Karlsruhe Reports in Informatics 2010,16
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Fallstudie: Parallelisierung der Erstellung von Tiefenkarten aus Stereobildern

Oliver Denninger

2010

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fallstudie: Parallelisierung der Erstellung von Tiefenkarten aus Stereobildern

Oliver Denninger

FZI Forschungszentrum Informatik
76131 Karlsruhe

Institut für Programmstrukturen und Datenorganisation (IPD)
Karlsruher Institut für Technologie (KIT)
76131 Karlsruhe

denninger@fzi.de

Zusammenfassung

Die Fallstudie stellt die Erfahrungen der Parallelisierung eines Algorithmus zur Erstellung von Tiefenkarten aus Stereobildern vor. Dabei werden mögliche Ansatzpunkte der Parallelisierung systematisch untersucht und mehrere daraus abgeleitete Implementierungen, auf drei Systemen mit 4, 8 und 32 Kernen, bewertet und diskutiert. Für alle Systeme können Konfigurationen mit annähernd linearer Beschleunigung ermittelt werden. Allerdings lassen sich gute Konfigurationen nicht intuitiv schätzen und unterscheiden sich zudem von System zu System.

Schlüsselwörter: Multicore, Parallelisierung, Skalierung

1 Einleitung

Die Ära der Mehrkernprozessoren hat bereits begonnen. Für Desktop-PCs und Server sind heute quasi keine Einkernprozessoren mehr im Handel. Und auch in eingebettete Systeme kommen Mehrkernprozessoren immer häufiger zum Einsatz. Im Gegensatz dazu beschränkt sich das Wissen über die Parallelisierung von Algorithmen vorwiegend auf die Bereiche des numerischen Rechnens und Hochleistungsrechnens. Allerdings lassen sich diese Erfahrungen nur selten auf nicht-numerische Anwendungen übertragen. Insbesondere fehlen etablierte Vorgehensweisen für den Entwurf paralleler Anwendungen bzw. die Parallelisierung existierender Anwendungen.

Zum gegenwärtigen Zeitpunkt sind Fallstudien deshalb ein wichtiges Mittel um Erfahrungen zum Thema Parallelisierung zu sammeln und weiterzugeben. Die Verfügbarkeit eines großen Repertoires an Fallstudien aus verschiedenen Anwendungsbereichen wie z. B. Bzip2 [1] oder Desktopsuche [2] ist Voraussetzung für die Ableitung von allgemeingültigen Vorgehensweisen zur Parallelisierung von Software.

Die vorliegende Fallstudie dokumentiert Vorgehensweise, Erfahrungen und Ergebnisse der Parallelisierung eines Algorithmus zur Erstellung von Tiefenkarten aus Stereobildern. Insbesondere werden systematisch die möglichen Ansatzpunkte bei der Parallelisierung vorgestellt und diskutiert. Anhand von Messwerten – ermittelt auf drei unterschiedlichen Mehrkernsystemen zwischen 4 und 32 Kernen – werden die bei der Konzeption und Implementierung getroffenen Entscheidungen, z. B. im Hinblick auf Skalierung, bewertet. Darüber hinaus werden die Auswirkungen des Einsatzes verschiedener C++-Bibliotheken für parallele Anwendungen präsentiert. Die Fallstudie beschränkt sich auf die Betrachtung von Systemen mit gemeinsamem Speicher.

Tiefenkarten sind zweidimensionale Bilder, bei denen jeder Bildpunkte eine Tiefeninformation (meist Abstand von der Kamera) darstellt. Wird mit einem Stereokamerasystem (zwei nebeneinander angebrachten Kameras) die gleiche Szene aufgenommen, so haben die beiden Bilder folgende Eigenschaft: Objekte die sehr weit entfernt sind werden im linken und rechten Bild auf den gleichen Bildpunkt abgebildet. Objekte die näher sind werden im linken und rechten Bild auf unterschiedliche Bildpunkte abgebildet – je weiter der Abstand der Bildpunkte zwischen den beiden Bildern, umso näher ist das Objekt[1].

Die Erstellung von Tiefenkarten ist ein wichtiger Bestandteil von Systemen die ihre Umgebung wahrnehmen können. Solche „autonomen“ und „intelligenten“ Systeme werden im Laufe des nächsten Jahrzehnts rasant Einzug in den Alltag vieler Menschen halten. Zwar können Tiefenkarten auch mit Hilfe von Laser- bzw. Radarsensoren erstellt werden, allerdings bietet die Erstellung aus Stereobildern zahlreiche Synergieeffekte. So ist z. B. bei der Erkennung von Verkehrszeichen neben der Form die Textur ausschlaggebend.

Abschnitt 2 stellt den Algorithmus zur Erstellung von Tiefenkarten vor, während Abschnitt 3 möglichen Ansatzpunkte bei der Parallelisierung aufzeigt. In Abschnitt 4 werden verschiedene konkrete Implementierungen vorgestellt, die anschließend in Abschnitt 5 evaluiert und diskutiert werden. Zuletzt werden in Abschnitt 6 Schlussfolgerungen aus der Fallstudie gezogen.

2 Was kann parallelisiert werden?

Als Grundlage für die Fallstudie dient die Beispielanwendung „StereoDepthMapDemo“ aus der IVT-Bibliothek[2] [3], die aus einem Stereobildpaar eine Tiefenkarte erstellt.

Das Verfahren besteht aus sieben aufeinanderfolgenden Schritten, siehe Abbildung 1. Die einzelnen Schritte werden nachfolgend erläutert.

Abbildung 1: Verfahren zur Erzeugung von Tiefenkarten

Laden Im ersten Schritt wird das Stereobildpaar entweder von Festplatte geladen oder von einem Kamerasystem empfangen.

[1] Praxisorientierte Einführungen in Stereogeometrie bieten z. B. [3] oder [4]
[2] Integrating Vision Toolkit (IVT) – http://ivt.sourceforge.net/ (Version 1.3.9)

Graustufenkonvertierung Im zweiten Schritt werden die Kamerabilder, bei denen es sich meist um Farbbilder handelt, in Graustufenbilder umgewandelt.

Rektifizierung Bei der Rektifizierung werden Verzerrungen, die durch die Kameras sowie die Lage der beiden Kameras zueinander in den Bilder enthalten sind, korrigiert. Zuerst wird jedes Bild einzeln entzerrt – mit Hilfe der individuellen Kamerakalibrierung werde die durch die Kameras verursachten Verzerrungen korrigiert. Dies sorgt dafür, dass gerade Kanten auch im Bild als gerade Kanten abgebildet sind. Danach erfolgt die eigentliche Rektifizierung des Stereobildpaars, bei der mittels der Stereokalibrierung des Kamerasystems die Lage der beiden Stereobilder zueinander korrigiert wird. Als Ergebnis liegen Bildpunkte, die im linken und rechten Bild übereinstimmen, auch in der gleichen Bildzeile. Dies vereinfacht die Suche von Stereokorrespondenzen erheblich, da nur noch in horizontaler Richtung gesucht werden muss. Eine detaillierte Erklärung der Rektifizierung findet sich in [4].

Gaußfilter für Belichtungsinvarianz Um die Einflüsse von Belichtungsunterschieden zu reduzieren wird in diesem Schritt die „Difference of Gaussians“ auf den Bilder berechnet.

Stereokorrespodenz In diesem Schritt wird die eigentliche Stereokorrespondenz berechnet. Dazu werden für alle Bildpunkte aus einem Bild Übereinstimmungen im anderen Bild gesucht und dann der horizontale Versatz (Disparität) in Bildpunkten berechnet. Das Ergebnis ist ein Tiefenbild, dass für jeden Bildpunkte die wahrscheinlichste Disparität zwischen linken und rechtem Bild enthält. Zur Reduktion des Rechenaufwands wird die maximal zu betrachtende Disparität begrenzt.

Gradientenfilter Unabhängig von der Stereokorrespondenz werden im linken Bild schwach texturierte Bereiche erkannt und entfernt. Dazu wird jeweils für ein bestimmtes Fenster der Intensitätsgradient aufsummiert und mit einem unteren Grenzwert verglichen.

Und-Filter Durch die Und-Verknüpfung des Tiefenbildes und des Gradientenbildes werden Bereiche in denen die Bilder keine Strukturierung enthalten entfernt, da Verfahren zur Stereokorrespondenz in gering-strukturierten Bereichen zu sehr vielen Fehlern neigen.

3 Vorgehen zur Parallelisierung

Ein bei der Entwicklung von Software generell erfolgversprechender Ansatz ist die Nutzung von Entwurfsmustern. Als Standard für parallele Entwurfsmuster hat sich in den letzten Jahren das Buch von Mattson, Sanders und Massingill [5] etabliert. Darin werden zwei wichtige Schritte der Parallelisierung unterschieden: das Finden von Parallelität und die Auswahl geeigneter paralleler Algorithmen. Das Vorgehen in dieser Fallstudie orientiert sich an diesen beiden Schritten und den vorgeschlagenen parallelen Entwurfsmustern.

Finden von Parallelität Grundsätzlich ist zwischen Aufgaben- und Datenparallelität zu unterscheiden. Bei der Aufgabeparallelität werden verschiedenen Aufgaben parallel ausgeführt, während bei der Datenparallelität die gleiche Aufgabe auf verschiedene Teile einer Datenstruktur parallel ausgeführt wird. In der Praxis werden oft ähnliche Aufgaben oder gleiche Aufgaben mit unterschiedlichen Parametern auf Datenstrukturen parallel ausgeführt, so dass eine Einordnung in Aufgaben- und Datenparallelität nicht immer trivial ist. Aus der Suche nach Parallelität in der Verarbeitungskette von Abbildung 1 ergibt sich die Darstellung in Abbildung 2.

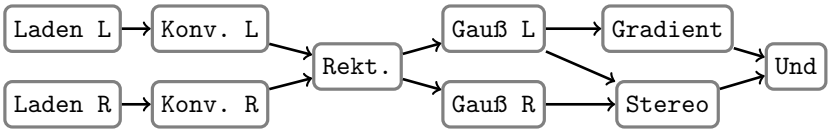

Abbildung 2: Parallelisiertes Verfahren zur Erzeugung von Tiefenkarten

Alle Schritte in denen das linke und rechte Bild unabhängig verarbeitet werden, können jeweils parallel ausgeführt werden. Darüber hinaus gibt es weitere, in der Abbildung nicht dargestellte, Möglichkeiten für Datenparallelität. So können die einzelnen Bilder zerlegt (partitioniert) und so parallel verarbeitet werden. Ein weiterer Ansatzpunkt für Datenparallelität bildet die Stereokorrespondenz, da in einer Schleife alle potentiellen Disparitäten unabhängig voneinander untersucht werden. Durch Betrachtung des Verfahrens auf höherer Ebene, lässt sich außerdem feststellen, dass mehrere Stereobildpaare parallel verarbeitet werden können, da die einzelnen Ausführungen des Verfahrens unabhängig voneinander sind.

Aufgabenparallelität ist lediglich bei der Ausführung der beiden unabhängigen Schritte Gradientenfilter und Stereokorrespondenz möglich.

Auswahl paralleler Algorithmen Mit der zuvor identifizierten Parallelität lassen sich geeignete Entwurfsmuster für parallele Algorithmen auswählen. Als erstes bietet sich das Muster *Fließband* an. Es bietet eine Kombination aus Aufgaben- und Datenparallelität, indem mehrere Stereobildpaare parallel verarbeitet werden können, wobei für jedes Bildpaar unterschiedliche Stufen des Fließbands parallel ausgeführt werden. Darüber hinaus können entsprechend Abbildung 2 in einigen Stufen linkes und rechtes Bild nach dem Muster *Parallele Aufgaben* ausgeführt werden.

Das Muster *Geometrische Zerlegung* bietet sich für die Zerlegung einzelner Bilder innerhalb eines Schrittes sowie die parallele Überprüfung der möglichen Disparitäten bei der Stereokorrespondenz an. Im Rahmen der Fallstudie wird nur die Zerlegung innerhalb einzelner Schritte untersucht, es wäre aber auch möglich die gesamte Verarbeitungskette auf Partitionen der Bilder auszuführen. Für die Erläuterung der konkreten Parallelisierungen im nächsten Abschnitt werden für Datenparallelismus die Darstellungselemente aus Abbildung 3 und für Aufgabenparallelismus die Darstellungselemente aus Abbildung 4 benutzt.

Abbildung 3 zeigt links die parallele Verarbeitung von linkem und rechtem Bild eines Bildpaares. In der Mitte ist die geometrische Zerlegung eines Bildes zu sehen, während rechts die parallele Berechnung der Disparitäten bei der Stereokorrespondenz dargestellt ist.

Abbildung 3: Darstellungselemente Datenparallelismus

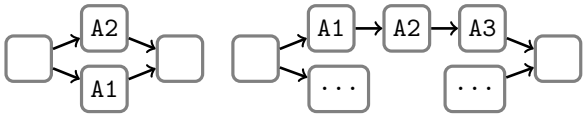

Abbildung 4: Darstellungselemente Aufgabenparallelismus

Abbildung 4 zeigt links die parallele Verarbeitung unterschiedlicher Aufgaben und rechts ein Fließband.

4 Parallelisierung

Die Anwendung für die Fallstudie ist in C++ geschrieben. Entsprechend werden drei verschiedene Bibliotheken zur Parallelisierung mit C/C++ untersucht: Pthreads (POSIX Threads) [6], OpenMP[3] [7] und Intel Threading Building Blocks (TBB)[4] [8]. Bei diesen drei Bibliotheken handelt es sich um die momentan am weitesten verbreiteten für C++.

Daneben gibt es noch Intel Cilk++[5], dass aber bisher nur geringe Verbreitung gefunden hat. Außerdem werden zunehmend Ansätze, die neben der CPU auch die GPU benutzen, eingesetzt – z. B. CUDA[6] oder OpenCL[7]. Diese Ansätze sind aber ebenfalls nicht Thema dieser Fallstudie.

Pthreads wurden 1995 standardisiert, da es bis zu diesem Zeitpunkt zahlreiche zueinander inkompatible Thread-Bibliotheken für diverse Betriebssysteme und Übersetzer gab. Pthreads werden in der Regel eins zu eins auf native Betriebssystem-Threads abgebildet, setzen also auf einer sehr niedrigen Ebene der Parallelisierung an. Entsprechend mächtig aber auf komplex sind Pthreads.

OpenMP für C++ wurde 1998 standardisiert mit dem Ziel die Parallelisierung von numerischen Anwendungen zu vereinfachen. Entsprechend ist OpenMP in erste Linie für Datenparallelität geeignet, bietet inzwischen aber auch Unterstützung für Aufgabenparallelität. Aktuell ist Version 3.0 standardisiert und in den meisten Übersetzern implementiert.

TBB wurde 2006 von Intel vorgestellt und seither kontinuierlich weiterentwickelt. Es handelt sich um eine Template-Bibliothek, die Datenstrukturen und parallele Algorithmen bietet, so dass Entwickler nicht mehr direkt mit Threads arbeiten müssen.

Als Vorbereitung auf die Umsetzung der Parallelisierung wurde zur Ermittlung kritischer Flaschenhälse ein Profiling des sequentiellen Algorithmus durchgeführt. Das Ergebnis ist, dass der Schritt Stereokorrespondenz für ca. 65% des Rechenaufwands verantwortlich ist, der Gaußfilter für 15% (2x 7,5%), der Gradientenfilter für ca. 8% und das Rektifizieren für ca. 1,5%. Die restlichen Filter

[3] OpenMP Architecture Review Board – http://openmp.org/
[4] Intel Threading Building Blocks (TBB) – http://www.threadingbuildingblocks.org/
[5] Intel Cilk++ – http://software.intel.com/en-us/articles/intel-cilk/
[6] NVIDIA CUDA – http://www.nvidia.com/object/cuda_home_new.html
[7] Open Computing Language (OpenCL) – http://www.khronos.org/opencl/

verursachen jeweils weniger als 1%.

Mit Hilfe der drei Parallelisierungs-Bibliotheken wurden verschiedene Varianten des Verfahrens zur Erstellung von Tiefenkarten umgesetzt. Die Varianten werden nachfolgend vorgestellt. Die Evaluierung der Varianten auf verschiedenen Systemen erfolgt anschließend in Abschnitt 5.

Ziel der Fallstudie ist die Vorgehensweise bei der Parallelisierung von Anwendungen aufzuzeigen. Es soll weder die maximale Leistung noch Qualität von Verfahren zur Erstellung von Tiefenkarten ermittelt werden – für beide Optimierungsziele gibt es in der Literatur zahlreiche Veröffentlichungen.

Sequentielle Variante Die sequentielle Variante folgt dem in Abbildung 1 dargestellten Ablauf. Gegenüber dem Originalbeispiel aus der IVT-Bibliothek wurde lediglich der Code für das Anzeigen der Ergebnisbilder entfernt. Die sequentielle Variante dient als Grundlage für die Berechnung der Beschleunigung der parallelisierten Varianten.

Datenparallele Variante Eine Variante des Algorithmus die ausschließlich Datenparallelität nutzt wurde mit OpenMP umgesetzt, indem Flaschenhälse mittels `#pragma omp parallel for` parallelisiert wurden. Die Umsetzung ist in Abbildung 5 schematisch dargestellt. Die parallele Verarbeitung des linken und rechten Bilds wird mittels `#pragma omp parallel sections` umgesetzt und ist auf Lade-, Konvertier- und Gaußfilter anwendbar. Die einzelnen Parameter der OpenMP-Variante sind im Anhang in Tabelle 6 aufgelistet. Zu beachten ist die `OMP_NESTED`-Umgebungsvariable, die das Verhalten bei geschachtelten `omp parallel for`-Schleifen steuert. Wird beispielsweise bei zwei geschachtelten Schleifen jeweils `num_threads(2)` gesetzt, so wird der Rumpf der inneren Schleife bei gesetzter Variable mit vier Threads parallel ausgeführt, bei nicht gesetzter Variable kommen lediglich zwei Threads zum Einsatz.

Es ist anzumerkend, dass bei der parallelen Überprüfung der möglichen Disparitäten, das Schreiben des Gesamtergebnisses nicht synchronisiert erfolgt, so dass Race-Conditions zu Fehlern führen können. Allerdings habe sich die dadurch entstehenden Fehler im Vergleich zu verfahrensbedingten Fehlern wie z. B. Verschmutzungen des Kameraobjektivs oder Rauschen des Bildsensors als vernachlässigbar erwiesen. Die Synchronisierung der Schreiboperation hat deutliche Auswirkungen auf die Geschwindigkeit. Auch die Varianten TBB und Pthreads/Fließband mit OpenMP nutzen diese nicht synchronisierte Schreiboperation.

Abbildung 5: Parallelisierung mit OpenMP

Aufgabenparallel Variante Mit Hilfe von Pthreads wurde eine Fließbandvariante umgesetzt, wobei jede Stufe durch einen oder mehrere Threads ausgeführt wird. Laden und Konvertieren sowie Gradient- und Und-Filter werden jeweils in einer Stufe ausgeführt, siehe Abbildung 6. Beide Stufen werden aufgrund ihres geringen Aufwands mit maximal einem Thread ausgeführt. Die verbleibenden

Stufen Rektifizieren, Gaußfilter und Stereokorrespondenz können mit mehreren Threads ausgeführt werden. Die Parameter dieser Variante sind im Anhang in Tabelle 9 aufgelistet.

Ein wichtiger Faktor bei der Nutzung eines Fließbands ist die Latenz, also die Zeitspannen zwischen Verfügbarkeit eines Bildpaares und dem Abschluss der Verarbeitung. Ein Fließband erhöht zwar den absoluten Durchsatz an Bilder, gleichzeitig steigt aber auch die Latenz. Da im Rahmen dieser Fallstudie keine Echtzeitanforderungen betrachtet werden, wurde der Grenzwert der maximal gleichzeitig im Fließband zugelassenen Bildpaare auf acht begrenzt. Neben der sequentiellen ist nur die OpenMP-Variante nicht von der erhöhten Latenz betroffen.

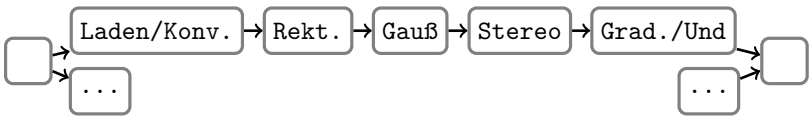

Abbildung 6: Parallelisierung mit Pthreads/Fließband

Gemischte Varianten Neben den jeweils rein Aufgaben- bzw. Datenparallelen Varianten wurden drei gemischte Varianten umgesetzt. Abbildung 7 zeigt eine TBB-Variante mit Fließband, wobei Laden und Konvertieren zu einer Fließbandstufe zusammengefasst wurden. Die genauen Parameter der TBB-Variante können im Anhang Tabelle 7 entnommen werden.

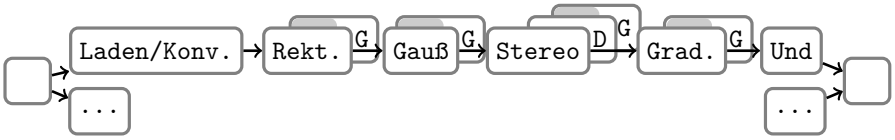

Abbildung 7: Parallelisierung mit TBB

Als Vergleich zur sowohl Aufgaben- als auch Datenparallelen TBB-Variante wurden die Pthread/Fließband- und OpenMP-Variante zu einer gemeinsamen „Pthread/Fließband mit OpenMP-Variante“ kombiniert. Die zugehörigen Parameter setzen sich entsprechend aus Tabelle 9 und Tabelle 6 zusammen und sind im Anhang in Tabelle 10 zur besseren Übersicht nochmals angegeben.

Als letzte Variante wurde eine Pthreads-Variante umgesetzt, die den Code der sequentiellen Variante mit mehreren Threads ausführt, so dass mehrere Bildpaare parallel verarbeitet werden, wobei jeder Thread ein Bildpaar vollständig verarbeitet. Diese Variante ist in Abbildung 8 schematisch dargestellt und entspricht dem Konzept eines Threadpools, wobei die Ausgabereihenfolge der Tiefenbilder sichergestellt ist. In Tabelle 8 im Anhang ist der einzige notwendige Parameter angegeben – die Anzahl der Threads.

5 Evaluierung

Zur Bewertung der Parallelisierung wurde für jede Variante die beste Konfiguration auf drei verschiedenen Systemen ermittelt. Dazu kamen ein 4-Kern-Desktop, eine 8-Kern-Workstation sowie ein durch das Intel Manycore Testing

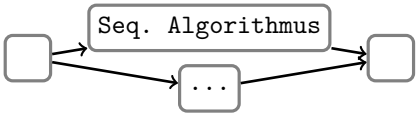

Abbildung 8: Parallelisierung mit Pthreads

Lab (MTL)[8] verfügbares 32-Kern-System zum Einsatz. Für die Evaluierung wurden Bilder mit 1296x966 Bildpunkten benutzt. Die jeweils beste ermittelte Konfiguration wird nachfolgend angegeben.

5.1 Ergebnisse

Die angegebenen Konfigurationen sind nicht zwingend optimal, da nicht der komplette Suchraum abgedeckt werden konnte. Für die OpenMP-Variante auf dem 8-Kernsystem ergibt sich beispielsweise folgende Eingrenzung der zu überprüfenden Konfigurationen: Generell sollten nicht mehr OpenMP-Threads gleichzeitig erzeugt werden als Kerne vorhanden sind. Für die Parameter lassen sich folgende Werte eingrenzen: für den 1., 2. und 3. Parameter jeweils 1,2,4 und 8 und für den 4., 5. und 6. Parameter jeweils 0 und 1. Daraus ergeben sich 512 zu prüfende Konfigurationen. Um aussagekräftigen Wert zu erhalten müssen mindestens 50 Bilder pro Konfiguration verarbeiten werden. Bei einer geschätzten durchschnittlichen Bildrate von 2,5 Bilder/s, bezogen auf alle Konfigurationen, dauert jede Konfiguration ungefähr 20 Sekunden – alle 512 Konfigurationen entsprechend 170 Minuten.

Da der 1. und 2. Parameter nicht unabhängig sind, lassen sich die möglichen Konfigurationen auf 320 einschränken.

Insgesamt wurden auf den 4- und 8-Kern-Systemen deutlich mehr Konfigurationen getestet als auf dem 32-Kern-System. Für die nachfolgend angegebenen Konfigurationen wurden jeweils fünf Durchläufe mit 1000 Bilder gemessen. Anschließend wurden jeweils Mittelwert sowie mittlere Abweichung bestimmt.

Die genaue Bedeutung der Konfigurationen kann im Anhang in den Tabellen 6 bis 10 nachgelesen werden.

4-Kern-System Das 4-Kern-System besteht aus einem Prozessor Intel Core-i7-860 (2,8 GHz) mit TurboBoost und Hyperthreading. Für die Messung der Ergebnisse kam Ubuntu 10.04 32bit mit GCC 4.4.3 und TBB 2.2 zum Einsatz.

Das 4-Kern-System wurde zusätzlich ohne Hyperthreading (HT) getestet (im BIOS deaktiviert). Laut [9] wird durch HT eine Beschleunigung von bis zu 30% erreicht. Im Beispiel führte HT zu einer Beschleunigung von bis zu 19%. Es ist zu beobachten, dass die Beschleunigung durch HT um so höher ist, je geringer die Ausnutzung der Parallelität der Variante an sich ist. Die größte Beschleunigung ergibt sich bei der Variante Pthreads/Fließband, die sonst vergleichsweise schlecht abschneidet.

8-Kern-System Das 8-Kern-System besteht aus zwei 4-Kern-Prozessoren Intel Xeon E5410 (2,33 GHz). Die Ergebnisse wurden unter Ubuntu 10.04 32bit mit GCC 4.4.3 und TBB 2.2 ermittelt.

[8] Intel Manycore Testing Lab (MTL) – http://software.intel.com/en-us/articles/intel-many-core-testing-lab/

Variante	Konfiguration	Bilder/s (mittl. Abw.)	Beschleunigung
Sequentiell	-	1,12 (<0,01)	-
OpenMP	(1,4,4,0,0,1)	3,42 (<0,01)	3,1
TBB	(6,0,1,0,4,0)	3,76 (<0,01)	3,4
Pthreads	(4)	3,76 (<0,01)	3,4
Pthreads/Fließband	(8,1,1,6)	2,49 (0,02)	2,2
Pthreads/Fließband mit OpenMP	(8,1,1,6,1,4,4,0,0,1)	2,99 (<0,01)	2,7

Tabelle 1: Messwerte 4-Kern-System

Variante	Konfiguration	Bilder/s (mittl. Abw.)	Beschleunigung (Vgl. ohne HT)
Sequentiell	-	1,12 (<0,01)	-
OpenMP	(1,8,8,0,0,1)	3,79 (<0,01)	3,4 (+11%)
TBB	(8,1,1,0,4,0)	4,09 (<0,01)	3,7 (+9%)
Pthreads	(8)	4,01 (<0,01)	3,6 (+7%)
Pthreads/Fließband	(8,1,1,6)	2,97 (<0,01)	2,7 (+19%)
Pthreads/Fließband mit OpenMP	(8,1,1,4,1,4,8,0,0,1)	3,33 (<0,01)	3,0 (+11%)

Tabelle 2: Messwerte 4-Kern-System (mit Hyperthreading)

32-Kern-System Das 32-Kern-System besteht aus vier 8-Kern-Prozessoren Intel Xeon X7560 (2,27 GHz) mit TurboBoost und Hyperthreading. Als Betriebssystem kam RedHat RHEL 5.4 64bit mit GCC 4.1.2 und TBB 3.0 zum Einsatz.

Bei den Ergebnissen in Tabelle 4 wurden Varianten die ein Fließband nutzen, auf maximal acht gleichzeitig im Fließband enthaltene Bildpaare begrenzt. Da somit nicht der optimale Durchsatz dieser Varianten ermittelt werden kann, sind in Tabelle 5 die Ergebnisse ohne Beschränkung angegeben. Bei den Varianten TBB und Pthreads waren deutliche Verbesserungen möglich, bei der Variante Pthreads/Fließband nur leichte. Für die Variante Pthreads/Fließband mit OpenMP konnte keine bessere Konfiguration ermittelt werden.

Variante	Konfiguration	Bilder/s (mittl. Abw.)	Beschleunigung
Sequentiell	-	0,77 (<0,01)	-
OpenMP	(1,8,8,0,0,1)	4,89 (0,04)	6,4
TBB	(8,0,4,0,2,0)	5,27 (0,02)	6,8
Pthreads	(8)	4,22 (<0,01)	5,5
Pthreads/Fließband	(8,1,1,6)	3,58 (0,02)	4,7
Pthreads/Fließband mit OpenMP	(8,1,2,4,1,8,2,0,0,1)	4,46 (<0,01)	5,8

Tabelle 3: Messwerte 8-Kern-System

Variante	Konfiguration	Bilder/s (mittl. Abw.)	Beschleunigung
Sequentiell	-	1,14 (0,02)	-
OpenMP	(4,8,32,0,0,1)	12,61 (0,02)	11,1
TBB	(8,1,2,1,0,0)	18,39 (0,23)	16,1
Pthreads	(8)	8,72 (0,02)	7,6
Pthreads/Fließband	(8,4,2,8)	8,43 (0,02)	7,4
Pthreads/Fließband mit OpenMP	(8,1,2,4,2,8,8,0,0,1)	35,95 (0,17)	31,5

Tabelle 4: Messwerte 32-Kern-System

Variante	Konfiguration	Bilder/s (mittl. Abw.)	Beschleunigung
TBB	(50,0,1,1,0,0)	28,87 (0,54)	25,3
Pthreads	(60)	26,06 (2,42)	22,9
Pthreads/Fließband	(30,20,20,40)	11,91 (0,34)	10,4

Tabelle 5: Messwerte 32-Kern-System (unbeschränktes Fließband)

5.2 Diskussion

Auf allen Systemen lassen sich annähernd lineare Beschleunigungen erreichen. Bei vier bzw. acht Kernen erzielt bereits die Ausnutzung von Datenparallelität durch die Parallelisierung von Schleifen sehr gute Ergebnisse. Auf dem System mit 32 Kernen skalieren Ansätze, die ein Fließband benutzen, deutlich besser. Wobei die maximale Leistung nur erreicht wird, wenn die einzelnen Fließbandstufen zusätzlich Datenparallelität ausnutzen.

Bei der Ermittlung der Werte wurden alle Bilder aus dem Hauptspeicher geladen, da dies dem Empfang von einem Kamerasystem entspricht. Beim Laden aufgezeichneter Bilder von Festplatte und Verarbeitung mittels Fließband kann auf dem 8-Kern-System eine superlineare Skalierung erreicht werden, da die langsamen Leseoperationen verdeckt werden.

5.3 Aufwand

Bei den nachfolgenden Betrachtungen zum Aufwand der Parallelisierung handelt es sich um Einschätzungen, die weder durch Messung von Zeitdauer noch Codeumfang bestätigt sind.

Der Aufwand für die Parallelisierung mittels OpenMP ist vergleichsweise gering. Voraussetzung dafür ist allerdings, dass der Code Schleifen mit unabhängigen Schleifendurchläufen enthält und der Code im Schleifenrumpf umfangreiche Berechnungen ausführt. Besonders sorgfältig müssen geteilte Datenstrukturen behandelt werden. Für geringe Kernzahlen lohnt sich OpenMP, bei großen Kernzahlen ist die Skalierung allerdings eingeschränkt, da nach Amdahls Gesetz [10] die sequentiellen Teile des Algorithmus überwiegen.

Der Aufwand für Parallelisierung mittels TBB ist deutlich größer. Methoden müssen in neue Klassen ausgelagert werden – unabhängig davon ob ein Fließband umgesetzt oder eine Schleife parallelisiert wird (mit Hilfe von Lambda-

Ausdrücken des neuen C++0x-Standards kann dieser Aufwand allerdings reduziert werden). Im Gegenzug wird der Entwickler durch zahlreiche parallele Algorithmen und Datenstrukturen unterstützt.

Sind einzelne Durchläufe eines Algorithmus unabhängig, so lässt er sich relativ einfach mit mehreren Threads parallel starten. Es muss lediglich am Ende die Reihenfolge bei der Bereitstellung der Ergebnisse sichergestellt werden.

Bei der Realisierung eines Fließbands mit Pthreads, muss neben der Reihenfolge nach jeder Stufe auch die Einhaltung der maximalen Anzahl an Bildern im Fließband sichergestellt werden. Dazu müssen explizit Puffer zwischen den Stufen umgesetzt werden, die auch bei parallelen Zugriffen fehlerfrei funktionieren. Der Aufwand hierfür ist beträchtlich und die Umsetzung fehleranfällig.

6 Schlussfolgerungen

Parallele Entwurfsmuster sind ein guter Einstiegspunkt in die Thematik der Parallelisierung. Für die heute verbreiteten Systeme mit wenigen Kernen, ist das im Bereich der Softwareentwicklung übliche Vorgehen Flaschenhälse zu optimieren ausreichend. Einfache Datenparallelität wie z.B. Schleifenparallelisierung mittels OpenMP ist vergleichsweise einfach umzusetzen und bietet bereits eine gute Leistungssteigerung.

Schon in wenigen Jahren werden aber Systeme mit 32 oder noch mehr Kernen an der Tagesordnung sein. Um auch diese Systeme optimal zu nutzen ist erheblich mehr Parallelisierungsaufwand notwendig. Es spricht deshalb einiges dafür dies bereits heute bei der Entwicklung von Software zu berücksichtigen.

Bibliotheken mit Algorithmen auf hoher Abstraktionsebene machen die Entwicklung einfacher und weniger fehleranfällig. Diese Unterstützung kann alternativ auch durch Programmiersprachen erfolgen.

Die Ermittlung gute Konfigurationen der im Rahmen dieser Fallstudie umgesetzten Variante des Algorithmus zur Erstellung von Tiefenkarten war sehr zeitaufwändig und vor allem für jedes Zielsystem erneut durchzuführen. Mit dieser Problematik werden sich in Zukunft viele Entwickler von Software auseinandersetzen müssen, da für gute Skalierung viele Freiheitsgrade zur Parallelausführung in einer Anwendung notwendig sind, aber gute Konfigurationen schwer zu ermitteln sind. Abhilfe verspricht Auto-Tuning, allerdings stehen entsprechende Techniken noch am Anfang ihrer Entwicklung.

Literatur

[1] Victor Pankratius, Ali Jannesari und Walter F. Tichy. Parallelizing Bzip2: A Case Study. In *Multicore Software Engineering*, IEEE Software, Vol. 26 (6), S. 70-77, Nov. 2009

[2] David J. Meder und Walter F. Tichy. *Parallelizing an Index Generator for Desktop Search.* Technischer Bericht, IPD, Universität Karlsruhe, 2010

[3] Pedram Azad, Tilo Gockel und Rüdiger Dillmann. *Computer Vision – Das Praxisbuch.* Elektor-Verlag, 2007

[4] Gary Bradski und Adrian Kaehler. *Learning OpenCV – Computer Vision with the OpenCV Library.* O'Reilly, 2008

[5] Timothy G. Mattson, Beverly A. Sanders und Berna L. Massingill. *Patterns for parallel programming.* Addison-Wesley, 2004

[6] Bradford Nichols, Dick Buttlar und Jacqueline Proulx Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly, 1996.

[7] Barbara Chapman, Gabriele Jost und Ruud van der Pas. *Using OpenMP: portable shared memory parallel programming.* MIT Press, 2008.

[8] James Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007

[9] Shameem Akhter und Jason Roberts. *Multicore Programmmierung.* Intel PRESS, 2008

[10] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, Vol. 30, S. 483–485, 1967

A Anhang – Parameterübersicht

Tabelle 6 enthält die Parameter der OpenMP-Variante.

Nr.	Typ	Bedeutung
1	int	Anzahl der OpenMP-Threads für die geometrische Zerlegung bei der Stereokorrespondenz
2	int	Anzahl der OpenMP-Threads für die Berechnung der Disparitäten bei der Stereokorrespondenz
3	int	Anzahl der OpenMP-Threads für die geometrische Zerlegung im Gauß-, Rektifizier- und Gradientenfilter
4	bool	Lade und Konvertiere linkes/rechtes Bild parallel
5	bool	Gaußfilter auf linkes/rechtes Bild parallel ausführen (falls aktiv, wird Parameter 3 für den Gaußfilter halbiert)
6	bool	`OMP_NESTED`-Umgebungsvariable gesetzt

Tabelle 6: Parameter der OpenMP-Variante

Tabelle 7 enthält die Parameter der TBB-Variante. Der Faktor zu Berechnung der `grainsize` teilt die Anzahl der Schleifendurchläufe und ermittelt so die beim Aufruf von `parallel_for` übergebene `grainsize` – d. h. wie viele Schleifendurchläufe TBB bei der Ausführung zusammenfassen soll.

Die Tabellen 8, 9 und 10 enthalten die Parameter der drei mittels Pthreads umgesetzten Varianten.

Nr.	Typ	Bedeutung
1	int	Maximale Anzahl an Bildern im Fließband
2	bool	Fließbandstufen Laden/Konvertieren und Rektifizieren sowie Gradienten- und Und-Filter zusammenfassen
3	int	Anzahl der Partitionen bei der Stereokorrespondenz
4	bool	Nutzung von `parallel_for`-Schleifen beim Rektifizieren, Gauß- und Gradientenfilter
5	int	Faktor zur Berechnung der `grainsize` der Schleifen
6	bool	Nutzung von `parallel_for` bei der Stereokorrespondenz

Tabelle 7: Parameter der TBB-Variante

Nr.	Typ	Bedeutung
1	int	Maximale Anzahl an Bildern die parallel verarbeitet werden

Tabelle 8: Parameter der Pthreads-Variante

Nr.	Typ	Bedeutung
1	int	Maximale Anzahl an Bildern im Fließband
2	int	Anzahl der Threads für Rektifizieren
3	int	Anzahl der Threads für Gaußfilter
4	bool	Anzahl der Threads für Stereokorrespondenz

Tabelle 9: Parameter der Pthreads/Fließband-Variante

Nr.	Typ	Bedeutung
1	int	Maximale Anzahl an Bildern im Fließband
2	int	Anzahl der Threads für Rektifizieren
3	int	Anzahl der Threads für Gaußfilter
4	bool	Anzahl der Threads für Stereokorrespondenz
5	int	Anzahl der OpenMP-Threads für die geometrische Zerlegung bei der Stereokorrespondenz
6	int	Anzahl der OpenMP-Threads für die Berechnung der Disparitäten bei der Stereokorrespondenz
7	int	Anzahl der OpenMP-Threads für die geometrische Zerlegung im Gauß-, Rektifizier- und Gradientenfilter
8	bool	Lade und Konvertiere linkes/rechtes Bild parallel
9	bool	Gaußfilter auf linkes/rechtes Bild parallel ausführen (falls aktiv, wird Parameter 3 für den Gaußfilter halbiert)
10	bool	`OMP_NESTED`-Umgebungsvariable gesetzt

Tabelle 10: Parameter der Pthreads/Fließband-Variante mit OpenMP