	129578	0	204	129002	356	293	63
119297	0	298	118359	640	564	76	119078	0	344	118411	323	277	46
116610	0	235	115531	844	797	47	116656	0	251	115896	509	509	0
135578	0	267	134702	609	578	31	134969	0	365	134136	468	438	30
127641	0	249	126627	765	749	16	126906	0	262	126117	527	496	31
120047	0	217	119068	762	762	0	120110	0	222	119511	377	377	0
123735	0	280	122737	718	686	32	123750	0	232	123097	421	390	31

112453	0	218	111564	671	639	32	112141	0	392	111437	312	296	16
127766	0	188	126833	745	730	15	127172	0	264	126410	498	436	62
118062	0	407	116887	768	705	63	117125	0	298	116322	505	457	48
127469	0	184	126487	798	766	32	127375	0	266	126675	434	418	16
113672	0	341	112672	659	659	0	112953	0	236	112312	405	374	31
132375	0	233	131198	944	897	47	131735	0	142	131159	434	371	63
124796	0	345	123714	737	721	16	124750	0	266	123921	563	500	63
148500	0	279	147490	731	669	62	148500	0	300	147696	504	425	79
129203	0	283	128107	813	797	16	129016	0	345	128170	501	422	79
124188	0	301	123159	728	712	16	123391	0	205	122715	471	455	16
137188	0	283	135993	912	897	15	136516	0	294	135737	485	453	32
109750	0	296	108595	859	859	0	109875	0	123	109378	374	342	32
128218	0	293	127159	766	718	48	127921	0	345	127030	546	516	30
121969	0	186	120952	831	815	16	121297	0	329	120624	344	312	32
131688	0	190	130753	745	730	15	131703	0	249	130998	456	426	30
128375	0	249	127485	641	610	31	128500	0	376	127702	422	376	46
136750	0	206	135920	624	592	32	136625	0	408	135797	420	372	48
144796	0	264	143719	813	798	15	144390	0	313	143751	326	310	16
134485	0	281	133521	683	667	16	133922	0	376	133042	504	441	63
140125	0	264	139024	837	805	32	140422	0	294	139737	391	360	31
126937	0	404	125658	875	860	15	126671	0	372	125938	361	330	31
122860	0	142	121889	829	829	0	122422	0	295	121784	343	312	31
117406	0	312	116255	839	823	16	116906	0	247	116287	372	326	46
114515	0	294	113314	907	860	47	114859	0	217	114345	297	297	0
114735	0	248	113854	617	586	31	114563	0	311	113865	371	355	16
116360	0	310	115175	875	875	0	116000	0	340	115238	422	391	31
133797	0	314	132661	807	807	0	133235	0	332	132512	391	343	48
138969	0	434	137755	780	780	0	139047	0	266	138302	479	463	16
155093	0	205	154246	642	611	31	154953	0	361	154001	591	560	31

127047	0	230	126086	731	700	31	126500	0	246	125785	469	391	78
132781	0	359	131781	641	626	15	133343	0	253	132670	420	420	0
118828	0	390	117624	814	814	0	119016	0	295	118205	516	516	0
111500	0	284	110342	874	874	0	111125	0	263	110362	500	469	31
136016	0	235	135075	706	674	32	135594	0	219	134907	468	452	16
123500	0	248	122632	620	620	0	123703	0	359	123005	339	324	15
141156	0	344	140082	730	715	15	140750	0	342	140067	341	326	15
136938	0	314	135840	784	784	0	136828	0	268	136125	435	403	32
144250	0	248	143111	891	891	0	144484	0	344	143587	553	521	32
144375	0	452	143187	736	704	32	144453	0	154	143876	423	376	47
116063	0	283	115136	628	598	30	115500	0	359	114623	518	503	15
145360	0	299	144204	857	857	0	144781	0	347	143952	482	482	0
122625	0	311	121536	778	763	15	122250	0	293	121487	470	454	16
111625	0	408	110688	529	529	0	111500	0	339	110834	327	327	0
127944,44	**0**	**282,18**	**126906,18**	**755,14**	**733,9**	**21,24**	**127697,18**	**0**	**290,46**	**126970,8**	**435,28**	**403,66**	**31,62**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
128057,66	3,6	127977,52	0,3	301,58	127070,42	755,14	733,9	31,62

Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – kleiner Scan – Thread 1&2

Small Scan

Time in [ms]

Total	Initial	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Tread 1 Lock2Unlock	Tread 1 Write2Unlock	Tread 1 Lock2Write	Tread 2 Total	Tread 2 Initial	Tread 2 Read	Tread 2 Calc	Tread 2 Lock2Unlock	Tread 2 Write2Unlock	Tread 2 Lock2Write
14343	0	14296	0	30	14187	79	79	0	14312	0	15	14281	16	16	0
12687	0	12656	0	32	12546	78	78	0	12687	0	46	12609	32	32	0
11281	0	11250	0	0	11202	48	48	0	11218	0	0	11187	31	31	0
11609	0	11593	0	0	11563	30	30	0	11609	0	15	11562	32	32	0
12750	0	12703	0	16	12607	80	80	0	12750	0	16	12687	47	47	0
10578	0	10578	0	48	10437	78	78	0	10515	0	16	10360	139	109	30
13953	0	13953	0	31	13842	80	80	0	13921	0	16	13842	63	63	0
13343	0	13312	0	15	13235	62	62	0	13328	0	32	13251	45	45	0
12796	0	12781	0	45	12658	78	78	0	12796	0	45	12658	93	93	0
11828	0	11828	0	30	11687	111	111	0	11812	0	16	11717	79	79	0
12328	0	12203	0	32	12108	63	63	0	12312	0	45	12157	110	110	0
11734	0	11671	0	0	11639	32	32	0	11718	0	16	11671	31	31	0
12968	0	12953	0	31	12813	109	109	0	12937	0	94	12782	61	61	0
11437	0	11437	0	16	11405	16	16	0	11390	0	31	11282	77	77	0
12109	0	12109	0	32	12030	47	47	0	12062	0	16	11998	48	48	0
12468	15	12453	0	30	12392	31	31	0	12297	0	0	12220	77	77	0
11984	0	11984	0	0	11889	95	95	0	11953	0	47	11858	48	48	0
12640	0	12609	0	32	12531	46	46	0	12640	0	61	12501	78	78	0
12312	0	12296	0	62	12219	15	15	0	12250	0	15	12174	46	46	0
15421	15	15421	15	16	15295	95	95	0	15359	15	0	15188	156	156	0
12625	0	12531	0	61	12299	171	155	16	12625	0	62	12500	63	63	0
11187	0	11187	0	0	11077	110	110	0	11187	0	48	11093	46	46	0
12687	0	12687	0	31	12593	63	63	0	12671	0	63	12562	46	46	0
10546	0	10500	0	15	10454	31	31	0	10546	0	0	10468	78	78	0

11343	0	11312	0	0	11234	78	78	0	11343	0	15	11266	62	62	0
10859	0	10859	0	16	10734	109	109	0	10859	0	16	10782	61	61	0
12234	0	12000	0	62	11876	62	62	0	12203	0	15	12125	63	63	0
11437	0	11421	0	0	11390	31	31	0	11437	0	63	11313	61	46	15
13265	0	13265	0	0	13219	46	46	0	13187	0	15	13110	62	62	0
12703	0	12687	0	31	12578	78	78	0	12703	0	62	12611	30	30	0
12343	0	12343	0	31	12264	48	48	0	12296	0	64	12202	30	30	0
12000	0	12000	0	16	11889	95	95	0	11906	0	46	11812	48	48	0
11781	0	11718	0	63	11577	78	78	0	11781	0	47	11718	16	16	0
10515	0	10515	0	47	10405	63	63	0	10484	0	64	10357	63	63	0
12546	0	12500	0	0	12469	31	31	0	12500	0	47	12422	31	31	0
11562	0	11562	0	0	11516	46	46	0	11562	0	31	11531	0	0	0
12078	0	12078	0	63	11936	79	79	0	11968	0	15	11861	92	76	16
12640	0	12640	0	95	12468	77	47	30	12578	0	16	12530	32	32	0
11062	0	11062	0	48	10937	77	77	0	11046	0	15	10999	32	32	0
12328	0	12296	0	0	12216	80	80	0	12296	0	16	12170	110	110	0
12656	0	12593	0	0	12548	45	45	0	12609	0	47	12499	63	63	0
12171	0	12156	0	16	12045	95	95	0	12171	0	47	12108	16	16	0
10812	0	10812	0	93	10608	111	111	0	10812	15	15	10688	94	94	0
11609	0	11515	0	47	11389	79	79	0	11515	0	78	11406	31	31	0
12171	0	12171	0	45	12062	64	64	0	12156	0	61	12016	79	79	0
14234	0	14218	0	46	14157	15	15	0	14234	0	47	14140	47	47	0
12906	0	12890	0	63	12780	47	47	0	12875	0	15	12813	47	47	0
12453	0	12406	0	48	12264	94	94	0	12390	0	95	12231	64	64	0
11015	0	10953	0	15	10893	45	45	0	11015	0	47	10923	45	45	0
11093	0	11078	0	16	10951	111	111	0	11078	0	0	11030	48	48	0
12188,6	**0,6**	**12160,82**	**0,3**	**29,32**	**12062,26**	**68,64**	**67,72**	**0,92**	**12157,98**	**0,6**	**34,28**	**12065,42**	**57,38**	**56,16**	**1,22**

Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – kleiner Scan – Thread 3&4

Small Scan

Time in [ms]

Thread 3 Total	Thread 3 Initial	Thread 3 Read	Thread 3 Calc	Thread 3 Lock2Unlock	Thread 3 Write2Unlock	Thread 3 Lock2Write	Thread 4 Total	Thread 4 Initial	Thread 4 Read	Thread 4 Calc	Thread 4 Lock2Unlock	Thread 4 Write2Unlock	Thread 4 Lock2Write
14343	0	64	14217	62	62	0	14281	0	0	14250	31	31	0
12687	0	16	12640	31	31	0	12578	0	31	12532	15	15	0
11281	0	16	11110	155	155	0	11281	0	48	11156	77	77	0
11593	0	0	11498	95	95	0	11562	0	15	11470	77	77	0
12703	0	0	12609	94	94	0	12687	0	32	12608	47	47	0
10515	0	15	10393	107	107	0	10531	0	47	10454	30	30	0
13953	0	32	13891	30	30	0	13953	0	31	13906	16	16	0
13328	0	31	13185	112	112	0	13343	0	46	13236	61	61	0
12796	15	30	12642	109	109	0	12687	15	48	12608	16	16	0
11796	0	31	11655	110	110	0	11812	0	45	11751	16	16	0
12187	0	0	12076	111	111	0	12328	0	47	12266	15	15	0
11718	0	47	11545	126	126	0	11734	0	46	11595	93	93	0
12968	0	31	12841	96	96	0	12906	0	0	12843	63	63	0
11406	0	15	11298	93	93	0	11437	0	48	11343	46	46	0
12109	0	0	11999	110	110	0	12046	0	47	11889	110	110	0
12453	0	46	12328	79	79	0	12360	0	15	12297	48	48	0
11984	0	31	11860	93	78	15	11921	0	48	11841	32	32	0
12625	0	15	12531	79	79	0	12609	0	0	12454	155	155	0
12312	0	46	12172	94	94	0	12218	0	15	12125	78	78	0
15406	0	16	15296	94	94	0	15391	0	47	15298	46	46	0
12531	0	62	12391	78	63	15	12562	0	61	12486	15	15	0

11171	0	48	11014	109	109	0	11156	0	31	11049	76	76	0
12656	0	32	12577	47	47	0	12687	0	47	12593	47	47	0
10546	0	16	10438	92	92	0	10500	0	30	10423	47	47	0
11343	0	15	11252	76	76	0	11328	0	15	11235	78	78	0
10843	0	15	10689	139	139	0	10828	0	0	10796	32	32	0
12046	0	15	11890	141	141	0	12234	0	47	12140	47	47	0
11437	0	45	11282	110	94	16	11437	0	48	11313	76	76	0
13265	0	15	13141	109	109	0	13265	0	45	13205	15	15	0
12671	0	31	12547	93	93	0	12671	0	48	12498	125	125	0
12328	0	31	12173	124	124	0	12234	0	31	12110	93	93	0
11984	0	0	11907	77	77	0	11937	0	48	11843	46	46	0
11625	0	16	11514	95	95	0	11656	0	30	11610	16	16	0
10515	0	31	10360	124	124	0	10484	0	47	10344	93	93	0
12546	0	63	12375	108	108	0	12546	0	30	12455	61	61	0
11562	0	32	11436	94	94	0	11546	0	31	11515	0	0	0
12078	0	79	11873	126	126	0	12015	0	46	11922	47	47	0
12593	0	16	12470	107	107	0	12515	0	63	12389	63	63	0
11062	0	32	10953	77	77	0	11062	0	47	10999	16	16	0
12328	0	31	12234	63	63	0	12312	0	77	12188	47	47	0
12656	0	30	12548	78	78	0	12640	0	62	12530	48	48	0
12156	0	47	12031	78	78	0	12156	0	78	12078	0	0	0
10766	0	32	10641	93	78	15	10796	15	0	10735	46	46	0
11609	0	77	11487	45	45	0	11531	0	47	11438	46	46	0
12171	0	31	12078	62	62	0	12140	0	46	12048	46	46	0
14234	0	63	14029	142	111	31	14171	0	47	14092	32	32	0
12906	0	92	12706	108	108	0	12859	0	61	12766	32	32	0
12437	0	15	12310	112	112	0	12421	0	16	12405	0	0	0
10921	0	48	10842	31	31	0	10968	0	32	10890	46	46	0
11093	0	0	11014	79	79	0	11062	0	16	10984	62	62	0

12164,84	**0,3**	**30,84**	**12039,76**	**93,94**	**92,1**	**1,84**	**12147,68**	**0,6**	**37,66**	**12060,02**	**49,4**	**49,4**	**0**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
12188,6	0,6	12164,84	0,6	37,66	12065,42	93,94	92,1	1,84

Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – großer Scan – Thread 1&2

Big Scan

Time in [ms]

Total	Initial	Thread 1 Total	Thread 1 Initial	Thread 1 Read	Thread 1 Calc	Thread 1 Lock2Unlock	Thread 1 Write2Unlock	Thread 1 Lock2Write	Thread 2 Total	Thread 2 Initial	Thread 2 Read	Thread 2 Calc	Thread 2 Lock2Unlock	Thread 2 Write2Unlock	Thread 2 Lock2Write
121468	0	121468	0	187	120468	813	813	0	120937	0	265	120109	563	563	0
127734	0	127703	0	280	126693	730	730	0	127734	0	237	126945	552	552	0
123015	0	123015	0	330	121919	766	766	0	122671	0	252	121978	441	441	0
137500	0	137484	0	389	136579	516	454	62	136968	0	261	136284	423	391	32
161234	0	161015	0	218	160042	755	739	16	159671	0	281	158762	628	612	16
112265	0	112109	0	283	111128	698	698	0	112265	0	360	111405	500	469	31
146484	0	146484	0	201	145690	593	593	0	146046	0	282	145199	549	549	0
114062	0	114062	0	250	113283	529	529	0	113921	0	269	113277	375	359	16
145890	0	145687	0	426	144594	667	621	46	145453	0	374	144765	314	314	0
99734	0	99593	0	280	98564	749	734	15	99734	0	281	99063	390	375	15
115406	0	115406	0	266	114564	576	576	0	115171	0	345	114403	423	423	0
113531	0	113421	0	250	112465	706	690	16	113421	0	235	112672	514	514	0
127093	0	126968	0	188	126177	603	603	0	126812	0	246	126068	498	498	0

127765	0	127609	0	206	126642	761	761	0	127765	0	173	127200	392	360	32
121578	0	121578	0	153	120831	594	594	0	121437	0	296	120606	535	488	47
125656	0	125484	0	268	124527	689	657	32	125218	0	235	124465	518	488	30
115078	0	114937	0	168	114115	654	638	16	114703	0	282	113886	535	503	32
116078	15	116000	15	248	115069	668	668	0	116078	15	378	115453	232	232	0
112078	0	112078	0	343	111084	651	620	31	111781	0	198	111241	342	327	15
120500	0	120453	0	283	119496	674	674	0	120500	0	267	119782	451	451	0
112515	15	112438	0	325	111338	775	775	0	112172	0	312	111361	499	484	15
109625	0	109531	0	487	108385	659	659	0	109625	0	280	108846	499	483	16
114203	0	114203	0	297	113204	702	670	32	113906	0	406	113014	486	486	0
114515	0	114515	0	236	113593	686	686	0	114468	0	359	113687	406	406	0
114328	0	114172	0	233	113128	811	779	32	114000	0	201	113268	531	500	31
119187	0	119078	0	156	118084	838	822	16	119187	0	221	118718	248	232	16
123578	0	123578	0	203	122620	755	755	0	123421	0	251	122701	469	453	16
126625	0	126578	0	298	125475	805	773	32	126281	0	235	125457	589	573	16
115781	0	115640	0	172	114735	733	733	0	115546	0	374	114610	562	546	16
124531	0	124265	0	172	123312	781	781	0	124531	0	373	123597	561	561	0
131437	0	131437	0	373	130426	638	623	15	131203	0	280	130424	499	467	32
136046	0	136031	0	249	135046	736	704	32	135953	0	142	135325	486	486	0
110031	0	109953	0	298	108994	661	661	0	109406	0	204	108653	549	533	16
113187	0	112859	0	203	111985	671	656	15	113187	0	263	112551	373	343	30
129562	0	129562	0	348	128429	785	785	0	129015	0	250	128230	535	503	32
130843	0	130843	0	264	129983	596	596	0	130500	0	268	129766	466	466	0
124765	0	124609	0	264	123565	780	765	15	124484	0	234	123891	343	328	15
120656	0	120656	0	261	119645	750	703	47	120640	0	175	120137	328	313	15
125937	0	125937	0	262	124907	768	736	32	125703	0	234	125125	344	344	0
117921	0	117921	0	250	116826	845	830	15	117281	0	251	116591	439	407	32
119468	0	119328	0	236	118405	687	672	15	119078	0	354	118175	549	549	0
124140	0	123859	0	316	122781	746	716	30	124140	0	328	123327	470	439	31

112468	0	112406	0	189	111447	770	770	0	112281	0	341	111521	419	404	15
114703	0	114609	0	284	113601	724	724	0	114609	0	451	113781	377	377	0
122593	0	122500	0	404	121267	829	814	15	122593	0	374	121782	422	390	32
107468	0	107468	0	313	106515	640	640	0	107343	0	264	106702	377	377	0
121171	0	120890	0	388	119718	784	753	31	121171	0	358	120284	529	513	16
125953	0	125921	0	346	124934	641	626	15	125546	0	235	124844	467	451	16
124859	0	124859	0	347	123562	950	918	32	124796	0	232	123836	728	680	48
123547	0	123547	0	252	122619	676	644	32	123422	0	221	122609	592	577	15
121916	**0,6**	**121834,94**	**0,3**	**272,86**	**120849,18**	**712,28**	**698,54**	**13,74**	**121675,48**	**0,3**	**279,76**	**120927,52**	**466,34**	**451,6**	**14,74**

Messwerte Schnitte-Algorithmus TC2 – POSIX-Threads – großer Scan – Thread 3&4

Big Scan

Time in [ms]

Thread 3 Total	Thread 3 Initial	Thread 3 Read	Thread 3 Calc	Thread 3 Lock2Unlock	Thread 3 Write2Unlock	Thread 3 Lock2Write	Thread 4 Total	Thread 4 Initial	Thread 4 Read	Thread 4 Calc	Thread 4 Lock2Unlock	Thread 4 Write2Unlock	Thread 4 Lock2Write
121468	0	204	120472	792	776	16	121265	0	263	120615	387	356	31
127703	0	380	126505	818	818	0	127406	0	312	126706	388	356	32
122906	0	313	121827	766	750	16	122593	0	221	121978	394	394	0
137343	0	221	136293	829	813	16	137500	0	405	136651	444	444	0
161234	0	282	160278	674	658	16	160984	0	299	160386	299	299	0
112218	0	280	111297	641	625	16	111796	0	348	111215	233	218	15
146187	0	251	145290	646	599	47	145531	0	283	144811	437	406	31
113390	15	329	112469	577	577	0	114015	15	251	113356	393	377	16
145890	0	199	144726	950	950	0	145515	0	250	144826	439	439	0
99640	0	187	98692	761	746	15	99687	0	266	98926	495	495	0

115156	0	281	114280	595	595	0	114781	0	156	114050	559	559	0
113500	0	329	112247	924	924	0	113531	0	284	112821	426	395	31
127093	0	248	125923	906	875	31	126546	0	404	125687	455	424	31
127703	0	296	126663	728	728	0	127078	0	171	126485	422	361	61
121421	0	299	120481	641	641	0	120906	0	405	120221	280	280	0
125593	0	284	124387	922	922	0	125656	0	419	124721	485	470	15
115078	0	221	114313	544	544	0	114703	0	278	114054	371	356	15
116031	0	327	114958	746	730	16	115688	0	246	114971	471	423	48
111906	0	342	110868	696	680	16	111671	15	171	110983	502	486	16
120438	0	298	119394	746	746	0	120469	0	345	119752	372	357	15
112500	0	265	111473	762	762	0	112266	0	155	111770	341	341	0
109578	0	173	108844	561	561	0	109171	0	298	108622	251	251	0
114031	0	278	113106	647	615	32	113781	0	217	113175	389	373	16
114062	0	293	113209	560	560	0	114375	0	248	113688	424	424	0
114328	0	345	113075	908	908	0	114219	0	280	113567	372	372	0
119046	0	233	117972	841	826	15	119046	0	296	118394	356	341	15
123390	0	235	122388	767	736	31	123171	0	295	122407	469	469	0
126500	0	377	125330	793	793	0	126625	0	345	125902	378	378	0
115781	0	389	114811	581	581	0	115609	0	264	114735	610	610	0
124484	0	331	123561	592	592	0	124031	0	343	123373	315	283	32
131359	0	248	130504	591	591	0	131156	0	329	130529	298	298	0
135875	0	236	135141	498	498	0	136046	0	310	135358	378	378	0
110031	0	250	109174	607	607	0	109812	0	156	109249	391	360	31
113109	0	310	112191	608	608	0	112546	0	217	111764	565	549	16
129468	0	141	128399	928	928	0	128796	0	250	128073	473	458	15
130468	0	234	129547	687	687	0	130828	0	219	130268	341	310	31
124765	0	456	123576	733	701	32	124546	0	229	123925	392	376	16
120625	0	312	119439	874	874	0	120468	0	373	119641	454	454	0
125843	0	279	124688	876	876	0	125562	0	374	124764	424	408	16

117890	0	283	116987	620	620	0	117890	0	332	117071	487	456	31
119468	0	234	118386	848	848	0	118937	0	345	118199	393	393	0
124062	0	266	123122	674	674	0	123562	0	235	123002	325	325	0
112296	0	361	111297	638	638	0	112468	0	361	111730	377	377	0
114703	0	157	113871	675	675	0	114468	0	312	113827	329	329	0
122359	0	286	121154	919	903	16	122328	0	344	121549	435	435	0
107312	0	219	106212	881	849	32	107234	0	237	106470	527	527	0
121093	0	296	120064	733	733	0	121156	0	344	120425	387	387	0
125953	0	310	124945	698	698	0	125250	0	219	124563	468	452	16
124843	0	312	123718	813	781	32	124250	0	394	123459	382	366	16
123406	0	266	122436	704	704	0	123063	0	235	122468	360	360	0
121810,52	**0,3**	**278,92**	**120799,66**	**730,38**	**722,48**	**7,9**	**121599,62**	**0,6**	**286,66**	**120903,64**	**406,86**	**394,7**	**12,16**

Time								
		Tread Max						
Total	Init	Total	Init	Read	Calc	Lock2Unlock	Write2Unlock	Lock2Write
121916	0,6	121834,94	0,6	286,66	120927,52	730,38	722,48	14,74

Messwerte Photo2Scan TC2 – sequenziell – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
4062	203	78	388	0	0	187	3393	111	170	1443	984
3984	203	142	452	0	0	187	3187	110	252	999	1079
4343	234	94	685	0	0	218	3330	124	235	1309	1063
5796	500	125	1939	0	0	187	3232	139	252	1205	1123
5562	203	141	1966	0	0	187	3252	249	92	1268	1053
5593	187	31	2034	0	0	171	3341	109	155	1501	966
6187	203	95	2561	0	0	187	3328	110	219	1281	908
6703	578	111	2892	0	0	187	3122	125	190	1246	871
4968	937	125	528	0	0	187	3378	218	190	1541	701
3968	203	123	327	0	0	187	3315	156	174	1423	983
5531	187	78	2038	0	0	171	3228	64	174	1423	826
5546	937	32	1328	0	0	187	3249	108	156	1406	1015
5890	203	80	2268	0	0	187	3339	202	170	1296	938
4593	203	110	827	0	0	187	3453	108	218	1479	964
6687	203	127	3123	0	0	187	3234	92	158	1372	1032
3968	187	77	642	0	0	187	3062	158	125	1318	918
3968	187	169	485	0	0	187	3127	251	220	1373	784
4906	203	78	1283	0	0	187	3342	140	235	1329	1043
5203	188	79	1562	0	0	188	3374	139	76	1561	1018
5734	187	61	2159	0	0	187	3327	110	158	1354	938
4953	187	77	1457	0	0	187	3232	109	63	1359	969
6812	484	94	3205	0	0	203	3029	128	173	1045	1027
4375	562	47	547	0	0	203	3219	172	217	1219	908
3937	203	47	234	0	0	187	3453	219	200	1350	934
3953	187	94	265	0	0	187	3407	219	234	1374	957
6406	203	142	2809	0	0	187	3252	141	189	1544	756
4718	203	158	1106	0	0	187	3251	155	189	1093	1096
6531	203	16	2951	0	0	187	3361	138	93	1471	1142
4062	203	126	465	0	0	187	3268	64	188	1278	1096
3953	187	141	232	0	0	187	3393	95	171	1356	1241
3921	203	142	343	0	0	187	3233	109	202	1434	830
6468	500	128	2526	0	0	187	3314	94	232	1448	958
5000	1000	60	471	0	0	187	3469	141	79	1404	1110
6421	203	80	2654	0	0	187	3484	171	205	1251	1015
4671	468	78	950	0	0	187	3175	126	251	1147	1041
5765	187	0	2234	0	0	171	3344	141	94	1594	797
5703	453	111	1625	0	0	203	3514	155	142	1471	1092
5171	203	77	1483	0	0	187	3408	47	62	1659	938
5546	203	79	1811	0	0	187	3453	32	80	1468	1188
5515	843	154	1403	0	0	187	3115	174	140	1127	892
6421	187	126	2970	0	0	187	3138	79	174	1127	1089

4312	203	158	669	0	0	187	3282	62	140	1255	1170
3953	203	93	407	0	0	187	3250	156	204	1296	969
7093	703	110	3080	0	0	187	3200	138	157	1129	1139
5046	578	47	951	0	0	187	3470	232	173	1177	1201
5296	187	123	1781	0	0	187	3205	78	141	1345	1005
5531	1031	94	1141	0	0	187	3265	110	190	1393	888
6125	203	93	2542	0	0	187	3287	219	173	1423	941
4109	187	157	578	0	0	187	3187	110	155	1295	981
4046	203	78	262	0	0	187	3503	174	204	1296	955
5180,1	**334,12**	**97,72**	**1452,78**	**0**	**0**	**187,64**	**3295,48**	**136,22**	**170,68**	**1339,1**	**990,64**

Messwerte Photo2Scan TC2 – sequenziell – großer Scan

Big Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
43921	421	7036	6998	15	16	328	29466	1519	1572	11861	8943
44328	312	7218	7331	0	0	250	29467	1419	1371	11772	9182
44468	406	7332	7016	0	0	343	29714	1285	1588	11637	8983
45062	406	7642	8575	0	15	328	28439	1123	1406	11673	8411
44093	281	7075	7388	0	0	218	29349	1295	1293	12181	8789
45765	531	6332	8504	0	16	468	30398	1275	1435	12245	8958
45750	546	7526	7717	0	32	468	29961	1236	1418	11795	9659
44375	343	6557	7625	0	31	250	29850	1376	1502	12445	8517
45453	406	6447	8165	0	0	343	30435	1386	1503	12906	9165
44609	406	6722	7446	0	15	328	30035	1200	1296	12392	9081
44171	296	6739	7375	0	0	218	29761	1250	1640	12693	7960
44484	437	6872	7293	0	16	343	29882	1093	1786	12068	8814
45984	812	6175	8416	0	0	750	30581	1395	1457	12662	8803
45875	281	8038	7403	0	16	218	30153	1309	1355	12469	8729
45093	406	7435	8382	0	15	328	28870	1357	1358	11857	8150
45531	515	8312	6942	0	15	453	29762	1148	1579	12698	8313
45046	265	6493	8417	0	0	203	29871	1049	1268	12748	8876
45890	406	7373	8118	0	15	328	29993	1168	1528	12542	8690
45390	640	6924	7918	0	32	546	29908	1427	1343	12281	9253
44562	281	6532	8467	0	16	218	29282	1506	1663	11707	8466
45250	1390	7172	7388	0	16	1312	29300	1140	1420	12583	8615
45203	406	6563	7300	0	0	343	30934	1673	1705	12465	9099
44546	281	7487	7274	0	0	218	29504	1152	1284	11965	9201
46062	656	6966	8988	0	16	593	29436	1408	1341	11769	8676
45031	437	7468	7334	0	31	328	29792	1122	1414	12609	8701
45203	312	6685	8356	0	16	234	29850	1247	1552	11956	8865
44953	390	6550	7198	0	15	328	30815	1123	1565	12590	9115
45171	406	6866	8370	0	15	328	29529	1453	1572	12021	8539
44593	281	7102	7184	0	15	203	30026	1379	1637	11938	9189
45140	421	5967	7944	0	31	328	30808	1313	1409	12553	8519

44640	437	6479	7261	0	16	359	30448	1107	1782	13312	8744
44078	328	6984	7283	0	31	250	29483	1350	1788	11875	8685
44984	437	6947	8309	0	15	375	29291	943	1410	12812	8532
45453	406	8250	7177	0	0	343	29620	1437	1293	12262	8845
44796	296	5150	8327	0	0	234	31023	1345	1653	13165	8820
46546	468	8898	7735	0	15	328	29445	1408	1612	11914	8739
44875	406	6307	8043	0	16	343	30104	1173	1456	12565	8898
45406	343	7135	7781	0	16	265	30147	1248	1261	12848	8291
45390	515	7730	7994	0	47	421	29151	1248	1253	11533	8704
45531	562	7724	8355	0	16	343	28890	1281	1640	11857	8375
44265	296	6267	7832	0	0	234	29870	1379	1532	12117	8493
44984	484	6488	8260	0	0	343	29752	1340	1536	12249	9049
45093	468	7292	7248	0	0	328	30085	1096	1493	11589	9681
45015	296	7323	8366	0	16	234	29030	1218	1831	11705	8361
44656	390	7347	7308	0	0	328	29611	998	1411	12648	8690
46140	515	6856	8955	0	32	421	29814	1481	1635	11876	8692
46016	297	8307	8392	0	16	219	29020	1124	1359	11748	9048
45250	390	6273	8265	0	15	328	30322	1425	1578	12114	8564
46062	406	8390	7664	0	16	343	29602	1191	1382	11937	8810
45765	312	7611	7903	0	16	234	29924	1312	1404	12558	8559
45118,94	**428,58**	**7067,28**	**7825,8**	**0,3**	**13,76**	**348,34**	**29796,06**	**1278,6**	**1491,38**	**12235,3**	**8776,82**

Messwerte Photo2Scan TC2 – sequenziell Pipeline – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
1843	203	48	1266	0	0	187	326	93	15	79	139
1812	203	154	1065	0	0	187	390	78	31	155	126
2984	187	189	2214	0	0	187	394	63	63	111	157
1843	187	173	1201	0	0	187	282	78	16	140	48
1828	187	31	1065	0	0	187	545	140	125	203	77
1875	203	123	1205	0	15	172	344	96	46	76	126
1781	203	204	952	0	0	187	422	110	77	142	93
1875	187	110	1154	0	0	187	424	140	32	124	128
1750	187	110	1108	0	0	187	345	96	62	109	78
1718	187	91	923	0	0	187	517	110	78	204	125
1703	187	125	1046	0	0	171	345	111	62	125	47
1734	187	109	1013	0	0	187	425	109	80	110	126
1812	203	77	1170	0	0	187	362	95	79	78	110
1828	187	110	1236	0	0	171	295	141	31	77	46
1781	203	127	1169	0	0	187	282	32	79	109	62
1843	187	124	1159	0	0	187	373	62	77	109	125
1843	203	111	1249	0	0	187	280	93	31	109	47
1859	187	156	1221	0	0	187	295	15	79	92	109
1828	187	126	1079	0	0	171	436	109	0	109	218
1843	203	92	1170	0	0	187	378	157	48	62	111

2000	187	111	1357	0	0	187	345	111	46	94	94
1843	187	48	1096	0	0	187	512	154	31	171	156
1890	187	157	1218	0	0	187	328	110	62	77	79
1860	188	155	1192	0	0	172	309	78	30	122	79
1890	187	78	1218	0	0	187	407	47	16	171	173
1734	187	95	1109	0	0	187	343	45	30	220	48
1734	187	109	1047	0	0	187	391	126	63	93	109
1781	203	111	1030	0	0	187	437	63	94	139	141
1734	187	77	1051	0	0	187	419	123	111	61	124
1671	187	0	1188	0	0	171	296	47	46	110	93
1703	187	109	1047	0	0	187	360	63	32	219	46
1703	203	92	999	0	0	187	409	93	30	207	79
1796	187	76	1065	0	0	187	468	95	47	156	170
1750	187	94	1110	0	0	187	359	62	62	126	109
1671	203	62	967	0	0	187	439	47	79	174	139
1687	203	78	1044	0	0	187	362	47	48	156	111
1734	187	93	1062	0	0	187	392	125	48	32	187
1734	187	110	1108	0	0	187	329	155	64	16	94
1750	203	170	1065	0	0	203	312	79	31	47	155
2000	187	78	1439	0	0	187	296	140	46	31	79
1859	187	94	1170	0	0	187	408	143	46	94	125
1843	187	78	1048	0	0	187	530	125	124	125	156
1781	203	172	1013	0	0	187	393	93	95	142	63
1719	188	125	1124	0	0	172	282	16	63	94	109
1750	187	79	1123	0	0	187	361	94	47	142	78
1843	187	31	1218	0	0	187	407	47	77	95	188
1718	203	94	1155	0	0	187	266	16	31	94	125
1703	203	110	980	0	0	187	410	111	62	157	80
1671	203	96	1076	0	0	187	296	78	31	124	63
1687	187	109	1033	0	0	187	358	63	63	108	124
1812,44	**192,48**	**105,62**	**1140,34**	**0**	**0,3**	**185,14**	**373,68**	**90,48**	**55,32**	**118,4**	**109,48**

Messwerte Photo2Scan TC2 – sequenzielle Pipeline – großer Scan

Big Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
30750	250	11903	15503	0	0	218	3094	865	457	1049	723
27859	390	10164	14214	0	0	328	3091	858	591	798	844
27359	375	9126	14937	0	0	328	2921	825	340	1079	677
28937	375	11648	13605	0	0	328	3309	871	621	1141	676
31578	375	13402	14490	0	0	328	3311	924	593	948	846
27171	328	10682	13142	0	0	296	3019	911	453	936	719
28015	359	12563	12259	0	15	313	2818	708	452	977	681
26984	390	9927	13411	0	0	328	3256	800	548	956	952
27640	390	10410	13775	0	0	328	3065	892	656	876	641
27484	375	9429	14570	0	0	328	3110	888	607	988	627

29062	265	10742	14708	0	0	218	3347	758	540	1121	928
24313	360	8936	11805	0	16	312	3212	892	560	1106	654
28968	390	10505	14807	0	0	344	3266	854	494	1038	880
30453	375	10916	15915	0	0	328	3247	833	503	1050	861
29375	390	11512	14404	0	0	328	3069	914	393	1095	667
30640	343	11699	15410	0	0	312	3188	764	542	1153	729
27953	375	10645	13752	0	16	328	3181	935	569	969	708
27375	390	9638	14524	0	0	328	2823	740	579	879	625
28859	375	11268	14286	0	15	328	2930	955	580	892	503
24562	390	8959	11393	0	16	328	3820	890	685	1199	1046
29046	265	10976	14614	0	0	234	3191	813	534	1014	830
26750	375	10923	12730	0	0	328	2722	699	501	908	614
26718	375	9581	13429	0	0	328	3333	887	544	995	907
28156	375	10408	14227	0	0	328	3146	721	392	1097	936
25500	375	9532	12571	0	0	313	3007	823	498	953	733
27859	359	12032	12424	0	0	328	3044	792	360	1187	705
28890	375	11762	13538	0	15	328	3215	814	655	897	849
26406	375	8472	14267	0	0	328	3292	908	659	992	733
32015	390	19393	9085	0	0	328	3147	878	551	1077	641
27812	375	10966	13206	0	0	328	3233	1042	576	880	735
29953	375	12162	14270	0	0	343	3130	908	498	812	912
26125	375	9948	12537	0	16	328	3265	872	531	1037	825
34109	375	21557	8944	0	0	328	3218	687	486	1231	814
31156	421	12967	14778	0	0	360	2990	804	531	937	718
35687	265	23573	9084	0	0	234	2765	704	517	809	735
29718	656	12195	13702	0	0	610	3165	870	719	1003	573
30610	375	11679	15141	0	0	344	3415	907	512	1185	811
26796	421	11107	12267	0	0	375	2970	752	563	982	673
26875	296	9944	13385	0	0	265	3235	828	574	971	862
30062	375	12658	13507	0	0	328	3506	833	595	1205	873
28953	375	12045	13489	0	15	328	3044	876	611	972	585
29578	2313	10905	12961	0	15	2235	3399	830	611	1077	881
30328	359	12627	14022	0	15	313	3305	763	390	1264	888
29437	265	11663	13983	0	0	234	3511	1188	503	1098	722
28218	390	9553	15289	0	0	328	2986	916	280	963	827
25984	375	8666	13574	0	0	328	3369	973	486	1187	723
26750	390	10157	13119	0	0	328	3068	877	405	989	797
23844	375	8925	11114	0	0	312	3430	846	670	1104	810
27562	250	10283	13635	0	0	203	3394	1004	671	873	846
26219	360	9229	13570	0	16	312	3060	860	611	932	657
28449,06	**405,7**	**11399,24**	**13467,44**	**0**	**3,4**	**357,68**	**3172,64**	**855,04**	**535,94**	**1017,62**	**764,04**

Messwerte Photo2Scan TC2 – parallele Pipeline – kleiner Scan

Small Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC

1968	218	298	1750	0	0	187	313	313	281	283	281
1984	234	297	1750	0	0	203	343	188	343	330	155
1921	218	314	1703	0	0	187	343	294	297	343	204
1953	218	282	1703	0	0	187	344	234	329	344	62
2281	218	236	2048	0	0	187	313	264	313	297	219
2015	234	298	1781	0	0	187	327	202	327	295	173
3281	218	266	3048	0	0	187	312	250	312	299	186
1937	187	249	1702	0	0	187	406	204	406	328	171
1859	250	249	1593	0	0	203	407	234	407	327	237
1750	172	296	1546	0	0	172	391	220	374	391	204
1750	187	282	1563	0	0	171	314	233	314	279	189
1703	187	235	1501	0	0	187	328	297	328	235	249
1718	187	327	1485	0	0	171	375	187	375	329	188
1718	187	203	1500	0	0	171	406	203	406	297	251
1906	218	361	1672	0	0	218	313	218	313	311	220
1891	172	360	1687	0	0	172	359	236	327	359	156
1843	187	329	1656	0	0	187	282	201	282	267	141
1703	187	297	1500	0	0	171	359	235	359	282	233
1703	172	312	1515	0	0	172	358	221	311	358	126
1671	171	235	1500	0	0	171	390	266	390	377	202
1687	187	330	1437	0	0	171	327	217	296	327	220
1687	187	283	1470	0	0	187	390	139	390	341	205
2328	187	265	2141	0	15	172	342	298	342	327	173
1921	187	252	1702	0	0	171	374	264	374	357	222
2000	172	360	1767	0	0	172	313	204	313	264	203
1890	187	267	1687	0	0	187	344	234	344	344	109
1890	187	217	1688	0	0	187	406	268	406	296	250
1968	187	251	1781	0	0	171	359	156	359	327	219
1937	171	299	1766	0	0	171	358	186	358	266	220
1937	187	250	1750	0	0	171	375	250	344	375	157
1875	187	312	1672	0	0	171	422	142	422	374	188
1796	187	281	1593	0	0	171	343	251	343	295	189
1718	171	204	1516	0	0	171	390	234	390	374	172
1687	171	282	1500	0	0	171	390	234	359	390	204
1875	187	376	1657	0	0	171	327	250	297	327	188
1953	187	281	1750	0	0	187	360	171	360	250	265
1968	187	282	1781	0	0	171	330	233	330	297	187
1953	187	266	1750	0	0	187	374	265	297	374	218
1875	187	265	1673	0	0	187	343	267	343	312	172
2078	187	345	1876	0	0	171	297	234	297	297	171
1735	172	189	1532	0	0	172	422	232	422	280	219
1750	187	217	1531	0	0	187	391	314	391	344	187
1953	187	219	1766	0	0	187	391	250	391	327	203
2359	187	203	2172	0	0	187	376	251	376	359	202
2000	187	217	1813	0	0	171	390	205	390	265	189
2000	187	281	1813	0	0	187	358	267	358	296	235

1937	187	282	1734	0	0	171	344	218	328	344	250
1765	187	265	1548	0	0	187	374	220	358	374	157
1953	187	250	1750	0	0	187	296	266	296	296	173
1953	203	139	1735	0	0	203	407	268	407	298	249
1919,66	**191,4**	**273,12**	**1711,08**	**0**	**0,3**	**180,7**	**349,5**	**233,76**	**349,5**	**320,56**	**196,86**

Messwerte Photo2Scan TC2 – parallele Pipeline – großer Scan

Big Scan

Time in [ms]

Total	Initial	Read	Write	IO	OO	Photo	Pipe	LE	CP2GS	CP2PC	PC2IC
19625	234	2004	19251	0	0	187	2939	1839	2939	2640	1266
17031	234	2003	16644	0	0	187	3218	1497	3218	2937	1205
19046	234	2595	18657	0	0	187	2778	2219	2778	2704	1077
17250	234	2104	16922	0	0	187	2938	1722	2938	2519	1464
19890	218	1797	19466	0	0	171	2953	1939	2953	2693	1258
19000	234	1713	18657	0	0	187	3204	1599	3204	2687	1063
16968	218	2675	16610	0	0	187	2764	2464	2764	2608	1049
17828	234	2113	17453	0	0	187	2969	1668	2969	2469	1109
18218	234	1860	17844	0	0	187	3234	1484	3234	2881	1072
17578	218	1922	17156	0	0	171	2953	2156	2953	2580	1407
18218	234	2074	17736	0	0	187	2798	2159	2798	2597	1027
19390	218	1577	19000	0	0	171	2733	1549	2733	2518	1138
18953	234	2300	18594	0	0	187	2528	2045	2528	2075	1410
16156	234	2077	15749	0	0	187	3201	1707	3201	3029	1220
17593	218	2810	17076	0	0	171	2994	2191	2749	2994	1049
17671	234	2057	17310	0	0	187	3112	2221	3112	2875	1373
18687	234	2330	18327	0	0	187	2965	2126	2965	2564	1094
15718	234	1655	15406	0	0	187	2889	1720	2889	2301	1262
17421	218	1813	16968	0	0	187	2800	1452	2800	2497	1061
19078	234	2327	18671	0	0	187	2953	2313	2953	2886	1175
18531	234	2565	18201	0	0	187	2867	2544	2781	2867	1025
16984	234	2408	16579	0	0	187	2873	2279	2873	2655	969
16235	219	1722	15828	0	0	172	3406	1917	3406	2972	1218
18515	218	1997	18204	0	0	171	2734	1738	2734	2018	1341
19703	234	1537	19312	0	0	187	3079	1730	3079	2669	1001
17546	234	2061	17202	15	0	172	2893	1892	2893	2733	1405
18125	219	1921	17813	0	0	172	2875	1954	2875	2748	986
16859	234	2000	16548	0	0	187	3455	1484	3455	2973	1214
17296	218	2276	16954	0	0	171	3064	2188	3064	2771	1295
18531	234	2625	18109	0	0	187	2641	2641	2477	2251	1145
18281	218	1962	17845	0	0	171	2969	1427	2969	2764	1111
16640	218	2052	16266	0	0	171	2987	1981	2983	2987	1151
17328	218	1780	17048	0	0	171	2969	1580	2969	2143	1263
19187	218	1984	18861	0	0	171	3173	1611	3173	2841	1267
17796	218	2282	17467	0	0	187	3016	1937	3016	2629	1166
17062	218	2138	16734	0	0	171	2801	2187	2801	2787	1039

18312	218	2465	18001	0	0	187	3001	2083	3001	2708	1351
15718	218	1859	15455	0	0	171	3092	1532	3092	2751	1138
17296	234	2765	16860	0	0	187	2823	2528	2693	2823	1144
17640	218	2234	17140	0	0	171	2986	1894	2986	2733	1124
18593	218	2222	18186	0	0	171	2656	2327	2656	2235	1283
18125	219	1565	17734	0	0	172	3172	1560	3172	2562	952
18171	234	2062	17829	0	0	187	2870	1845	2861	2870	1267
17796	250	2222	17452	0	0	203	3076	1995	3076	2768	1109
17578	219	1983	17203	0	0	172	2970	1919	2970	2722	1170
16531	234	1942	16249	0	0	187	2828	2168	2828	2355	1115
17406	234	2405	17109	0	0	187	2861	2033	2861	2450	1064
17343	218	1642	17029	0	0	187	2939	1825	2939	2698	1193
18265	218	1791	17876	0	0	171	2986	1676	2986	2780	1204
18719	219	1924	18328	0	0	172	3202	1593	3202	2939	1421
17868,62	**226,42**	**2083,94**	**17498,38**	**0,3**	**0**	**180,72**	**2950,98**	**1922,76**	**2950,98**	**2665,12**	**1178,2**