Methoden des parallelen Postprocessing numerischer Strömungssimulationsdaten für die echtzeitfähige Visualisierung und Interaktion in VR-basierten Arbeitsumgebungen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Andreas Gerndt

aus Berlin

Berichter: Universitätsprofessor Christian H. Bischof, Ph. D.
Universitätsprofessor Dr. Rüdiger Westermann

Tag der mündlichen Prüfung: 20. Januar 2006

Vorwort

Ohne die tatkräftige Unterstützung von Diplomanden und studentischen Hilfskräften wäre es kaum möglich gewesen die dieser Dissertation zugrunde liegenden zeitintensiven Implementierungen und Evaluierungen der Ergebnisse durchzuführen. Daher möchte ich mich für das Zustandekommen vor allem bei meinen Diplomanden Christoph Langewisch, Bernd Hentschel, Marc Wolter, Samuel Sarholz und Christian Jansen bedanken. Mein Dank gilt aber auch meinem Studienarbeiter Mark Asbach und den an dem Projekt beteiligten studentischen Hilfskräften Uwe Scholl und Ben Sartor.
Ich danke ferner dem Leiter der VR-Gruppe am Rechen- und Kommunikationszentrum der RWTH Aachen Dr. Torsten Kuhlen. Er schaffte die notwendigen Freiräume und unterstützte mich in meiner wissenschaftlichen Arbeit.
Das hier vorgestellte Framework für paralleles Postprocessing ist ein Bestandteil der von der gesamten VR-Gruppe entwickelten Software mit dem Namen ViSTA. Der Erfolg der eigenen Arbeit wurde somit von allen meinen Kollegen und den an diesem Basisprojekt beteiligten Studenten befördert. Daher möchte ich mich ebenfalls bei ihnen für ihre Unterstützung und für weit reichende Anregungen bedanken.
Schließen möchte ich dieses Vorwort mit einem herzlichen Dank an Professor Christian Bischof, der mir die Möglichkeit zur Promotion eröffnete und durch zahlreiche Anmerkungen zur Verbesserung meiner Dissertation beitrug, sowie an Professor Dr. Rüdiger Westermann für die Übernahme des Koreferats.

Motivation

Die Motivation für diese Arbeit entsprang aus den ersten Erfahrungen mit dem Postprocessing von Ergebnissen aus Strömungssimulationen (engl.: *Computational Fluid Dynamics*, CFD) an einer Holobench des Rechen- und Kommunikationszentrums der RWTH Aachen. Bereits einfache Datensätze mussten erst durch zeitraubende Voruntersuchungen analysiert werden. Daten, die dann explizit für die Visualisierung aufbereitet wurden, erlaubten keinerlei interaktive Exploration. Zudem dauerte das Starten dieser individuell zusammengestellten Anwendung immer noch einige Minuten. Lief dann endlich die VR-Applikation, fiel die Bildwiederholungsrate unter die erträgliche Schwelle. An eine wirklich interaktive Exploration beliebiger Datensätze war nicht zu denken.
Das erklärte Ziel dieser Arbeit war daher die Verarbeitung wirklich großer Datensätze mithilfe eines universellen Postprocessing-Frameworks. Das Hauptaugenmerk lag auf der Behandlung instationärer Strömungen unter Aufrechterhaltung einer echtzeitigfähigen Interaktion in virtuellen Umgebungen. Eine nahe liegende Lösung zur Beschleunigung des Postprocessings liegt in der Parallelisierung von zeitaufwändigen Berechnungsverfahren. Daher wurde in einem ersten Schritt ein paralleles Framework mit dem Namen Viracocha entwickelt. Besonderen Wert wurde auf ein flexibles Design gelegt. Die Unterteilung in mehrere logische Schichten ermöglicht den Einsatz von Viracocha auch für andere Gebiete als der Strömungsanalyse. Hierfür muss lediglich die oberste Schicht, die so genannte Algorithmusschicht, ersetzt werden. Damit dies schnell und einfach geschehen kann, wurden Verwaltungs- und Kommunikationsfunktionalitäten in darunter liegenden Schichten gekapselt und durch wenige übersichtliche Schnittstellen zur Verfügung gestellt. Somit können neue Algorithmen schnell integriert werden, ohne sich um die eigentliche Parallelisierung kümmern zu müssen.
Neben der Anforderung der schnellen Erweiterbarkeit durch neue Algorithmen stand das Ziel der beliebigen Skalierbarkeit. So lässt sich Viracocha sowohl auf geclusterten SMP-Knoten als auch auf PC-Clustern mit verteilten Speichersystemen einsetzen. Die zu verwendende Anzahl von Berechnungsknoten ist beliebig wählbar. Letztendlich ist der Einsatz auf einer Einzelworkstation ebenso möglich. Da jedoch das Hauptaugenmerk auf der Behandlung sehr

großer Datensätze lag, wurden vorrangig Mehrprozessorsysteme für die Analyse des Framework-Designs berücksichtigt.
Datengröße und Datenstrukturen spielten eine wesentliche Rolle in der Entwicklung und Bewertung von Parallelisierungsansätzen für das CFD-Postprocessing. Obwohl das Framework beliebige CFD-Datensätze verarbeiten können sollte, lag der Schwerpunkt auf der Behandlung von Multi-Block-Strukturen. Diese werden recht häufig in der Vernetzung von Strömungsfeldern eingesetzt und lassen sich aufgrund der bereits verteilten Datenstruktur gut für Parallelisierungsstrategien einsetzen. Weitergehende Partitionierungsverfahren, z. B. für unstrukturierte Datensätze, werden in der vorliegenden Arbeit nicht untersucht. Für die Systemanalyse wurden neben Standardextraktionsalgorithmen, wie Schnittflächen-, Isoflächen- und Stromlinienberechnungen, auch instationäre Partikelverfolgung, Wirbelextraktionsverfahren und vektorfeldtopologische Methoden implementiert.
Es stellte sich schnell heraus, dass die mögliche Berechnungsbeschleunigung durch Parallelisierung in der Regel durch zeitintensives Nachladen von Daten erheblich begrenzt wird. Es traten deutliche Balancierungsprobleme auf und das System skalierte nur schlecht. Um diesen Flaschenhals in den Griff zu bekommen, wurde ein weit reichendes, zentral verankertes Datenmanagement, das *Viracocha Data Management System* (VDMS), entwickelt. Dieses basiert vorwiegend auf Caching- und Prefetching-Mechanismen, greift aber auch auf Wahrscheinlichkeitsaussagen per Markov-Ketten zurück. Die Bewertung des Datenmanagements bezüglich unterschiedlicher Datensätze und Extraktionsalgorithmen wird ausführlich dargestellt.
Durch Parallelisierung und innovatives Datenmanagement konnte ein gut skalierendes und balanciertes System entwickelt werden. Zudem wurde durch die Entkopplung der Berechnung eine entscheidende Entlastung des VR-basierten Visualisierungsmoduls ViSTA FlowLib erreicht. Aber von der Möglichkeit interaktiv mit den Daten arbeiten zu können, war man doch noch ein gutes Stück entfernt. Dafür lag zwischen Berechnungsanforderung und Bereitstellung der Resultate immer noch zu viel Zeit. Daher wurde zur Überbrückung der Wartezeit das so genannte Daten-Streaming zwischen Visualisierungsrechner und Parallelisierungsrechner integriert, das vor allem mithilfe von Multiresolution-Techniken schneller erste Ergebnisse präsentieren kann.
Alle integrierten Elemente wurden in dieser Arbeit auf ihre Einsatztauglichkeit für das CFD-Postprocessing überprüft. Experimentelle Messungen belegen die Effizienz der Ansätze, zeigen aber auch Nachteile und Grenzen auf. Zusammenfassend kann festgehalten werden, dass die vorliegende Arbeit darstellt, welche Möglichkeiten der Verteilung und Parallelisierung für VR-basierte Explorationssysteme großer Datensätze existieren. Mit Viracocha und ViSTA FlowLib steht nun am Rechen- und Kommunikationszentrum der RWTH ein erster leistungsfähiger CFD-Postprocessor für den Einsatz in virtuellen Umgebungen zur Verfügung, der als Basis weitergehender Forschungsprojekte dienen kann.

Kurzfassung

Mit zunehmender Verfügbarkeit leistungsfähiger Rechnersysteme wuchs die Bedeutung der numerischen Strömungssimulation. Das Postprocessing extrahiert charakteristische Strukturen und Merkmale aus den abstrakten Simulationsdaten, die mithilfe der wissenschaftlichen Visualisierung einen Überblick über die Strömung geben können. Dabei ist die explorative Analyse in virtuellen Umgebungen besonders gut geeignet die zugrunde liegenden Phänomene zu erforschen.
In der vorliegenden Arbeit wird ein verteiltes System vorgestellt, das die Interaktivität innerhalb virtueller Umgebungen auch dann gewährleistet, wenn sehr große, instationäre Strömungsdaten untersucht werden. Durch die Auslagerung datenintensiver Bearbeitungsschritte auf einen Hochleistungsrechner stehen dem Visualisierungsrechner alle Systemressourcen für die interaktive Darstellung der Extraktionsergebnisse zur Verfügung.
Neben der Beschreibung des flexiblen Designs des Parallelisierungssystems, werden zwei Aspekte ausführlicher dargestellt. Zum einen wurde ein Datenmanagementsystem implementiert, das Balancierungs- und Skalierungsprobleme, die durch das Nachladen von benötigten Daten verursacht werden, mithilfe von Caching- und Prefetching-Strategien beseitigt. Der zweite Aspekt ist das Daten-Streaming, das die Zeit, die ein Anwender auf erste Daten einer angeforderten Berechnung warten muss, minimiert. Das System sendet bereits erste Zwischenergebnisse zur virtuellen Umgebung noch bevor das endgültige Ergebnis vorliegt. Dabei wird auch auf Multiresolution-Ansätze zurückgegriffen.

Abstract

Because of the steadily increasing performance of supercomputers, computational fluid dynamics (CFD) simulations are capable of producing constantly growing amounts of raw data. These data sets are essentially useless without subsequent post-processing. One particularly attractive evaluation approach is the interactive exploration within virtual environments. However, common visualization systems are not able to process large data sets while maintaining real-time interaction and visualization at the same time. Therefore, the obvious idea is to decouple flow feature extraction from visualization.
The work presented mainly covers the functionality of the parallel CFD post-processing toolkit Viracocha. The distributed framework architecture relieves the visualization host by moving all time-consuming computation tasks to the parallelization backend. This makes it easier to guarantee real-time interaction within virtual environments. Additionally, the response time a user has to wait before requested post-processing results are visualized can be substantially reduced by optimized extraction algorithms adapted to parallel environments.
Two further aspects are discussed in more detail. A considerable bottleneck in parallelization of CFD post-processing is the time needed to load large data sets. Therefore, a first extension of Viracocha aims at the reduction of the loading time, by implementing strategies mainly based on data caching and prefetching. The second aspect concerns an approach called data streaming, which minimizes the time a user has to wait for first results of a requested extraction. In order to achieve this, Viracocha sends back coarse intermediate data to the virtual environment before the final result is available. Some implemented streaming strategies also make use of multi-resolution data structures.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listing-Verzeichnis

1 Einleitung

1.1 CFD-Postprocessing und Virtual Reality

Mit der zunehmenden Verfügbarkeit leistungsfähiger Rechnersysteme wuchs die Bedeutung der numerischen Strömungssimulation (engl.: *Computational Fluid Dynamics*, CFD) [SPUR96, LOMA98, SCHR00b]. Allerdings sind die erzeugten Simulationsdatensätze nicht direkt auswertbar. So extrahiert erst das so genannte Postprocessing charakteristische Strukturen und Merkmale, die durch geeignete visuelle Repräsentanten mithilfe der wissenschaftlichen Visualisierung einen Überblick über diese abstrakten Daten geben können.
Die weiterhin steigenden Rechenkapazitäten im Bereich des *High Performance Computing* (HPC) führen zu immer detaillierteren und exakteren Ergebnissen. Andries van Dam et al. weisen allerdings in [DAM00] auf das Problem hin, dass die so gewonnenen Rohdaten wesentlich schneller wachsen als die Fähigkeit zur Weiterverarbeitung und Analyse. Langfristig lässt sich dieses Problem laut Autoren nur durch den Einsatz von Methoden der künstlichen Intelligenz lösen, die eine vollständig automatisierte Vorverarbeitung der Rohdaten ermöglichen. Für den kurz- und mittelfristigen Zeitraum schlagen sie den Einsatz der *Immersive Virtual Reality* (IVR) vor. Durch geeignete Systeme soll ein möglichst vollständiges „Eintauchen“ in die virtuelle, computergenerierte Welt ermöglicht werden, um dem Simulationsdatensatz zugrunde liegende Phänomene interaktiv erforschen zu können.
Diese auch als explorative Analyse [SCHU00] bezeichnete Vorgehensweise untersucht die Simulationsergebnisse allein auf Basis der Rohdaten mit dem Ziel, so schnell wie möglich charakteristische Strukturen zu entdecken, Hypothesen aufzustellen oder anderweitige, weit reichende Erkenntnisse zu gewinnen. Grundlegende Arbeitsweise ist ein *Trial-and-Error*-Verfahren. Daten, die anhand vorgegebener initialer Parameter extrahiert wurden, werden häufig wieder verworfen, da sie den Erwartungen nicht entsprechen. Nach erfolgter Modifikation der Parameter wird das ausgewählte Extraktionsverfahren erneut gestartet. Dieses iterative Verfahren wird solange durchgeführt, bis die Extraktionsergebnisse einer individuellen Erwartungshaltung entsprechen.
Dieser explorative Ansatz lebt ganz erheblich von der Interaktivität des verwendeten Systems. Für virtuelle Umgebungen lassen sich im Wesentlichen zwei Interaktionskriterien definieren, die ein flüssiges Arbeiten sicherstellen sollen:

1. **Mindest-Bildwiederholungsrate**: Die Applikation darf eine minimale Bildwiederholungsrate, die nach Bryson [BRYS96] mindestens 10 Hz betragen und nach Kreylos [KREY03] sogar über 30 Bilder pro Sekunde liegen sollte, nicht unterschreiten.
2. **Maximal-Systemreaktionszeit**: Sowohl Bryson als auch Kreylos haben zudem die Reaktionszeit des VR-Systems nach einer Benutzereingabe untersucht. Dabei kamen beide unabhängig voneinander zu dem Ergebnis, dass der Benutzer maximal 100 Millisekunden bis zum erfolgten Feedback toleriert.

Diese Grenzen sind in einer virtuellen Umgebung unbedingt einzuhalten. Das Unterschreiten der Mindestbildwiederholungsrate führt unweigerlich zu einem als unangenehm empfundenen „Ruckeln“. Das Überschreiten der Maximalreaktionszeit kann darin resultieren, dass der Benutzer ausgeführte Aktionen wiederholt oder präsentierte Systemveränderungen nicht mehr unverzüglich zuordnen kann.
Die hier angeschnittenen Aspekte werden vor allem von Wissenschaftlern untersucht, die sich ausgiebig mit interaktiven Mensch-Maschine-Schnittstellen (engl.: *Human Computer*

Interfaces, HCI) beschäftigen. Für das Anwendungsfeld der interaktiven Exploration von Simulationsdaten kommt eine weitere Zeitgröße hinzu:

3. **Maximale Sekundärreaktionszeit**: Sie gibt an, bis wann Berechnungsergebnisse spätestens vorliegen müssen, damit der Anwender eine vergangene Berechnungsanforderung noch mit eintreffenden Daten in Beziehung setzen kann.

Dieser Bewertungsmaßstab ist besonders wichtig für rechenintensive Anwendungen wie dem in dieser Arbeit entwickelten parallelen Postprocessing-System. Die maximale Sekundärreaktionszeit liegt deutlich höher als die mit 100 Millisekunden angegebene maximale Systemreaktionszeit. Wird sie überschritten, kann sich im schlimmsten Fall bereits vor dem Eintreffen von Ergebnissen beim Anwender die Annahme einstellen, dass ein fehlerhafter Zustand des Systems eingetreten ist.

1.2 Verteiltes Postprocessing-System

HPC-Systeme und Visualisierungsrechner sind für ihren jeweiligen Einsatzbereich optimiert. Während Hochleistungsrechner über einen großen Hauptspeicher, viele Prozessoren und eine schnelle Anbindung an Sekundärspeicher für die Deckung des Ressourcenbedarfs paralleler Berechnungsprogramme (z. B. Strömungslöser) verfügen, besitzen Visualisierungssysteme optimierte Graphik-Subsysteme, die vor allem für das aufwändige Rendern komplexer graphischer Szenen geeignet sind. Es liegt nun nahe, dass der Hochleistungsrechner, der bereits die CFD-Rohdaten erzeugt hat, auch für das Postprocessing ausreichend Rechen- und Speicherkapazitäten zur Verfügung stellen kann. Der zentrale Ansatz dieser Arbeit ist es daher alle Extraktionsberechnungen auf Parallelrechner auszulagern. Durch diese Entkopplung stehen auf dem Visualisierungsrechner alle Systemressourcen für die Darstellung der Extraktionsergebnisse zur Verfügung. Zudem sind üblicherweise die Visualisierungsdaten um Faktoren kleiner als die Ausgangsdaten.
Durch die Verteilung von Komponenten wird die Einhaltung der oben beschriebenen Interaktionskriterien deutlich erleichtert. Das Visualisierungssystem kann nun eher die geforderte Bildwiederholungsrate erfüllen, und der Hochleistungsrechner sorgt für eine schnellere Berechnung der angeforderten Extraktionsergebnisse. Da diese allerdings insbesondere bei sehr großen Datensätzen nicht in Echtzeit berechenbar sind, muss die für interaktive Systeme geforderte Systemreaktionszeit (2. Interaktionskriterium) durch die Darstellung temporärer Visualisierungsobjekte oder durch das Einblenden von statusanzeigenden Elementen der Benutzungsoberfläche (Beispiele: graphische Sanduhrmetaphern, Progressbalken, etc.) erfüllt werden.
Das erklärte Ziel dieser Arbeit ist es insbesondere sehr große, instationäre Datensätze in interaktiven virtuellen Umgebungen explorieren zu können. Daher rückt vor allem die sekundäre Reaktionszeit in den Vordergrund. Durch das Auslagern auf Mehrprozessorsystemen kann diese durch die Parallelisierung der Extraktionsberechnung deutlich reduziert werden. Allerdings stehen nicht immer und überall Hochleistungsrechner für das interaktive Postprocessing zur Verfügung. Insbesondere sind die Ingenieurswissenschaften auch nur bedingt bereit für die interaktive Analyse ihrer Strömungsdaten jedes Mal extra in ein Rechenzentrum zu gehen, wo sich neben dem HPC-Cluster häufig auch die Großraumvisualisierungssysteme für die interaktive Exploration befinden. Möchte man die Technik als tägliches Arbeitsmittel integrieren, müssen solche Systeme bei den jeweiligen Instituten installiert werden. Hierzu gehören neben kleineren VR-Systemen dann auch PC-Cluster, die das Budget weniger belasten und daher eher erschwinglich sind. Häufig sind solche Systeme schon für andere Anwendungsfälle beschafft worden, sodass diese bereits zur Verfügung stehen.
Während die heutigen Großrechner vor allem verclusterte *Shared-Memory*-Maschinen sind, arbeiten beispielsweise Linux-Cluster auf der Basis von *Distributed Memory*. Ein verteiltes

Postprocessing-System sollte daher beide Architekturen unterstützen und weitestgehend plattformunabhängig sein. Jedoch kann das Laufzeitverhalten paralleler Algorithmen auf diesen beiden Rechnerarchitekturen erheblich voneinander abweichen. Daher spielt in den folgenden Kapiteln der Vergleich von Aspekten wie Kommunikationslatenz, Balancierung und Skalierung auf den jeweiligen Cluster-Varianten eine wichtige Rolle.
Neben der Anbindung des Hauptspeichers haben auch noch andere Systemkomponenten bedeutenden Einfluss auf die erwähnten Bewertungsaspekte. Bei großen Datensätzen, die auf Festplatten abgespeichert vorliegen, sind dies vorrangig die verwendeten Dateisysteme. Lassen sich die benötigten Daten nicht schnell genug in den Hauptspeicher laden, können viele Parallelisierungsstrategien nicht den erwünschten Geschwindigkeitsvorteil erbringen. Ohne parallele Dateisysteme sind gewisse Ladestrategien erst gar nicht einsetzbar.
Aber nicht nur die installierte Hardware, sondern auch das Management der Datensätze kann einen erheblichen Einfluss auf den Erfolg einer implementierten Parallelisierungsstrategie nehmen. Hier sind vor allem durch das Postprocessing-System implementierte Caching- und Prefetching-Ansätze zu nennen, die sich für die jeweilige Anwendungsdomäne optimieren lassen.
Zwar lässt sich durch ein optimiertes Dateimanagement und durch Parallelisierungsstrategien eine erhebliche Beschleunigung der Berechnung (engl.: *Speed-up*) erreichen. In Abhängigkeit der verwendeten Daten und der Komplexität der angeforderten Merkmalsextraktion kann es jedoch weiterhin dazu kommen, dass die durch das 3. Interaktionskriterium definierte obere Laufzeitschranke überschritten wird. Daher müssen Mechanismen entwickelt werden, die die notwendige Gesamtlaufzeit einer Berechnung mit Zwischenergebnissen überbrückt. Solche Ansätze werden als *Data Streaming* bezeichnet. Hiermit kann erreicht werden, dass sich bereits kurz nach der Berechnungsanforderung erste Ergebnisse präsentieren lassen, obwohl der Extraktionsalgorithmus noch rechnet. Durch laufend neu eintreffende Teilergebnisse bleibt eine zeitlich enge Verknüpfung zwischen dem vom Anwender ausgelösten Befehl und den neu dargestellten graphischen Objekten. Somit kann eine höhere Akzeptanz auch für rechenintensivere Postprocessing-Anwendungen in interaktiven virtuellen Umgebungen erreicht werden. Die eigentliche Bewertungsgröße ist dabei die so genannte Streaming-Latenz, die angibt, wie viel Zeit vom Absetzen des Extraktionskommandos bis zur Visualisierung der ersten dargestellten Teildaten vergeht.
Um ein effizientes verteiltes Postprocessing-System für interaktive Visualisierungs-umgebungen zu implementieren, müssen daher Algorithmen untersucht und entwickelt werden, die auf die folgenden drei Ziele ausgerichtet sind:

- **Effiziente Parallelisierung**
- **Optimiertes Datenmanagement**
- **Kurze Latenzzeiten**

1.3 Bewertung paralleler Postprocessing-Algorithmen

Der zentrale Ansatz, eine interaktive Arbeit mit großen Datensätzen zu ermöglich, ist die Parallelisierung. Aber nicht immer ist es sinnvoll die Daten zu partitionieren und anschließend alle verfügbaren Prozessoren auf den eigentlich sequentiellen Algorithmus anzuwenden. Die Güte einer Parallelisierung lässt sich zum Teil durch die Balancierung und Skalierung bewerten. Letztendlich entscheidet aber nur der *Speed-up* über die Effizienz eines parallelen Algorithmus.
Die Berechnungsgeschwindigkeit ist nicht der einzige Bewertungsmaßstab für parallele Algorithmen. Insgesamt lassen sich drei Kriterien definieren:

- **Geschwindigkeit**
- **Speicherbedarf**
- **Durchführbarkeit** (engl.: *Usability*)

Dabei wird der Begriff der Geschwindigkeit im Kontext der interaktiven CFD-Analyse durch zwei unterschiedliche Laufzeitaspekte definiert. Die erste Geschwindigkeitsgröße bewertet die Gesamtlaufzeit, die ein Algorithmus benötigt, bis das endgültige Ergebnis visualisiert werden kann. Die zweite Größe beurteilt dagegen die im vorangegangenen Abschnitt definierte Latenzzeit von Streaming-Ansätzen.
Das zweite Kriterium untersucht den Speicherbedarf eines Algorithmus. Eine parallel durchgeführte Berechnung hat üblicherweise wegen der Verteilung der Datenlast eine geringere Speicheranforderung als der serielle Ansatz. Aber beim Zusammenfügen von Teildaten und für das Versenden von Ergebnissen kommt es sowohl auf der Visualisierungsseite als auch auf der Bearbeitungsseite zu zusätzlichem Hauptspeicherbedarf. Algorithmusbedingte Datenbehandlungsschritte können ebenfalls zu temporären Zusatzdaten und Duplikaten führen. Die Notwendigkeit und die entstehenden Systembelastungen sind von Fall zu Fall zu beurteilen.
Des Weiteren profitieren speziell optimierte Verfahren häufig von zusätzlich erzeugten Metadaten, die üblicherweise einmalig im Voraus berechnet und dann abgespeichert werden. Dieser Vorverarbeitungsschritt geht häufig mit einer Restrukturierung der Originaldaten einher, was diesmal zulasten des Festplattenspeichers geht. So greifen beispielsweise so genannte *Out-of-Core*-Strategien in aller Regel auf in Baumstrukturen gespeicherte Meta-Daten zurück, um gezielt nur wirklich benötigte Datensatzfragmente vom Dateisystem zu laden. Diese Strategien führen allerdings umgekehrt dazu, dass nun der Hauptspeicherbedarf deutlich reduziert werden kann.
Das dritte Bewertungskriterium untersucht Extraktionsalgorithmen daraufhin, ob sie sich überhaupt oder mit einem vertretbaren Aufwand in eine parallele Variante überführen lassen. Ein Beispiel dafür, dass ein Algorithmus prinzipiell nicht parallelisiert werden kann, ist die Berechnung einer Partikelbahn, da die gängigen Integrationsverfahren immer auch die vorangegangene Position für die Berechnung der aktuellen mit einbeziehen.
Zudem lässt sich hiermit die Eignung von Algorithmen beurteilen in eine streaming-fähige Variante überführt zu werden. Beispielsweise sind viele Multiresolution-Ansätze nur mit hohem Aufwand parallelisierbar und für das Streaming einsetzbar. Hier hat sich besonders die Entwicklung progressiver Extraktionsalgorithmen als problematisch erwiesen.
Alle Bewertungskriterien sind nochmals in Abbildung 1 graphisch zusammengefasst. Es ist dabei nahe liegend, dass einzelne Kriterien üblicherweise nicht isoliert betrachtet werden können. Besonders Geschwindigkeit und Speicherbedarf bedingen sich häufig einander.

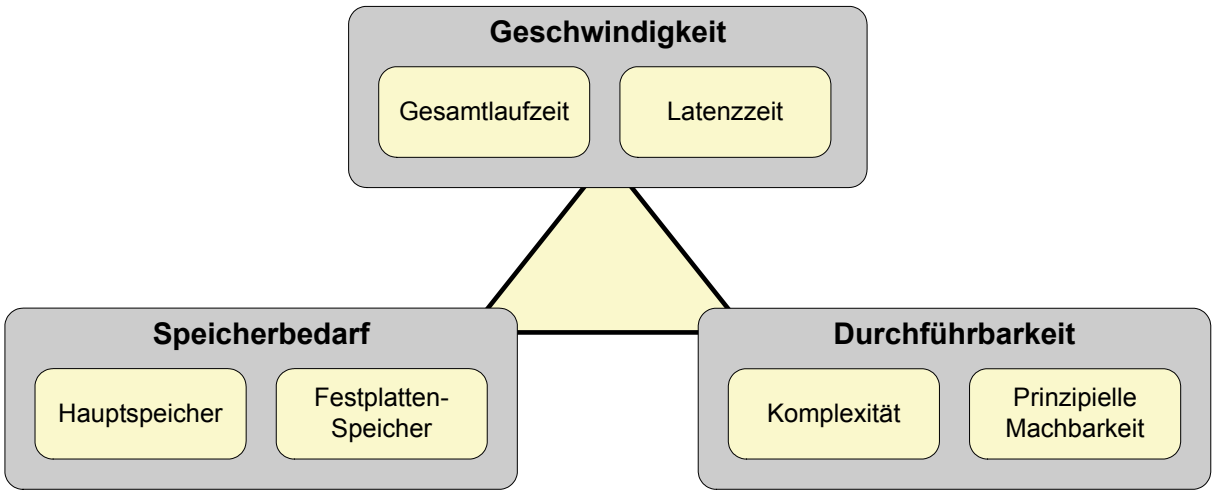

Abbildung 1: Kriterien zur allgemeinen Bewertung von parallelen Extraktionsalgorithmen

1.4 Gliederung

Im Zentrum der vorliegenden Arbeit steht das selbst entwickelte, parallele CFD-Postprocessing-Framework mit dem Namen Viracocha. Es ist als Extraktions-*Backend* in eine virtuelle Umgebung eingebetet, sodass sich hiermit vor allem die Fragestellung ergibt wie sich parallele Postprocessing-Algorithmen für die echtzeitfähige Visualisierung einsetzen lassen. Im folgenden Kapitel wird daher ausführlicher der Datenfluss in der Visualisierung-pipeline als zentraler Bestandteil des Postprocessings behandelt. Darauf aufsetzend werden mögliche prinzipielle Parallelisierungsansätze diskutiert und eine Klassifizierung verteilter Visualisierungssysteme vorgenommen. Eine Darstellung im Kontext dieser Arbeit wichtiger Systeme schließt dieses Kapitel ab.
Die meisten heute verfügbaren Visualisierungssysteme weisen eingeschränkte VR-Fähigkeiten auf. Zudem ist die Exploration sehr großer Datensätze üblicherweise nicht interaktiv möglich. Werden für die Berechnungsbeschleunigung Mehrprozessorcluster eingesetzt, sind lediglich einfache Parallelisierungsansätze integriert. Ein weit reichendes Datenmanagement, wie es in Kapitel 4 präsentiert wird, findet sich bei keinem der vorgestellten Postprocessing-Systeme. Daher wurde im Rahmen dieser Arbeit das Parallelisierungs-Framework Viracocha entwickelt, dessen objekt-orientiertes Design im Kapitel 3 beschrieben wird. Die Stärke dieses Designs ist die Kapselung funktionaler Basiskomponenten in disjunkte Software-Schichten, sodass eine leichte Wartbarkeit und Erweiterbarkeit garantiert werden kann. So erlauben einfache Schnittstellen die schnelle Integration neuer Extraktionsalgorithmen.
Nach der Beschreibung der Software-Architektur rücken vor allem die Verteilung und Verarbeitung großer Datensätze in den Mittelpunkt der Betrachtungen. Die unterschiedlichen Datenstrukturen, wie sie in der Strömungssimulation Anwendung finden, werden genauer analysiert. Im Mittelpunkt dieser Arbeit stehen so genannte Multi-Block-Datensätze. Dabei spielt die Topologie eine wichtige Rolle, sodass Nachbarschaftsbeziehungen von Daten ausgenutzt werden können, um optimierte Berechnungsvorschriften zu entwickeln und somit Laufzeiten einzusparen.
Viracocha hat einige generelle Parallelisierungsparadigmen implementiert, die von beliebigen Extraktionsverfahren eingesetzt werden können. Diese Konzepte werden genauer dargestellt und auftretende Laufzeitanteile identifiziert. Darauf aufbauend werden Postprocessing-Algorithmen untersucht, inwiefern sie sich mithilfe dieser Schnittstellen parallelisieren lassen. Die in der Einleitung angesprochenen Fragestellungen der Parallelisierbarkeit werden dabei untersucht. Vor allem die Bewertung der Skalierung und Balancierung steht im Mittelpunkt der Betrachtung.
Wie sich herausstellt, ist die beschriebene Problematik hauptsächlich dadurch bedingt, dass die den Ansatz des *Load-on-Demand* verfolgenden Strategien ständig Daten für das Postprocessing nachladen zu müssen. Als zentralen Lösungsansatz wird in Kapitel 4 das *Viracocha Data Management System* (VDMS) ausführlich dargestellt. Dieses Kapitel wird eingeleitet mit einem Abschnitt über bestehende *Middleware*-Ansätze, die die grundlegenden Strategien des optimierten Datenzugriffs und der Datenverwaltung für parallele Dateisysteme beschreiben. Das VDMS greift nun diese Ideen auf und setzt sie für den optimierten Einsatz von CFD-Datensätzen und Postprocessing-Algorithmen um. Die hier integrierten Mechanismen basieren primär auf Caching und Prefetching. Aber auch Wahrscheinlichkeits-graphen kommen zum Einsatz, was einen völlig neuen aber effizienten Ansatz in dieser Domäne darstellt. Der große Vorteil des VDMS ist, dass es sich transparent in das Gesamtdesign integriert. Unabhängig von der Algorithmusentwicklung können somit die hier implementierten Strategien selektiert und konfiguriert werden. Die ausführliche Beschreibung der entwickelten Mechanismen wird durch eine detaillierte Analyse abgerundet.

Eine ebenfalls bei den in Kapitel 2 vorgestellten VR-basierten Postprocessing-Systemen fehlende Komponente wird in Kapitel 5 vorgestellt und zielt auf eine verbesserte Interaktivität während der Exploration sehr großer CFD-Datensätze. Im Mittelpunkt steht der eingangs erwähnte Streaming-Ansatz, der sich als zentrale Verbindungskomponente zwischen parallelem Postprocessing und echtzeitfähiger Visualisierung herauskristallisiert hat. Einleitend werden die grundsätzlichen Ideen aus dem Bereich der Videoübertragung im Internet beschrieben. Die dort verwendeten Streaming- und Multiresolution-Techniken dienen wiederum Viracocha als Grundlage für das parallele CFD-Postprocessing. Mögliche Ansätze werden in einer Streaming-Matrix klassifiziert. Exemplarische Implementierungen werden vorgestellt und wiederum unter Verwendung verschiedener Datensätze auf Eignung und Schwächen hin untersucht.
Im letzten Kapitel werden die verschiedenen in dieser Arbeit betrachteten und implementierten Ansätze nochmals zusammengefasst und ausblickend bewertet.

2 Verteilte Postprocessing-Systeme

In diesem Kapitel wird zunächst ein Überblick über die üblicherweise notwendigen Verarbeitungsschritte für das CFD-Postprocessing gegeben. Diese werden in der so genannten Visualisierungspipeline zusammengefasst, die auch die maßgebliche Grundlage für die Bewertung von Parallelisierungsstrategien darstellt. Welche Möglichkeiten hier zur Verfügung stehen wird anschließend betrachtet. Um bestehende Visualisierungssysteme besser einordnen zu können, werden zudem ein Klassifizierungsschema und die darunter subsumierten wichtigsten Vertreter vorgestellt. Das Kapitel wird mit einer Untersuchung bestehender Visualisierungssysteme, die sich auf die Verarbeitung sehr großer Datenmengen spezialisiert haben, abgeschlossen.

2.1 Die Visualisierungspipeline

Parker bezeichnet in [PARK99] die Visualisierung als „Wissenschaft oder Kunst, Zahlen in Bilder zu fassen, die dem Verständnis der Zahlen dienen." Der Prozess vom Rohdatum bis zum visualisierten Strömungsobjekt wird üblicherweise mit dem Begriff *Postprocessing* umschrieben [HAIM94]. Um diesen Analysevorgang flexibel steuern zu können, werden die einzelnen Arbeitsschritte in einer so genannten Visualisierungspipeline (siehe Abbildung 2) zusammengefasst.

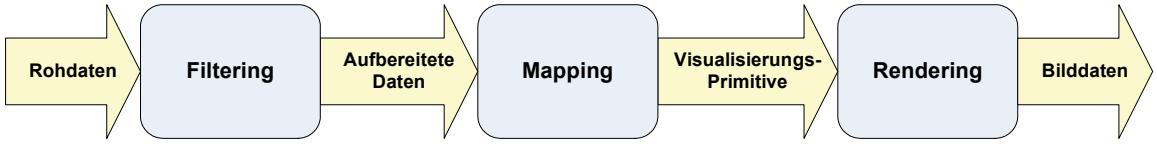

Abbildung 2: Verarbeitungsstufen der Visualisierungspipeline

Frühauf [FRUE97, S.29ff] unterteilt die Visualisierungspipeline in drei voneinander getrennte Verarbeitungseinheiten. Die Vorverarbeitung der Rohdaten wird dabei als *Filtering* bezeichnet. Der weitere Schritt der Abbildung der vorgefilterten Daten auf graphische Objekte erfolgt mit dem *Mapping*. Das *Rendering* erzeugt abschließend die pixelbasierten Bilddaten. Bei der explorativen Analyse werden die Parameter für das Filtern und *Mapping* entsprechend der durch die Betrachtung der dargestellten Ergebnisse gewonnenen Erkenntnisse ständig geändert. Nach einem Update der Pipeline können die sodann extrahierten Daten erneut untersucht werden.
Zentral für das CFD-Postprocessing ist das Filtern der Daten. Dabei stellt sich diese Pipelineebene nicht als eine einstufige, monolithische Transformationsverarbeitung dar. Vielmehr können mehrere Filter hintereinander geschaltet werden. Zudem kann ein Filter auch mehrere Eingangsdaten gleichzeitig zu einem Filterergebnis verarbeiten. Gängige Aufgaben, die während der Filterung gelöst werden, sind:

- Umwandlung von Datenstrukturen
- Zusammenfügen und Zerteilen von Daten
- *Resampling* von Rohdaten. Diese Filter werden u. a. für die Erzeugung von Multiresolution-Datenstrukturen (vgl. Abschnitt 5.3.2) verwendet.
- Konvertierung von Datenwerten. Dadurch kann sich die Dimension des Datums ändern.

Beim *Mapping* erfolgt dann das Erzeugen graphischer Primitive. In Abhängigkeit der Dimensionalität der aus der Filterung hervorgegangenen Extraktionsobjekte spricht man auch von einem *Mapping* skalarer bzw. vektorieller Daten.

Die Versorgung des Graphiksystems mit geeignet aufbereiteten Visualisierungsdaten stellt allerdings einen Falschenhals dar. So kann es schnell dazu führen, dass der gesamte Visualisierungsrechner vollständig mit der Verarbeitung komplexer geometrischer Szenarien und dem anschließenden Rendering ausgelastet ist. Des Weiteren werden für die Extraktion interessanter Strömungsphänomene aus den Rohdaten und das daran anschließende Erzeugen von geometrischen Objekten für die Visualisierung viel Arbeitsspeicher, Prozessorzeit und Berechnungszeit benötigt. Müssen diese Anforderungen vom Visualisierungssystem erfüllt, stehen die hierfür bereit gestellten Ressourcen nicht mehr für den Visualisierungsprozess zur Verfügung. Zwar wird das Problem der eingeschränkten Verfügbarkeit von wichtigen Ressourcen häufig durch nebenläufige Prozesse (*Multi-Threading*) gelindert. Dennoch reichen Prozessoranzahl und Speicherausbau oftmals nicht aus sehr große Simulationsdatenmengen zu verarbeiten.
Daher bietet es sich an, die Verarbeitung der CFD-Datensätze von Rechnern durchführen zu lassen, die bereits für die Simulation verwendet wurden. Diese Systeme sind optimiert für *High Performance Computing* (HPC), wofür viele parallel arbeitende Prozessoren und viel Hauptspeicher (mehrere Giga- bis Tera-Byte) zur Verfügung stehen. Dagegen kommen sie zum Teil auch völlig ohne Graphik-Hardware aus.
Es ist nun nahe liegend, die Visualisierungspipeline in zwei Teile aufzubrechen und den ersten Teil vom HPC-Rechner sowie den finalen Teil vom Visualisierungsrechner durchführen zu lassen (vgl. auch [OLBR01]). Welche Teile dabei von welcher Seite bearbeitet werden, kann durchaus variieren. Zwei Varianten der Aufteilung zeigt Abbildung 3.

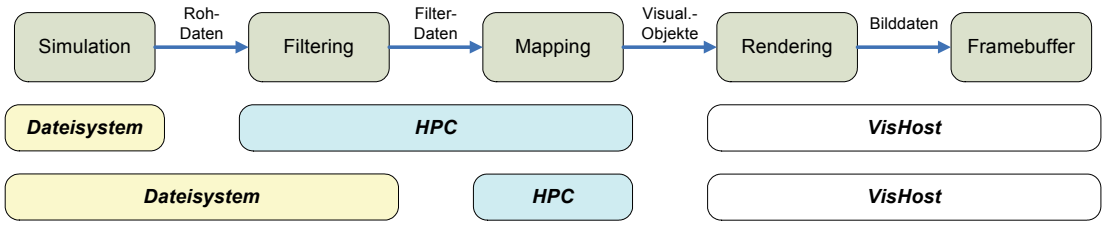

Abbildung 3: Aufteilungsvarianten zwischen Berechnung und Visualisierung

Die Entkopplung von Berechnung und Visualisierung kommt auf Echtzeit angewiesenen Systemen, wie virtuelle Umgebungen, zugute. Man kann mit bereits dargestellten Objekten interagieren, auch wenn im Hintergrund noch Berechnungen laufen. Zudem ist die Bildwiederholungsrate durch die alleinige Nutzung aller Ressourcen durch den Visualisierungsrechner ausreichend hoch, sofern nicht die zu rendernde Szene bereits so komplex ist, dass die Echtzeitfähigkeit darunter leidet. Aber durch Multiresolution-Techniken stehen auch für diese Problematik Lösungsansätze zur Verfügung.
Viele Faktoren bestimmen mit, ob die Bemühung der Parallelisierung der Visualisierungspipeline zum Erfolg führt. Dabei hängt die mögliche Effizienz zuallererst einmal vom entstehenden Kommunikationsaufwand ab [LAW99]. Um Ansätze für die Parallelisierung zu klassifizieren, wird die Behandlung einzelner Komponenten der Visualisierungspipeline zugrunde gelegt. Nach Ahrens et al. [AHRE00] lassen sich folgende drei Unterscheidungen finden:

Datenparallelität: Datenparallelität ist immer dann möglich, wenn sich der Ausgangsdatensatz oder Zwischenergebnisse in Teilmengen unterteilen lassen, die zumindest für eine gewisse Zeit unabhängig voneinander weiterverarbeitet werden können.

Prozessparallelität: Die Prozessparallelität bezeichnet dagegen die Möglichkeit, dass der Algorithmus und das System in der Lage sind, eine oder mehrere Aufgaben parallel zu bearbeiten. Üblicherweise sind die involvierten Berechnungseinheiten und die zu bearbeitenden Aufgaben vollständig unabhängig.

Pipeline-Parallelität: Der dritte Parallelisierungsansatz untersucht Wege, den Algorithmus so aufzubrechen, dass Berechnungsschritte, die sich hintereinander ausführen lassen, auf

mehrere Prozesse verteilt werden. Die Ergebnisse der einzelnen Schritte werden dann wie auf einem Fließband für die Weiterverarbeitung an den nachfolgenden Prozess durchgereicht.

Einen Einsatz von Pipeline-Parallelität für *Volume-Rendering* großer Datensätze beschreibt Kniss et al. in [KNIS01]. Dass sich jedoch nicht immer alle Parallelisierungsansätze für das CFD-Postprocessing umsetzen lassen, liegt auf der Hand. So beschreibt zum Beispiel Ahrens in [AHRE00] die Probleme, die auftreten, wenn Stromlinien auf verteilten Datenfragmenten berechnet werden sollen. Andere Extraktionsverfahren benötigen Nachbarschaftsbeziehungen, sodass zur Parallelisierung wieder verstärkt kommuniziert werden muss und somit das Problem der Kommunikationslast zunimmt.

2.2 Parallelisierungsschnittstellen

Um überhaupt parallele Berechnungsabläufe realisieren zu können, werden Programmierschnittstellen benötigt, die es ermöglichen vom Betriebssystem angebotene Multiprozessfunktionalität zu nutzen oder gar Prozesse über Rechnergrenzen hinweg zu verteilen. Die Parallelisierung ist dann jedoch Sache des Programmierers, obwohl auch immer mehr Compiler-Hersteller Optionen anbieten, seriellen Programm-Code automatisch zu parallelisieren. Funktioniert dies zum Teil bei großen numerischen Berechnungsprogrammen, versagen diese Ansätze in der Regel wegen der komplexen, objekt-orientierten Software-Architektur beim CFD-Postprocessing.
Die im Rahmen dieser Arbeit untersuchten und implementierten Schnittstellen werden im Folgenden beschrieben:

Threads: Grundsätzlich bieten alle heutigen Betriebssysteme Funktionalitäten an, einen Prozess in mehrere Ablaufstränge, so genannte *Threads*, zu spalten. Seit einigen Jahren existiert ein allgemeiner POSIX-Standard, der kurz als *pthread* [NICH96] bezeichnet wird.
Die von einem Prozess abgezweigten Programmfäden laufen alle in einer gemeinsamen Speicherumgebung. Durch eine unvorhersehbare Reihenfolge von Schreib- und Lese-Zugriffen kann es zu nicht-deterministischen Speicherinhalten und Programmabläufen kommen. Um dieses als *Race Condition* bezeichnete Verhalten zu vermeiden, müssen kritische Sektionen in der Programmabfolge definiert und durch Ausschlussmechanismen, wie mit *Mutex*- und *Semaphor*-Objekten [DIJK65], die im Betriebssystem verankert sind, eingeklammert werden.
Dieser Ansatz ist auf einen einzigen Rechnerknoten beschränkt, wodurch die Beschleunigung der Berechnung durch die Anzahl lokal installierter Prozessoren an einem gemeinsamen Speicher limitiert ist.

Message Passing: Alternativ zu *Threads* existieren Bibliotheken, die Prozess-Parallelisierung durch das Versenden von Nachrichten untereinander realisieren. Hier hat sich als Quasi-Standard das *Message Passing Interface* (MPI) [GROP99, SNIR96, PACH96] durchgesetzt.
MPI spezifiziert vor allem eine Schnittstelle zur Kommunikation zwischen verschiedenen Prozessen. Dabei sind die parallel laufenden Prozesse nicht auf *Shared-Memory*-Architekturen beschränkt, sondern können auch in *Distributed-Memory*-Systemen eingesetzt werden. Aber auch heterogene Einzelrechner lassen sich mit MPI zu einem Parallelisierungs-Cluster verbinden.

OpenMP: Für die Parallelisierung von Algorithmen auf *Shared-Memory*-Maschinen steht eine leistungsfähige Alternative mit dem Namen *OpenMP*[1] [DAGU98, CHAN00] zur Verfügung. Diese vor allem für die Programmiersprachen Fortran, C und C++ existierende Schnittstelle ist eine Pragma-Sprache und lässt sich in bestehenden

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

sequentiellen Code einbringen. Im Gegensatz zur grobgranularen Parallelisierung durch Nachrichtenversand werden hier vom Compiler vorrangig Schleifen in mehrere *thread*-parallele Prozesse zerlegt. Dabei ist eine automatische Lastbalancierung möglich, die durch die Wahl des entsprechenden *Scheduler*-Parameters bestimmt wird:

- ***Static***: Die angegebene Anzahl von *Threads* teilen sich die zu bearbeitenden Daten gleichmäßig auf. Ein optionaler Parameter, als *Chunk Size* bezeichnet, bestimmt die Anzahl der Daten, die jeweils auf einmal zugeordnet wird. Damit erreicht man eine Auffächerung der Daten auf die *Threads*, wobei zur Compile-Zeit schon klar ist, welcher *Thread* welche Daten verarbeitet.
- ***Dynamic***: Die *Threads* verarbeiten Blöcke von Daten, deren Größe wiederum durch die *Chunk Size* angegeben wird. Hat ein *Thread* die Bearbeitung eines Datenblocks abgeschlossen, wird der nächste Datenblock verarbeitet. Die Anzahl der von einem *Thread* zu bearbeitenden Blöcke ist im vornherein nicht bekannt, da diese zur Lastbalancierung dynamisch auf die einzelnen *Threads* verteilt werden.
- ***Guided***: In diesem Fall ist die jeweils zu bearbeitende Blockgröße ebenfalls dynamisch, während die Anzahl der *Threads* vorgegeben ist. Üblicherweise werden zuerst größere Blöcke zugewiesen, die im Laufe der Verarbeitung immer kleiner werden. Dadurch soll eine möglichst optimale Lastverteilung erreicht werden.

Durch die als *Nested OpenMP* bezeichnete Option in verschachtelten Programmschleifen dynamisch zusätzliche Prozesse für die Berechnung zu verwenden, ist eine noch filigranere Balancierung der Last erreichbar.

CORBA: Eine ganz andere Art des Einsatzes von mehreren Rechnern für die Verarbeitung eines Berechnungsproblems ist die Verteilung von Software-Komponenten. Dabei wird in der Regel nicht das Ziel verfolgt Verarbeitungsschritte parallel durchführen zu lassen, sondern durch Verteilung der Schritte in ihrer Gänze die zur Verfügung stehenden Ressourcen optimaler ausnutzen zu können. Der bekannteste Vertreter dieser Ansätze ist CORBA (*Common Object Request Broker Architecture*), das ein objekt-orientiertes Programmiermodell verfolgt. [OMG02, SIEG00, TANE02]

Bis auf CORBA wurden alle Parallelisierungsschnittstellen im Framework Viracocha, das in Kapitel 3 ausführlich dargestellt wird, verwendet. Auf den Einsatz von CORBA wurde verzichtet, da es ein ganz anderes, konträres Programmierparadigma verfolgt. Dennoch wurde in [GERN04b] die Möglichkeit untersucht CORBA für die Parallelisierung von Visualisierungspipelines auszunutzen. Dabei konnte gezeigt werden, dass auch CORBA für die Parallelisierung von wissenschaftlichen Visualisierungsanwendungen einsetzbar ist, wenn das entsprechende Framework konsequent auf das von CORBA verfolgte komponentenbasierte Programmiermodell aufsetzt.

2.3 Visualisierungssysteme

Für die Visualisierung wissenschaftlicher Daten existiert eine ganze Reihe von Bibliotheken und Systemen. Einen aktuellen Überblick bietet [BROD04]. Nach Thomas Frühauf [FRUE97] lassen sie sich in vier Kategorien unterteilen:

Graphik-Bibliotheken: Diese stellen rudimentäre 3D-Funktionalitäten zur Verfügung. Die bekanntesten Bibliotheken sind OpenGL [NEID93, ROGE92] und Direct-3D [GRAY03]. Zusammen mit Toolkits für die Gestaltung von Benutzungsoberflächen, wie beispielsweise Qt[2] oder X/Motif [QUER91], lassen sich dann eigene Visualisierungssysteme entwickeln.

[2] http://www.trolltech.com

Visualisierungsbibliotheken: Neben allgemeinen Graphikfunktionen werden weitere Funktionalitäten angeboten, um spezielle Visualisierungsanforderungen realisieren zu können. Das *Visualization Toolkit* (VTK) ist der am weitesten verbreitete Vertreter solcher Bibliotheken.

Monolithische Visualisierungssysteme: So genannte *Turnkey Visualization Systems* sind fertige Anwendungen, die ohne zusätzlichen Programmieraufwand sofort benutzt werden können. Sie weisen eine applikationsspezifische Benutzungsoberfläche auf, und der Befehlsumfang ist fest vorgegeben. Der Vorteil liegt in der sofortigen Benutzbarkeit, der einfachen Datenhandhabung und der häufig hohen Render-Leistung. Die bekanntesten Systeme dieser Kategorie sind *Fieldview* von Intelligent Light[3], FAST [BANC90] vom NASA Ames Research Center und *TecPlot*[4].

Datenflussorientierte Visualisierungssysteme: Im Gegensatz zu den monolithischen Systemen lassen sich hier Visualisierungsmodule interaktiv zusammenstellen. Dadurch wird zwar eine höhere Flexibilität erreicht, umgekehrt allerdings die Bedienung erschwert, was in der Regel zu Lasten der Interaktionsfähigkeit geht. Die folgenden Systeme sind dieser Kategorie zuzuordnen: *SCIRun* [PARK99] von der University of Utah, *Vis5D* [HIBB90] von der University of Wisconsin-Madison, AVS [UPSO89, KROG93] und AVS/Express von Advanced Visual Systems[5], *Amira* [STAL04] von TGS Inc.[6], *VisiQuest* von AccuSoft[7] (ursprünglich als *Khoros* von der University of New Mexico entwickelt), EnSight [GRIM89] von CIE, *OpenDX*[8] (ehemals *Visualization Data Explorer* von IBM).

2.3.1 Das Visualization Toolkit

Das *Visualization Toolkit* (VTK) wurde 1993 einem Buch von William Schroeder, Ken Martin und Bill Lorensen über 3D-Computergraphik beigelegt [SCHR00a]. Seitdem erfreut es sich einer zunehmenden Beliebtheit. Seine Verbreitung verdankt es vor allem der Tatsache, dass es als *Open-Source*-Projekt angelegt ist. Es ist eine in C++ geschriebene objekt-orientierte Bibliothek, die unter den unterschiedlichsten Betriebssystemen eingesetzt werden kann.

Für die Analyse von Strömungsdaten existieren neben der Unterstützung der üblichen Datengitterstrukturen und Zelltypen, für die es eine ganze Reihe von Manipulationsklassen gibt, auch einige Klassen für die Extraktion von stationären Strömungsmerkmalen, wie beispielsweise Schnittflächen, Isoflächen und Stromlinien.

Der Kernmechanismus von VTK basiert ebenfalls auf einer Visualisierungspipeline. Besonders viel Aufmerksamkeit wurde auf deren Update-Mechanismus gelegt, der nach Modifikationen an der Pipeline immer nur denjenigen Teil erneut durchrechnet, der von den Änderungen betroffen ist. Dieser Ansatz wird als implizites Update-Verfahren [SCHR98, S. 92ff] bezeichnet. Bei VTK muss der Update-Prozess jeweils manuell angestoßen werden, sodass vorher mehrere Änderungen an der Pipeline gebündelt vorgenommen werden können. Die Update-Anforderung wird dabei auch immer nur in Richtung des Quellknotens propagiert. Der Prozessablauf ist beispielhaft in Abbildung 4 dargestellt.

Der von VTK implementierte Pipeline-Mechanismus kommt vor allem interaktiven Systemen zugute, die häufig immer wieder die gleichen Parameter innerhalb der Pipeline für eine explorative Analyse anpassen müssen. Auch Viracocha verwendet die Visualisierungspipeline von VTK. Da mit diesem Postprocessing-*Backend* jedoch lediglich Strömungsmerkmale

[3] http://www.ilight.com
[4] http://www.techplot.com
[5] http://www.avs.com
[6] http://www.tgs.com; http://amira.zib.de
[7] http://www.accusoft.com
[8] http://www.opendx.org

extrahiert aber nicht visualisiert werden sollen, kommt hier die um die entsprechenden Stufen verkürzte VTK-Pipeline zum Einsatz.

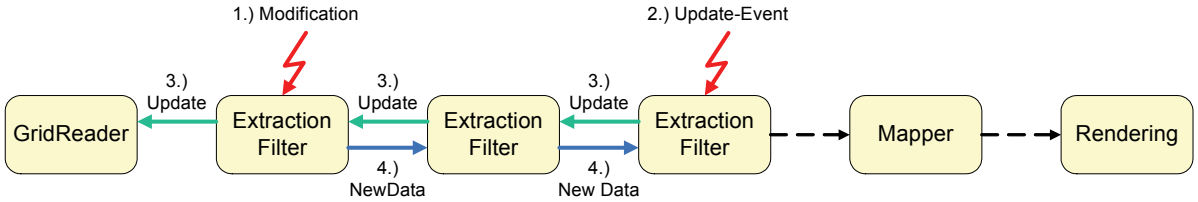

Abbildung 4: Impliziter Update-Mechanismus einer Visualisierungspipeline

VTK lässt sich mit unterschiedlichen Optionen übersetzen. Bereits in der Standardversion besteht für viele implementierte Filterobjekte die Möglichkeit diese parallel mithilfe von *Multi-Threading* auszuführen. Ein Parallelisierungsschritt beschränkt sich hier immer nur auf einen einzigen Knoten der Pipeline. Stehen auf der Graphik-Workstation jedoch genügend Prozessoren zur Verfügung, kann hierdurch bereits eine erhebliche Beschleunigung der Extraktion erreicht werden.
Um die Einschränkungen der *Thread*-Parallelisierung zu überwinden, wurde VTK durch die Möglichkeit der Pipeline-Parallelität ergänzt [AHRE00]. Um einen möglichst geringen Eingriff in die implementierte Pipeline-Struktur vornehmen zu müssen, wurden *Input*- und *Output-Ports* definiert, die man nun nach Bedarf zwischen den Knotenverbindungen einfügen kann. An diesen Stellen wird dann die Pipeline aufgebrochen und verschiedenen Prozessen zugeordnet. Müssen Daten über Prozessgrenzen hinweg verteilt werden, ist es notwendig zuerst einen seriellen Datenstrom aus den anliegenden komplexen Datenstrukturen zu erzeugen (Serialisierung). Auf der anderen Seite der Pipeline wird aus dem Datenstrom wieder das ursprüngliche Datenobjekt generiert (Deserialisierung) und die Verarbeitung fortgesetzt. Abbildung 5 zeigt den Mechanismus andeutungsweise anhand eines Beispiels.

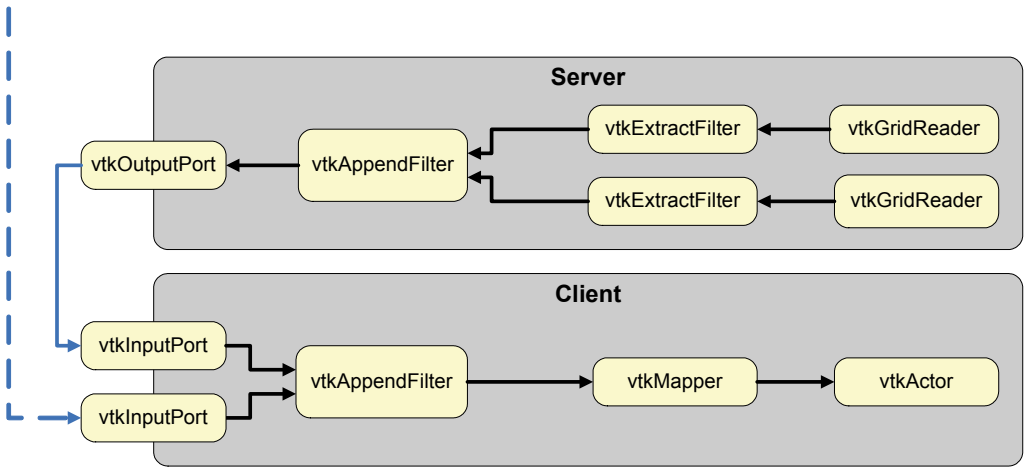

Abbildung 5: Beispiel einer Verteilung der VTK-Pipeline über *Ports*

2.3.2 Systeme für große wissenschaftliche Datensätze

Sollen große Datensätze verarbeitet werden, führt dies zu einer hohen Speicheranforderung oder zu einer intensiven CPU-Auslastung [PARK99], sodass wegen langer Verarbeitungszeiten üblicherweise der Einsatz in Echtzeitsystemen nicht mehr möglich ist. Als Lösung wurden daher in den letzten 20 Jahren zunehmend verteilte Postprocessing-Systeme entwickelt, die in diesem Abschnitt genauer betrachtet werden.
Eine große Verbreitung erlebte das speziell für die Analyse von CFD-Datensätzen bestimmte *Plot3D*. Es wurde bereits 1985 von Buning et al. in [BUNI85] vorgestellt und sollte die Darstellung auch großer dreidimensionaler Datenmengen erlauben. Bis heute wird das mit diesem Programm eingeführte Datenformat von vielen Anwendungen als Eingabeformat

unterstützt. Ein Beispiel hierfür ist das von Bancroft et al. [BANC90] entwickelte FAST, das zur Berechnungsbeschleunigung die Verwendung von Multiprozessorsystemen vorsieht. Beide bieten bereits die Möglichkeit an auch instationäre Daten zu evaluieren. Dabei beschränkt sich die Analyse in der Regel darauf, dass sich einzelne Zeitschritte anwählen lassen um sodann stationäre Postprocessing-Berechnungen durchführen zu können. Neben diesen Systemen gibt es noch weitere frühe Systeme (Beispiele: COMADI [VOLL91], HIGHEND [PAGE93], *Visual3* [HAIM91]). Die ersten Postprocessing-Programme, die auch instationäre Extraktionsverfahren aufwiesen, wurden von Smith et al. [SMIT89] mit den Systemen PLOT4D und *Streaker* entworfen, gefolgt von *pV3* von Haimes [HAIM94] und UFAT von Lane [LANE94]. Einen Vergleich dieser ersten Systeme findet man in [LANE97].

2.3.2.1 Active Data Repository

Ein weit reichendes Applikations-Framework, das vorrangig aus der medizinischen Bildverarbeitung kommt, aber auch für andere Bereiche der wissenschaftlichen Visualisierung einsetzbar ist, wurde 1998 als T2 in [CHAN99] vorgestellt und später in [KURC01] als *Active Data Repository* (ADR) ausführlich beschrieben. Das Hauptaugenmerk liegt auf der Verwendung von *Distributed-Memory*-Systemen wie PC-Cluster, da hier kostengünstigere und dennoch leistungsfähige Postprocessing-Systeme möglich sind. Entwickelt wurde es als eine objektorientierte C++-Bibliothek, die für die Kommunikation untereinander auf MPI aufbaut. Die ermittelten Ergebnisse lassen sich dann mit VTK visualisieren.

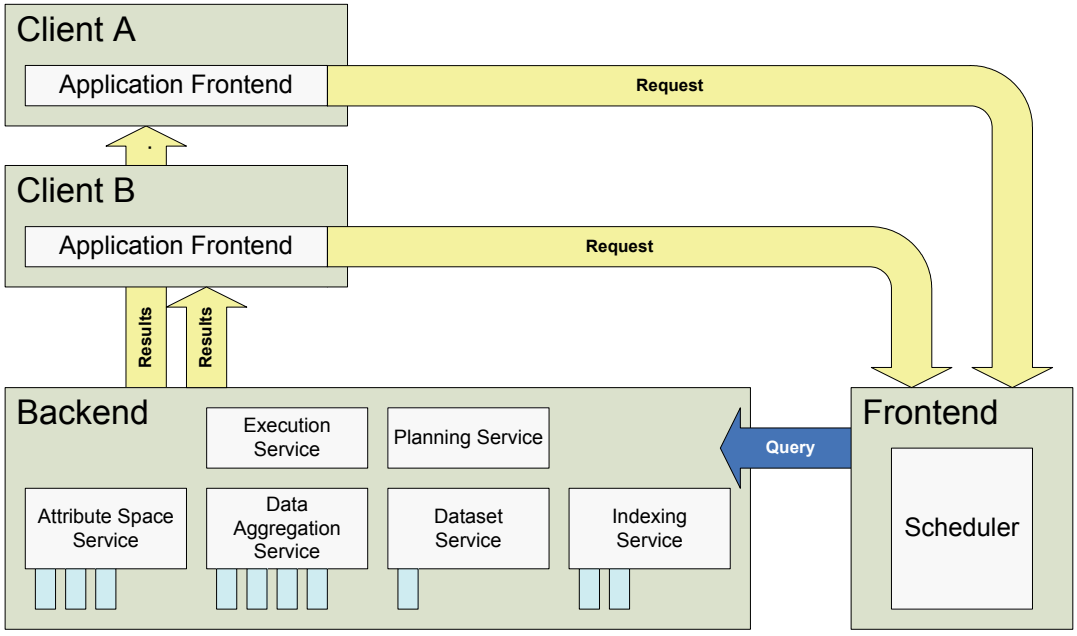

Abbildung 6: Zwei Anwendungen (*Clients*) sind mit der ADR-Laufzeitumgebung verbunden

Dabei geht ADR deutlich über die einfachen parallelen Ansätze der anderen Systeme hinaus. So ist das Laufzeitsystem von ADR, in Abbildung 6 skizzenhaft dargestellt, in drei Teile aufgegliedert. Zum Einbinden in eigene Applikationen, den *Clients*, existiert für den Anwendungsentwickler ein Interface, das als *Application Frontend* bezeichnet wird. Das eigentliche Berechnungs-Framework teilt sich in zwei weitere Teile, in ein *Frontend* und ein *Backend*, auf. Anforderungen nach Berechnungsdaten werden in ADR-verständliche Anfragen verpackt und per TCP/IP an das auf dem HPC-Cluster befindliche Berechnungs-*Frontend* verschickt. Dieses nimmt die Anfrage an und übergibt sie zunächst einem *Scheduler*, der alle Anfragen verwaltet. Sobald das *Backend* weitere Berechnungs-Jobs annehmen kann, entscheidet der *Scheduler*, welche anliegende Anfrage als nächstes bearbeitet werden soll. Die wichtigste Komponente von ADR ist jedoch das *Backend*. Hier werden die

Datenzugriffe organisiert und die Berechnungen durchgeführt. Eine Planungseinheit hilft beim Verwalten von Ressourcen, die den Berechnungs-Jobs zugeordnet werden können.
Neben dieser fest verankerten *Backend*-Komponente existieren Schnittstellen, über die erst die anwendungsspezifischen Verwaltungs- und Berechnungsvorschriften, als *Services* bezeichnet, integriert werden müssen. Die vier Schnittstellen zum *Backend* definieren sich wie folgt:

Data Aggregation Service: Hierunter werden Funktionen verstanden, die konkret bestimmen, welche Ergebnisdaten berechnet werden sollen und welcher Algorithmus hierfür zur Anwendung kommt. Die an dieser Stelle implementierten *Services* können sodann vom *Frontend* direkt ausgewählt werden.

Dataset Service: Der Zugriff auf die Daten, die von den Berechnungsalgorithmen benötigt werden, lässt sich hier definieren.

Indexing Service: Datensätze, die im ADR-*Backend* gespeichert werden, müssen über einen Index beschrieben werden. Dies geschieht durch die Implementierung spezieller Dienste durch den Anwender, wobei sich der Index üblicherweise durch einen Tupel aus Dateiname, Offset und Größe definiert.

Attribute Space Service: Hierüber werden multidimensionale Attribute registriert und verwaltet. Damit ein gewisses Attribut bei der Berechnung der Ergebnisdaten verwendet werden kann, muss der Benutzer zuerst Abbildungsfunktionen für den *Attribute Space Service* implementieren.

Eine wichtige Annahme beim Zugriff auf Daten ist, dass beim Postprocessing von großen Datensätzen vorwiegend auf einer Teilmenge gearbeitet wird, die einen räumlich zusammenhängenden Bereichsausschnitt darstellt. Anfragen nach Daten sind dann immer auch Bereichsanfragen, so genannte *Range Queries*. Daher unterteilt ADR die Daten in logische Einheiten, die als *Chunks* bezeichnet werden. Dabei sollten die in einem *Chunk* versammelten Daten möglichst nahe beieinander liegen. Danach werden sie auf die lokalen Festplatten mittels eines Verfahrens, das auf Hilbert-Kurven [MOON01] basiert, verteilt. Dadurch soll gewährleistet werden, dass der Zugriff auf hintereinander liegende Datenstücke möglichst effizient erfolgt.
Die Leistungsfähigkeit des Datenmanagements und einiger konkreter Berechnungsstrategien beim Verarbeiten verschiedener, großer multidimensionaler Datensätze konnte in [KURC99] aufgezeigt werden.

2.3.3 VR-basierte Systeme für das verteilte CFD-Postprocessing

Aktuelle Visualisierungsprogramme weisen zumindest die Möglichkeit auf auch stereoskopische Abbildungen zu generieren. Doch nur die wenigsten sind tatsächlich in virtuellen Umgebungen einsetzbar. Hierzu fehlt die Unterstützung von Großprojektionssystemen und Tracking-Geräten. Unterdessen existieren dennoch einige echtzeitfähige Systeme, die als vorrangiges Ziel das interaktive Postprocessing in virtuellen Umgebungen verfolgen. Die wichtigsten werden nun kurz vorgestellt.
Eines der frühen auf VR basierenden Systeme für die Strömungsvisualisierung wurde 1991 von Bryson und Levit in [BRYS91] beschrieben. Dieses als *Virtual Windtunnel* (VWT) bezeichnete System wurde am NASA Research Center entwickelt. Das um ein vektorisiertes Postprocessing-*Backend* erweiterte System *Distributed Virtual Windtunnel* [BRYS92] lagert die Merkmalsextraktion auf einen Hochleistungsrechner aus. Dabei ist das System fest an die Anwendung und an spezielle Hardware gebunden und damit wenig flexibel.
Primär auf kollaboratives Arbeiten ausgerichtet ist COVISE (*COllaborative VIsualization and Simulation Environment*) [RANT98a, WIER01]. Dieses datenflussorientierte Postprocessing-System erlaubt es, dass weitere Anwender an einer Arbeitssitzung eines Hauptbenutzers partizipieren können. Ursprünglich als Desktop-Analysetool konzipiert, wurde die VR-Tauglichkeit erst später durch das optionale COVER-Modul (Abkürzung für: *COVISE Virtual*

Environment) integriert [RANT98b]. Hierüber ist es möglich mithilfe von Tracking-Systemen und der Ansteuerung von Multi-Screen-Displays Arbeitssitzungen auch in virtuellen Umgebungen auszuführen.
COVISE gestaltet sich durch die Aufteilung auf Module sehr flexibel. So existieren u. a. spezielle Komponenten für die Ein-/Ausgabe, die Datenfilterung und das Rendering. Wegen der kollaborativen Ausrichtung von COVISE wurde von vornherein auf einen interaktiven Arbeitsablauf geachtet. Für diesen Zweck lassen sich beispielsweise gemeinsam benötigte Daten zuvor duplizieren und verteilen. Die einzelnen Module lassen sich dann lokal auf den einzelnen Workstations ausführen. Somit beschränkt sich die Kommunikationslast auf minimale Information, wie beispielsweise *Viewpoints*, die während einer Arbeitssitzung untereinander verschickt werden müssen. Dieses Konzept ermöglicht den Einsatz von COVISE auch in Arbeitsumgebungen, die lediglich über geringe Bandbreiten verfügen. Um auch Mehrprozessorsysteme für eine Beschleunigung der Berechnungsschritte ausnutzen zu können, lässt sich wie bei vielen datenflussorientierten Programmen wiederum Pipeline-Parallelität ausnutzen. Dabei werden die einzelnen Berechnungsmodule auf verschiedene Berechnungsknoten verteilt. Die Datenkommunikation zwischen den einzelnen Pipeline-Modulen findet dann entweder über Netzwerkprotokolle wie TCP/IP oder über *Shared-Memory*-Datenaustausch statt [RESC99].

3 Paralleles Postprocessing-Framework

In diesem Kapitel werden zunächst das zugrunde liegende VR-Toolkit sowie das Basisrahmenwerk für das VR-basierte, verteilte CFD-Postprocessing vorgestellt. Anschließend wird ausführlicher auf das Design des parallelen Postprocessing-Frameworks eingegangen bevor abschließend allgemeine Fragestellungen der Datenparallelisierung und parallele Algorithmen für die Strömungsanalyse betrachtet werden.

3.1 Das VR-Toolkit ViSTA

Seit 1998 wird am Rechen- und Kommunikationszentrum der Rheinisch-Westfälischen Technischen Hochschule (RWTH) in Aachen eine VR-Software mit dem Namen ViSTA (*Virtual Reality for Scientific Technical Applications*) entwickelt [REIM00]. Das primäre Ziel ist die einfache Integration technisch-wissenschaftlicher Anwendungen in virtuelle Umgebungen. Durch die Kapselung VR-spezifischer Aspekte muss sich der Applikationsentwickler kein zusätzliches Wissen über VR-Technologie und deren Ansteuerung aneignen, sodass er sich vor allem auf die Entwicklung von Algorithmen für den eigentlichen Anwendungsfall konzentrieren kann.
In ViSTA ist der Szenegraph das zentrale Organisationsobjekt. Für seine Verwaltung wurde eine eigene Managementkomponente mit dem Namen *Graphics Manager* integriert. Wegen der Komplexität übernimmt der *Graphics Manager* jedoch die Darstellung des Szenegraphen nicht selbst, sondern greift hierfür auf Renderer von Drittanbietern zurück.
Auch die anderen Aufgaben einer VR-Software wurden in voneinander unabhängigen Managementgruppen zusammengefasst. So besteht die Kernbibliothek (*Vista Kernel*) neben dem *Graphics Manager* aus einer Gruppe zur Ansteuerung von Eingabegeräten (*Interaction Manager*), aus einer Gruppe für die Konfiguration von mehrflächigen Projektionssystemen einschließlich der *View-Point*-Verwaltung (*Display Manager*), sowie aus einer Gruppe zur Verwaltung selektierter VR-Objekte (*Pick Manager*). Zustandsänderungen innerhalb der von den Managern verwalteten Objekte werden mithilfe des *Event Managers* im gesamten System bekannt gegeben. Die einzelnen ViSTA-Komponenten des Kernel-Moduls sind in Abbildung 7 dargestellt.

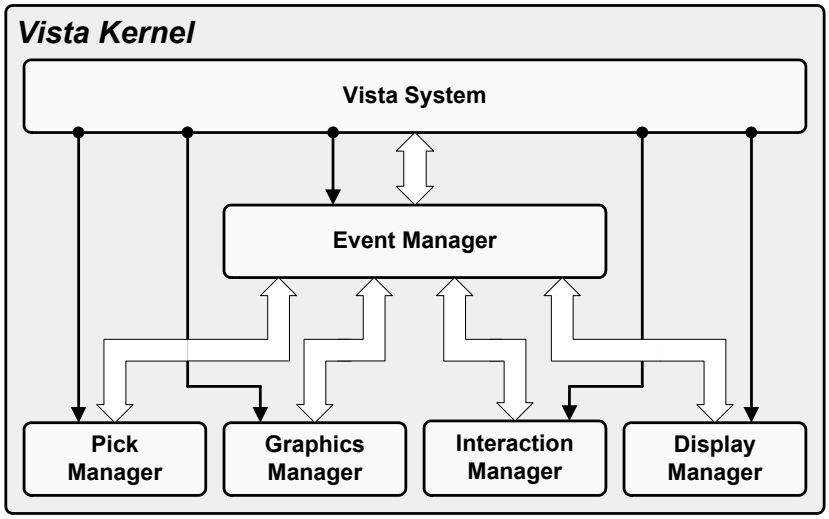

Abbildung 7: Daten- und Kontrollfluss zwischen internen Komponenten des Kernel-Moduls von ViSTA

Sollen nicht nur einfach Modelle eingeladen und visualisiert werden, muss die jeweilige Applikation einen eigenen *Application Event Manager* beim *Vista System* registrieren. Dieser muss vom *Event Manager* abgeleitet sein. Danach kann auf den vom *Vista System* kontrollierten Programmablauf, der wiederum durch *Events* gesteuert wird, sowie auf die durch die beschriebenen Manager getriggerten Systemänderungen gezielt reagiert werden.
ViSTA ist weitestgehend plattformunabhängig. Betriebssystemspezifische Eigenarten wurden gekapselt. Durch das Bereitstellen logischer Geräte muss sich der Applikationsentwickler in der Regel auch nicht um hardware-spezifische Besonderheiten kümmern. Die Spezifikation des zu verwendenden VR-Systems wird durch Konfigurationsdateien vorgenommen, sodass die Applikation von der zum Einsatz kommenden VR-Hardware unabhängig ist. Ohne Änderungen an der Applikation lässt sich der VR-Arbeitsplatz somit beliebig von der üblichen Desktop-Umgebung mit Monitor und 2D-Maus bis hin zu CAVE-ähnlichen Großprojektionssystemen [CRUZ92, CRUZ93] mit integriertem Tracking skalieren.
Um die Flexibilität weiter zu erhöhen, wurden funktionale Einheiten in unabhängige Module, die in der Regel als selbständige Bibliotheken zur Verfügung stehen, zusammengefasst. Die wichtigsten, auf die auch der Kernel von ViSTA zugreift, sind Module für immer wiederkehrende mathematische Grundoperationen, für einige Helferklassen (wie z. B. zum Auslesen der Konfigurationsdateien und *Timer*-Klassen), sowie für Klassen, die sich mit der Kommunikation zwischen Prozessen beschäftigen. Das zuletzt erwähnte IPC-Modul (*Inter-Process Communication*) wird gerade in verteilten Anwendungen und für nebenläufige Programmabläufe benötigt. So befinden sich hier Klassen für Hardware-Schnittstellen, für die Kommunikation über TCP/IP und UDP, für *Multi-Threading* (inklusive Kontrollstrukturen für kritische Sektionen) und für *Shared-Memory*-Programmierung.
Neben diesen drei Basis-Modulen existieren weitere optionale Module z. B. für 3D-Menüs, für Multimedia-Anwendungen, für die Integration einer Skriptsteuerung, für physikalisch basierte Modellierung [STEF01] und für 3D-Akustik [ASSE04]. Wie bereits für den *Graphics Manager* beschrieben, werden für das jeweilige Modul nicht immer sämtliche Funktionalitäten nachimplementiert sondern auf weitere Toolkits zurückgegriffen. In diesen Fällen entstehen üblicherweise zusätzliche Abhängigkeiten zu weiteren Software-Bibliotheken. Das prinzipielle Moduldesign ist in Abbildung 8 dargestellt.

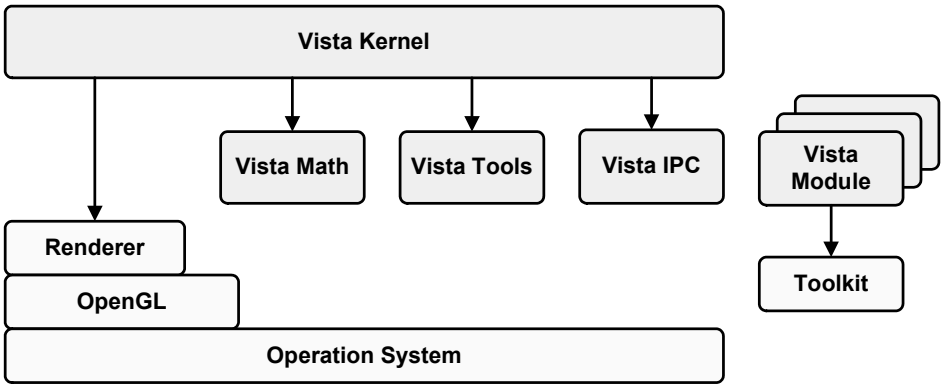

Abbildung 8: Das modulare Konzept von ViSTA

3.2 CFD-Postprocessing mit ViSTA FlowLib

Durch die Verwaltung von geometrischen Objekten mithilfe des in ViSTA verankerten Szenegraphen lassen sich auch wissenschaftliche Visualisierungen realisieren. Die weit reichende Flexibilität, die sich mit dem Einsatz von Visualisierungspipelines erzielen lässt, ist dadurch jedoch nicht ohne weiteres umsetzbar. Um diese Pipeline-Funktionalitäten nicht jeweils erneut durch eine Applikation nachimplementieren zu müssen, wurde ein ViSTA-Modul speziell für die wissenschaftliche Visualisierung mit dem Namen ViSTA FlowLib

entwickelt [SCHI03]. Die Bezeichnung ergab sich aus der Tatsache, dass der Fokus vor allem auf das Postprocessing großer, instationärer Strömungsdatensätze gesetzt wurde. ViSTA FlowLib ist in drei disjunkte Verwaltungseinheiten unterteilt:

- **Mensch-Maschine-Schnittstelle** (HCI): verwaltet auf die Strömungsexploration zugeschnittene Interaktionsmethaphern.
- **Visualisierungseinheit:** Sorgt für das korrekte Zeit-Mapping instationärer Visualisierungsobjekte.
- **Extraktions-Manager**: Übernimmt die Koordination von Extraktionsanfragen an das Parallelisierungs-*Backend* Viracocha.

Die wesentlichen Komponenten und die wichtigsten Bibliotheksabhängigkeiten des gesamten Postprocessing-Frameworks sind in Abbildung 9 dargelegt.

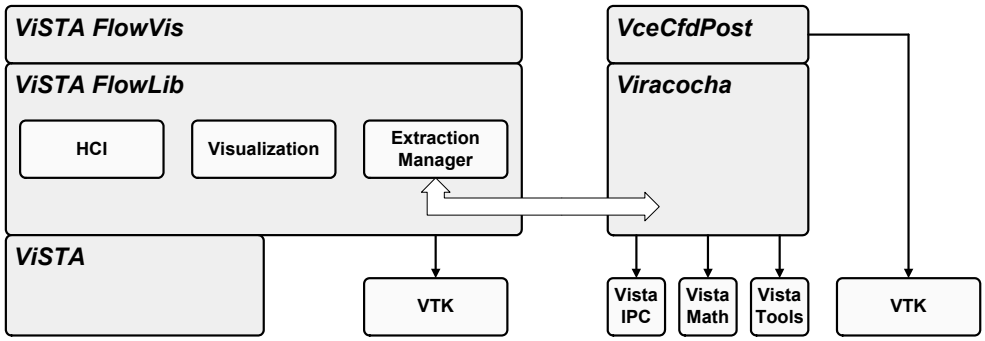

Abbildung 9: Komponenten und Abhängigkeiten von ViSTA FlowLib

Auch ViSTA FlowLib hat nicht das komplexe Konzept der Visualisierungspipeline nachimplementiert. Vielmehr bindet es VTK [SCHR98, SCHR01] ein (vgl. Absatz 2.3.1), das neben dem Pipeline-Konzept auch gleich eine ganze Reihe von Postprocessing-Funktionalitäten mitliefert. Da es möglich ist VTK auf beliebigen Betriebssystemplattformen zu übersetzen, erfüllt auch ViSTA FlowLib die allgemein von ViSTA erhobene Anforderung der Plattformunabhängigkeit.
Ursprünglich ist ViSTA FlowLib entwickelt worden, um in Anwendungen zum Einsatz zu kommen, die ganz spezielle Strömungsprobleme betrachten und daher sehr individuelle Anforderungen an das CFD-Postprocessing stellen. Dennoch wurde ViSTA FlowLib eine auf ihr aufbauende lauffähige Applikation mit dem Namen ViSTA FlowVis zur Seite gestellt. Dieses Programm dient zum einen als Evaluierungsplattform für die in ViSTA FlowLib verankerten Funktionalitäten. Zum anderen ist es aber auch als vollwertiger Datensatzbetrachter für instationäre Strömungsdatensätze geeignet, der dank ViSTA FlowLib mit weit reichender Postprocessing-Möglichkeit ausgestattet ist. Daher wird hiermit eine Applikation angeboten, die für die meisten Standarduntersuchungen zeitvarianter Strömungsphänomene vollkommen ausreichend ist.

3.3 Das Design von Viracocha

Da neben der Visualisierung instationärer Strömungsvorgänge die Behandlung sehr großer Datensätze eines der wesentlichen Ziel von ViSTA FlowLib darstellt, kommt es schnell dazu, dass die Extraktion von Strömungsmerkmalen sehr rechenintensiv wird und hohe Anforderungen an die Rechnerressourcen gestellt werden. Würde daher das Postprocessing auf dem gleichen Rechner durchgeführt, der auch für die Visualisierung der instationären Strömungsphänomene zuständig ist, käme es zwangsläufig zur Beeinflussung der Render-Performanz. Da jedoch, wie in der Einleitung dargelegt, die flüssige Visualisierung und die echtzeitfähige Interaktion die wesentliche Grundlage jeder virtuellen Umgebung sind, muss der Renderleistung auf dem Visualisierungsrechner absolute Priorität eingeräumt werden. Dies würde dann wegen der verlangsamten Berechnung jedoch zu Lasten der Reaktionszeit

des Postprocessing-Systems gehen. Daher wurde in ViSTA FlowLib das zeitaufwändige Berechnen von Extraktionsmerkmalen von der Visualisierung entkoppelt und auf einen zweiten Rechner ausgelagert. Auf der Visualisierungsseite bleibt somit nur noch ein *Extraction Manager* übrig, der die Verwaltung von Berechnungsaufträgen und der entstehenden Kommunikation zwischen Visualisierung und Berechnung übernimmt.
Das eigentliche Postprocessing wird nun von einem Framework namens Viracocha durchgeführt. Bereits durch die Entkopplung stehen nicht nur der Visualisierung, sondern nun auch der Berechnung sämtliche lokale Rechnerressourcen zur Verfügung. Außerdem lässt sich jeweils optimierte Hardware verwenden. Während sich auf dem Visualisierungsrechner speziell für das Rendern komplexer Szenen optimierte Graphiksubsysteme integrieren lassen, befinden sich auf dem Extraktionsrechner z. B. effiziente I/O-Systeme für den schnellen Zugriff auf die zu verwendenden großen Datensätze. Zudem ist hier der Hauptspeicher entsprechend ausgebaut um solche sehr großen Datenmengen verarbeiten zu können.
Das besondere Merkmal von Viracocha ist jedoch, dass das Postprocessing parallelisiert vonstatten gehen kann. Durch den Einsatz von Hochleistungsrechnersystemen kann so ein erheblicher *Speed-up* der Berechnung erreicht werden. Viracocha ist dabei nicht auf eine spezielle Hardware-Architektur festgelegt. So lassen sich einfach vernetzte Rechnerverbünde, optimierte Linux-Cluster bis hin zu beliebigen Supercomputersystemen einsetzen. Sogar das Ausführen zusammen mit der Visualisierung auf einer einzigen Workstation ist möglich, was dem Bestreben der Entkopplung widerspricht, für eine Standalone-Lösung jedoch wünschenswert ist.
Viracocha verbindet den Parallelrechner (als *WorkHost* bezeichnet) über einen Befehls- und einen Datenkanal mit dem Visualisierungsrechner (im Folgendem nur noch als *VisHost* bezeichnet). Der Aufbau der Verbindungskanäle geschieht zur Laufzeit. Dadurch können *VisHost* und *WorkHost* unabhängig voneinander gestartet werden. So spricht auch nichts dagegen, dass mehrere *VisHosts* einen gemeinsamen *WorkHost* verwenden, bzw. ein *VisHost* seine Anfragen auf mehrere *WorkHosts* verteilt. Da der *WorkHost* mehrere *VisHost*-Verbindungen verwalten kann, ist es jederzeit möglich, dass sich ein *VisHost* neu an- bzw. wieder abmeldet. Somit dient der *WorkHost* quasi als Dienstleistungsserver, der jederzeit Postprocessing-Aufträge über den Befehlskanal empfangen kann und die Ergebnisdaten später über den Datenkanal zurücksendet.

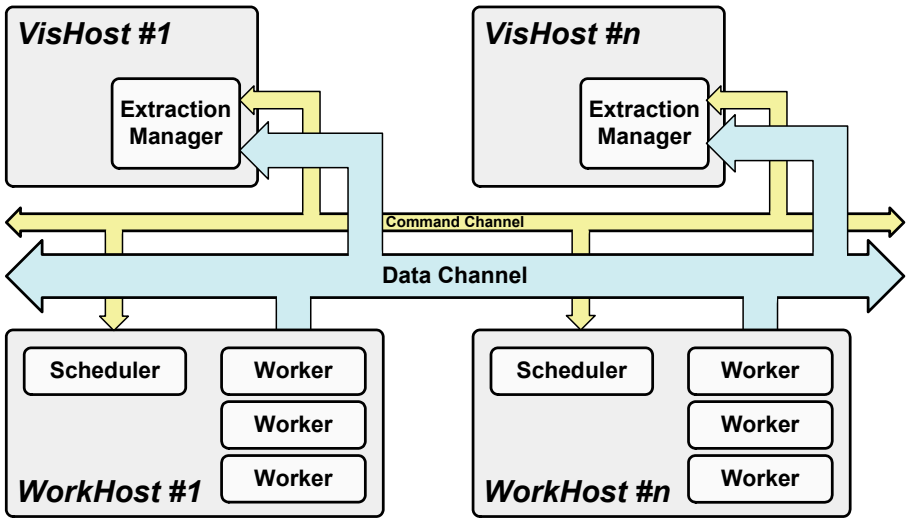

Abbildung 10: ViSTA FlowLib als verteiltes System

Die wesentlichen Objekte, die von Viracocha auf dem *WorkHost* angeboten werden, sind der *Scheduler* und eine feste Anzahl von *Workern*. Werden Berechnungsanfragen vom *VisHost* an den *WorkHost* gesendet, werden diese vom *Scheduler* als Aufgabenverwaltungseinheit empfangen. Sind genügend *Worker* für die beauftragte Berechnung verfügbar, wird eine Arbeitsgruppe (*Work Group*) zusammengestellt, die sodann den Auftrag parallel bearbeitet.

Schematisch sind die wichtigsten Komponenten des verteilten Postprocessing-Systems in Abbildung 10 dargestellt.
Um eine bessere Kontrolle über die belegten Ressourcen zu erhalten, werden *Scheduler* und *Worker* jeweils als einzelne Prozess-Elemente (PEs) realisiert. Bei einer ausreichenden Anzahl von Prozessoren sollten die einzelnen PEs dann tatsächlich parallel ablaufen. Dabei gibt Viracocha weder die zu verwendende Parallelisierungsschnittstelle noch die Kommunikationsprotokolle vor. Noch nicht einmal die tatsächlich zum Einsatz kommenden Berechnungsalgorithmen sind Viracocha bekannt. Diese Flexibilität wird durch die Definition mehrerer Abstraktionsschichten erreicht, die im Folgenden ausführlicher dargestellt werden.

3.3.1 Das Layer-Konzept

Das Parallelisierungs-Framework wird in drei Schichten (engl.: *Layers*) unterteilt. Durch die Definition einiger weniger Schnittstellen zwischen den einzelnen Schichten wird eine Kapselung erreicht, die es ermöglicht ohne Seiteneffekte die jeweiligen Konzepte und Komponenten weiterzuentwickeln. Die Schichten mit ihren Hauptkomponenten sind in Abbildung 11 zusammengefasst.

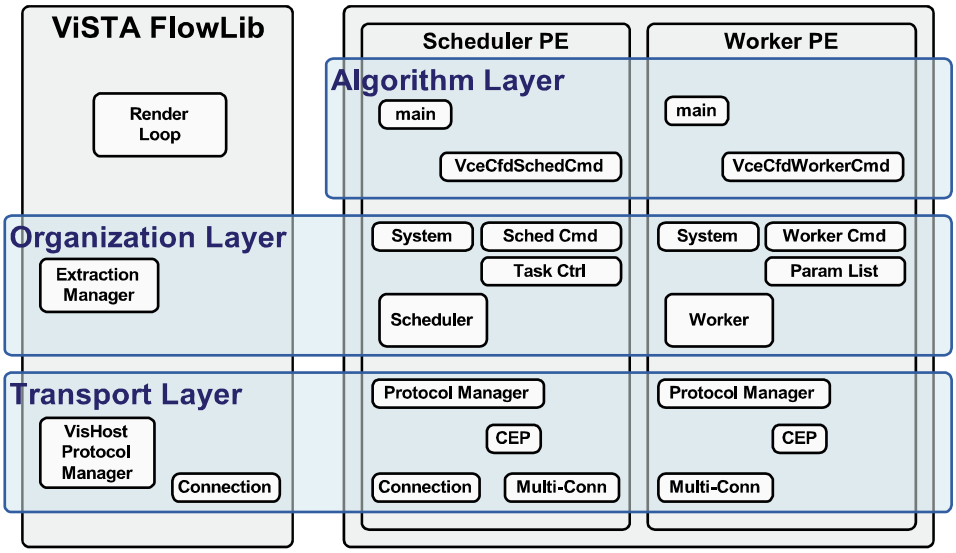

Abbildung 11: Die drei Schichten des Parallelisierungskonzeptes mit den wichtigsten Komponenten

Die bisher dargestellten Hauptkomponenten von Viracocha befinden sich vor allem auf der mittleren Schicht, die als Organisationsschicht bezeichnet wird. Hier werden Verbindungen zu *VisHosts* entgegengenommen, Berechnungsaufträge empfangen und verwaltet und bei ausreichender Anzahl freier *Worker* neue Berechnungen gestartet.
Die Behandlung der Verbindungserstellung, Auswahl geeigneter Protokolle, sowie das Verschicken von Informationen und Daten über Prozess- und Rechnergrenzen hinweg werden in der Transportschicht gekapselt. *Scheduler* und *Worker* können in der Organisationsschicht somit ihre Arbeit auf einem höheren Level organisieren, ohne konkretes Wissen über Eigenarten des jeweils verwendeten Transportmechanismus besitzen zu müssen.
Die zu berechnenden Aufträge werden in der Organisationsschicht jedoch ebenfalls nur abstrakt als *Tasks* behandelt. In Parameterlisten verpackt, werden die Auftragsdaten lediglich vom *Scheduler* an selektierte *Worker* weitergeleitet, wo dann wiederum *Class Factories* (zu Design-Mustern: vgl. [GAMM96]) zur Erzeugung der tatsächlichen Berechnungsprozesse mit den Parameterlisten gefüttert werden. Die *Class Factories* werden von der dritten Schicht, der Algorithmenschicht, erzeugt und dann in der Organisationsschicht registriert. Die Berechnungsklassen sind allesamt von in der Organisationsschicht bekannten Basisklassen abgeleitet. Somit wird auch hier eine weitestgehende Abstrahierung erreicht.

3.3.1.1 Die Transportschicht

In der untersten Ebene wird der Transport von Daten und Nachrichten übernommen. Dabei ist das zu verwendende Transportprotokoll nicht vorgegeben. Zudem können unterschiedliche Protokolle für die Kommunikation zwischen *VisHost* und *WorkHost* sowie zwischen den PEs im *WorkHost* zum Einsatz kommen. Während vorliegende Viracocha-Lösungen für die *WorkHost*-interne Kommunikation momentan ausschließlich auf MPI aufsetzen, kann für die Kommunikation zwischen *VisHost* und *WorkHost* zwischen TCP/IP und MPI gewählt werden.
Für die transparente Behandlung des Datentransports existieren einige verallgemeinerte Datenobjekte, mit denen auch auf höheren Ebenen von Viracocha gearbeitet werden kann, ohne bereits ein spezielles Protokoll vorauszusetzen. Um zwischen zwei Prozessen kommunizieren oder Daten austauschen zu können, müssen *Connections* aufgebaut werden. Die Adressen der Kommunikationspartner werden durch *Connection End Points* (CEP) spezifiziert. Da sich Adressenbeschreibungen von Protokoll zu Protokoll stark unterscheiden können, kapseln CEPs die exakte Adressspezifizierung. CEPs sind zudem serialisierbar, d. h. sie lassen sich zu einem Datenstrom dekodieren und sodann leicht als Informationseinheit über das Netzwerk verschicken. So können sie verwendet werden um verteilte Prozesse mit notwendigen Informationen für den Aufbau neuer Verbindungen zu versorgen.
Mehrfachverbindung lassen sich durch *Multi-Connections* kreieren. Auch hier dienen die CEPs zur Beschreibung der Kommunikationsendpunkte. Allerdings stellt die *Multi-Connection* eine *n*-zu-*n*-Verbindung dar, die dazu dient Daten an alle beteiligten Verbindungsenden gleichzeitig zu versenden. Neben diesem *Broadcasting* ist aber auch eine 1-zu-1-Kommunikation möglich, ohne dass eine separate *Connection* erstellt werden müsste (vgl. Abbildung 12).

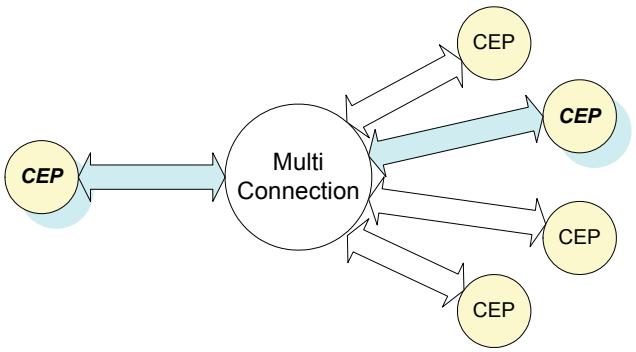

Abbildung 12: *Multi-Connection* für die Kommunikation zwischen mehreren Prozessen

Damit die richtigen Objekte in der Transportschicht erzeugt werden können, dient ein so genannter *Protocol Manager* als *Class Factory*. Mit dem Wissen versehen, welche Verbindungen nun tatsächlich von Objekten der nächst höheren Schicht benötigt werden, erzeugt er die richtigen spezialisierten Verbindungsinstanzen, liefert aber lediglich die allgemeinen Schnittstellen nach außen zurück. Im Gegensatz zum *Protocol Manager* besitzen die restlichen Klassen der Transportschicht keinerlei Wissen über ihre Verwendung durch Objekte höherer Schichten und verwenden nur diese Schnittstellen. Somit wird eine weitestgehende Kapselung der Transportschicht erreicht.

3.3.1.2 Die Organisationsschicht

Das zentrale Objekt in der Organisationsschicht ist der *Scheduler*. Er ist als eigenständiger Prozess ausgelegt und übernimmt ausschließlich Verwaltungs- und Organisationsaufgaben. Seine Hauptaufgaben, die er der Reihe nach permanent in einer Endlosschleife bearbeitet, sind:

1. ***VisHost*-Verbindung**: eingehende Verbindungen akzeptieren und geschlossene Verbindungen aus der Liste der aktuellen *VisHost*-Verbindungen austragen.

2. **Anfragen bearbeiten**: neue Berechnungsanforderungen annehmen und vorbereiten, Steuerungsbefehle umsetzen und sonstige Anfragen beantworten.
3. **Berechnungen durchführen**: in Verwaltungslisten nach unbearbeiteten Anforderungen schauen und, bei ausreichender Anzahl von freien *Workern*, Berechnung starten.
4. **Berechnungen koordinieren**: laufende Berechnungen durch den so genannten *Scheduler Command* koordinieren und steuern.

Viracocha ist als selbständiger Server konzipiert. Beliebige Visualisierungs-Clients können demnach konnektieren und Extraktionsberechnungen beauftragen. Um eintreffende Verbindungsanfragen von Clients zu erkennen, fragt der *Scheduler* ständig beim so genannten *Protocol Manager* nach, der für den Verbindungsaufbau zuständig ist. Liegt tatsächlich eine neue Verbindung vor, liefert der *Protocol Manager* aber nicht einfach die zugehörige *Connection* sondern einen speziellen *Socket* zurück. *Sockets* sind Bestandteile der Organisationsschicht und bieten Methoden für die sichere Kommunikation zwischen zwei Kommunikationspartnern an, indem speziell für die jeweilige Verbindung angepasste Methoden mit optimierter Fehlerbehandlung zur Verfügung gestellt werden. Im Falle einer Verbindung zu einem *VisHost* erhält der *Scheduler* einen *SchedulerVisHostSocket*, der Methoden zur Abfrage von Anfragen sowie deren Beantwortung enthält.
Für die Kommunikation zwischen *VisHost* und *WorkHost* werden zwei Arten von Verbindungskanälen unterschieden:

Command Channel: Dieser besteht immer nur zwischen einem *VisHost* und dem *Scheduler*. Über ihn werden Extraktionsanforderungen verschickt und als Quittierung die *Task ID* zurückgesendet. Ebenso werden hierüber die weiteren Anfragen über den Status des *WorkHosts* und sonstige Steuerungsbefehle abgewickelt.

Data Channels: Diese werden üblicherweise von *Workern* initiiert und für das Versenden der Berechnungsergebnisse verwendet. Findet das Zusammenfügen der parallel berechneten Ergebnisse auf dem *WorkHost* statt, wird lediglich ein einziger Datenkanal pro Zeitschritt benötigt. Beim Streaming (siehe Kapitel 5) kommt dagegen üblicherweise ein Datenkanal pro involviertem Worker zum Einsatz. Die benötigte Bandbreite für *Data Channels* liegt um Faktoren über der des *Command Channels*.

Der *SchedulerVisHostSocket* liefert bereits den zugehörigen geöffneten *Command Channel* zurück. Aber auch die den *Workern* zugeordneten *Sockets* übernehmen die sichere Erstellung der *Data Channels*, ohne dass sich die *Worker* um das konkrete Zustandekommen der Verbindung und der exakten Reihenfolge etwaiger *Handshake*-Parameter oder ähnliches kümmern müssen. Dieses vereinfachte Kommunikationskonzept per *Sockets* existiert für alle Bestandteile der Organisationsschicht, die hierüber auch untereinander kommunizieren.
Trifft über einen *Command Channel* eine Berechnungsanforderung ein, sendet der *Scheduler* eine Art Auftragsbestätigungsnummer, die so genannte *Task ID*, zurück. Anhand dieser kann der *VisHost* später über einen Datenkanal eintreffende Daten wieder dem Auftrag zuordnen und bei Bedarf weiterverarbeiten. Mithilfe der *Task ID* kann er aber auch den Status seiner Berechnungsanforderung abfragen. Für die explorative Analyse besonders wichtig ist jedoch die Möglichkeit eine laufende Berechnung abbrechen zu können, um dann eine modifizierte Anforderung zu übermitteln.
Der *Scheduler* veranlasst aber nach dem Eintreffen eines Extraktionsbefehls nicht unverzüglich die Berechnung. Vielmehr generiert er einen *Task Controller*, der für die ordnungsgemäße Durchführung der parallelen Bearbeitung zuständig ist, und hängt ihn in eine Liste mit noch wartenden *Tasks*. Neben dieser Liste verfügt der Scheduler noch über eine Liste gerade laufender Berechnungen (*Busy Tasks*), sowie über eine Liste von zurzeit nicht beschäftigter *Worker*. Sind hier ausreichend viele *Worker* frei, werden sie dem *Task Controller* mit der höchsten Priorität übergeben, der von der Liste der wartenden *Tasks* in die

Busy Task List umsortiert wird. Nach der Abarbeitung des Berechnungsbefehls werden die zugeordneten *Worker* wieder in die Liste der freien *Worker* einsortiert. Der *Task Controller* wird aus der Liste der beschäftigten *Tasks* gelöscht und anschließend zerstört.
Der *Task Controller* ist also die zentrale Instanz für die Behandlung einer Berechnungsanforderung. Neben dem CEP des Auftraggebers, einer Liste von *Workern* und der *Task ID* erhält er zusätzlich eine Parameterliste, in der die konkreten Informationen über Art und Umfang der Berechnung stecken. Außer für die ersten beiden Parameter, die eine *Scheduler Command ID* und eine *Worker Command ID* beinhalten, ist er nicht in der Lage diese Informationen zu interpretieren, da hier vor allem algorithmusspezifische Daten kodiert sind. Daher erzeugt er nun zuerst einen *Scheduler Command*, indem er die von der Algorithmusschicht instanziierte *Scheduler Command Factory* verwendet und mit *Scheduler Command ID* und der Parameterliste versorgt. Diese liefert einen *Scheduler Command* zurück. Danach sendet der *Task Controller* allen *Workern* die *Worker Command ID* und die Parameterliste zu, die sodann die *Worker Command Factory* zum Erzeugen des *Worker Commands* aufrufen. Abschließend bildet der *Scheduler Command* mit den *Worker Commands* eine lokale Arbeitsgruppe. Für die einfachere Adressierung erhält jede Instanz in dieser *Work Group* eine lokale *Rank ID* über die untereinander kommuniziert werden kann. Während der *Scheduler Command* Koordinationsaufgaben innerhalb der *Work Group* übernehmen soll, wird von den *Worker Commands* die eigentliche Berechnung parallel durchgeführt. Die einzelnen hier dargestellten Schritte sind nochmals schematisch in Abbildung 13 skizziert.

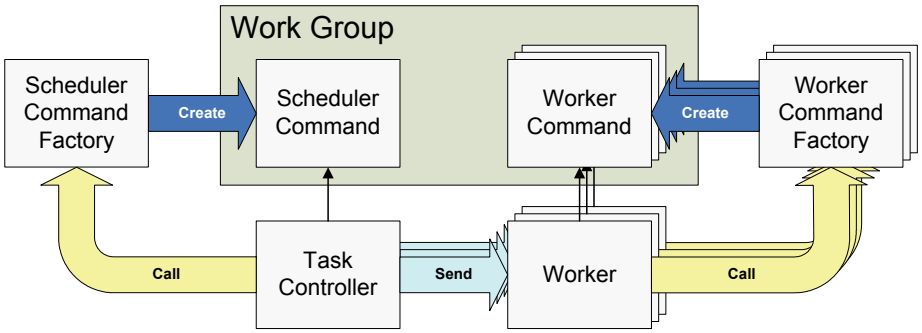

Abbildung 13: Erzeugung einer *Work Group* mit den wesentlichen Schnittstellen zur Algorithmusschicht

Durch die beschriebene Organisation von Berechnungserzeugung und -verwaltung wird erreicht, dass Viracocha ein anwendungsneutrales Parallelisierungs-Framework ist, d. h. dass bisher nicht festgelegt wurde, welche Datenstrukturen und welche Berechnungsvorschriften angewendet werden. Wie einfach sich nun Algorithmen für spezielle Aufgabengebiete implementieren lassen, beschreibt der folgende Abschnitt.

3.3.1.3 Die Algorithmenschicht

Der Anspruch für die Algorithmenschicht ist ähnlich dem von ViSTA-Applikationen: Die Entwicklung eigener Algorithmensammlungen sollte möglichst einfach gestaltet werden. Dies bedeutet, dass viel framework-spezifisches Wissen von Viracocha gekapselt werden muss und die zu implementierenden Schnittstellen leicht verständlich sein müssen. Daher gibt es wie bei ViSTA eine *System*-Komponente, die von der Berechnungsapplikation zuerst erzeugt und initialisiert werden muss. Dadurch wird kaskadierend durch alle Schichten hinweg eine ganze Reihe weiterer Viracocha-Komponenten erzeugt und initialisiert, sodass das Parallelisierungs-Framework sofort einsatzfähig ist.
Dieser erste Schritt geschieht klassischerweise in der *main*-Routine, die als Startfunktion einer jeden Applikation gilt. Um jedoch von außen später auch tatsächlich eigene Berechnungsalgorithmen aufrufen zu können, müssen die *Class Factories* für eigene *Scheduler Commands*

und *Worker Commands* erzeugt und bei Viracocha registriert werden. Die *main*-Routine sieht daher üblicherweise folgendermaßen aus:

```
int main (int argc, char* argv[])
{
    CVceSystem  viracocha;

    CVceCfdWorkerCmdFactory *    pCfdWorkerCmdFac = NULL;
    CVceCfdSchedulerCmdFactory * pCfdSchedCmdFac  = NULL;

    pCfdWorkerCmdFac = new CVceCfdWorkerCmdFactory();
    pCfdSchedCmdFac  = new CVceCfdSchedulerCmdFactory();

    viracocha.SetApplWorkerCommandFactory    (pCfdWorkerCmdFac);
    viracocha.SetApplSchedulerCommandFactory (pCfdSchedCmdFac);

    // endless run loop
    viracocha.StartProcessElements (argc, argv);

    delete pCfdWorkerCmdFac;
    delete pCfdSchedCmdFac;
    return 0;
}
```

Listing 1: *Main*-Routine zum Initialisieren und Starten von Viracocha

Die Implementierung der jeweiligen *Factory* gestaltet sich extrem einfach. Es muss lediglich eine einzige Methode zum Erzeugen neuer anwendungsbezogener *Scheduler Commands* bzw. *Worker Commands* implementiert werden. Mit der Übergabe einer entsprechenden Kommando-ID wird die spezialisierte Kommandoinstanz erzeugt und zurückgegeben. Der *Task Controller* verwendet dann die Schnittstelle des *Scheduler Commands* für einen Initialisierungsaufruf, und später wird ständig dessen *Run*-Methode aufgerufen, um Koordinations- und Kontrollaufgaben der *Work Group* durchzuführen.
Die *Worker* hingegen initialisieren ihre jeweilige Instanz des *Worker Commands* mit dem Schnittstellenaufruf der *Parse*-Routine, der die vom *VisHost* übermittelte Parameterliste mitgegeben wird. Erst die spezialisierte Variante des *Worker Commands* ist nun in der Lage, die in speziellen Parametern abgespeicherten Berechnungsinformationen auszuwerten. Ist beispielsweise durch die Auswahl einer entsprechenden *Worker Command ID* von der *Worker Command Factory* ein *Isosurface Worker Command* erzeugt worden, wird dieser in der Parameterliste zumindest die Angabe über den zu verwendenden Isowert finden. Nach dem Parsen der Parameterliste ist der *Worker Command* initialisiert und kann mit dem Aufruf der *Run*-Routine gestartet werden.
Da alle Instanzen eines *Worker Commands* in der Regel aus identischem Code bestehen, findet eine Arbeitsteilung vor allem über die in der *Work Group* eindeutig zugewiesen lokalen *Rank IDs* statt. Dabei ist der *Worker Command* nun jedoch nicht zur starren Ausführung des implementierten Codes gezwungen. Vielmehr hat es Zugriff auf einen *Command Socket*, um mit den Teilnehmern der *Work Group*, also dem einen *Scheduler Command* und den anderen Instanzen des *Worker Commands*, zu kommunizieren. Hierüber ist eine weit reichende Dynamisierung der Berechnung möglich.

3.4 Paralleles CFD-Postprocessing mit Viracocha

Aufbauend auf Viracocha und unter Verwendung der beschriebenen Schnittstellen für die *Class Factories* und *Commands* wurde als Algorithmusschicht die CFD-Postprocessing-Applikation mit dem Namen VceCfdPost[9] entwickelt. Das Hauptaugenmerk lag dabei vor

[9] Das Kürzel *Vce* ist die historisch bedingte interne Bezeichnung für Viracocha, und wird für die Programmierung auch als Bezeichner für den Namensraum verwendet.

allem auf der Extraktion instationärer Strömungsvorgänge und auf der Detektion globaler Strömungsmerkmale, wie Wirbelstrukturen und Topologieinformationen skalarer und vektorieller Datenfelder. In VceCfdPost ist wiederum die Visualisierungspipeline (vgl. Abschnitt 2.1) zentral, die für alle Berechnungen zum Einsatz kommt. Um nicht den Update-Mechanismus und benötigte Datenstrukturen vollständig neu entwickeln zu müssen, wurde auf VTK (vgl. Abschnitt 2.3.1) zurückgegriffen. Für einfache Standardextraktionsverfahren, wie der Berechnung von Schnittflächen, Isoflächen und Stromlinien, wurden ebenfalls bereits von VTK angebotene Klassen verwendet. Aufwändigere Verfahren wurden dagegen selbst entwickelt.
Neben der Entlastung des Visualisierungsrechners ermöglicht Viracocha als Parallelisierungs-Framework zuallererst die Beschleunigung der Berechnung. Zudem dient es als Testplattform für neue, effiziente Algorithmen. Hierunter sind vor allem Methoden zu verstehen, die durch einen Vorverarbeitungsschritt Datensätze umorganisieren oder mit Meta-Daten versehen, um anschließend wesentlich schneller Berechnungsaufträge durchführen zu können. Dabei lässt sich auch untersuchen, wie zeitaufwändig es ist, wenn Viracocha die Erzeugung der Meta-Daten zur Laufzeit mit durchführt um zusätzliche Daten auf Festspeichermedien zu vermeiden. Letztendlich kann der durch Viracocha gewonnene Spielraum genutzt werden, um durch aufwändigere Verfahren höhere Genauigkeiten zu erreichen. Die vier Hauptziele des parallelen Postprocessings sind somit:

- Entlastung des VR-Systems
- *Speed-up* durch Parallelisierung
- Integration effizienter Algorithmen
- Höhere Genauigkeit

In diesem Abschnitt werden zuerst einige grundsätzliche Fragenstellungen bezüglich der Parallelisierung von Datenstrukturen behandelt. Anschließend werden neben Schnittflächen vor allem Partikelverfolgungsverfahren, Isoflächenextraktion, sowie die Berechnung von kritischen Punkten auf ihre grundsätzliche Parallelisierungsfähigkeit hin untersucht und mit verschiedenen Datensätzen evaluiert. Dabei werden unterschiedliche Teilaspekte, die sich während der Auswertung der Ergebnisse ergaben, genauer untersucht. Neben dem *Speed-up* stellten Skalierbarkeit und Balancierungsaspekte wichtige Bewertungskriterien dar.

3.4.1 Parallelisierung mithilfe des Master-Worker-Konzepts

Der einfachste Ansatz Algorithmen zu parallelisieren ist die Datenparallelisierung (vgl. Abschnitt 2.1), die den sequentiellen Abschnitt der Berechnungspipeline repliziert und auf alle *Worker* verteilt. Diese arbeiten unabhängig voneinander auf einem Teilbereich des Eingangsdatensatzes, der anhand einer bestimmten Heuristik bestimmt wird. Dieses Vorgehen funktioniert für alle Extraktionsalgorithmen, die lediglich lokale Informationen für die Berechnung benötigen, d. h. ihre Teilergebnisse aus den vorliegenden Gitterzellen ermitteln können.
Die berechneten Teile müssen wieder zu einem Gesamtergebnis zusammengesetzt werden. Das implementierte *Master-Worker*-Schema selektiert einen der berechnenden *Worker* als *Master*, der die Aufgabe hat nach Beendigung seiner eigenen Berechnung die Teilergebnisse der anderen einzusammeln und zu einem Gesamtpaket zusammenzufügen. Dieser Arbeitsschritt wird auch als Datenreduktion bezeichnet. Das Resultat wird sodann von ihm an den *VisHost* übermittelt. Ausgewählt wird diese Strategie, indem der *VisHost* bereits bei der Berechnungsanforderung zusätzlich eine entsprechende ID für den *Scheduler Command* angibt. Es wird dann der *Master Worker Scheduler Command* für die aktuelle *Work Group* instanziiert, der dafür sorgt, dass der erste *Worker*, der sich beim *Scheduler Command* zurückmeldet, von ihm zum *Master* ernannt wird. Grundsätzlich kann aber auch ein *Worker* (z. B. derjenige mit der lokalen *Rank-ID* 1) von vornherein als *Master* bestimmt sein, an den

dann die anderen ihre Teilergebnisse schicken müssen. Es entfällt dann die Kommunikation mit dem *Scheduler*. Ein Beispielszenario ist in Abbildung 14 dargestellt.

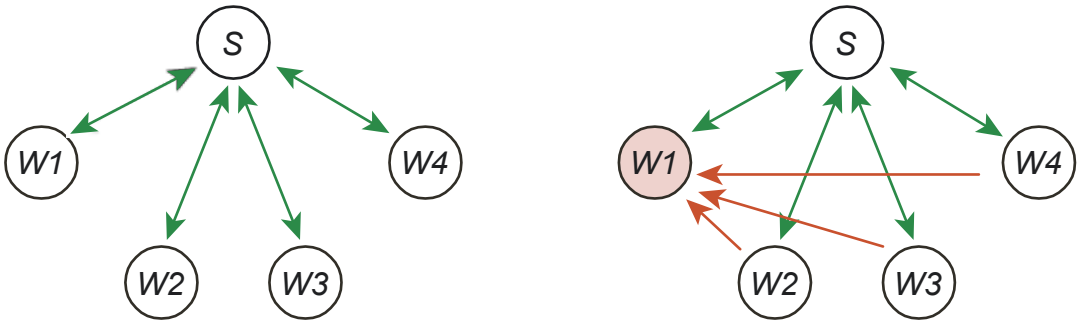

Abbildung 14: Der *Scheduler* versendet in einem ersten Schritt die Berechnungsbefehle (links), ein *Master-Worker* (hier W1) sammelt abschließend alle berechneten Daten zentral ein (rechts)

Das Konzept des *Master-Workers*, bei dem nur ein dedizierter *Master* existiert, der von den anderen *Workern* nach Beendigung ihrer Berechnungen bedient wird, lässt sich noch erweitern:

Hierarchische *Master*: Beim Konzept des *Master-Workers* kann es zu Balancierungsproblemen kommen, wenn viele Berechnungsknoten zur selben Zeit Daten zum *Master* schicken wollen. Beim hierarchischen *Master* werden dagegen bei Bedarf mehrere *Worker* dynamisch zu einsammelnden *Mastern* ernannt und können anschließend die zusammengefügten Teildaten weiterschicken. Abbildung 15 verdeutlicht diesen Ansatz:

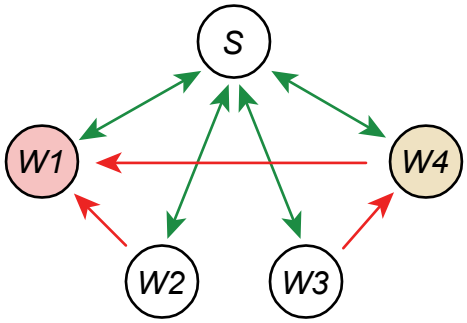

Abbildung 15: Der *Scheduler* bestimmt mehrere Knoten zum hierarchischen Sammeln von Teildaten

Dynamische Pipelineparallelisierung: Statt lediglich im abschließenden Schritt der Datenreduktion den Austausch von Daten über Prozessgrenzen hinweg vorzusehen, lassen sich auch Algorithmen entwickeln, die Teilergebnisse anderer *Worker* für weiterführende Berechnungen verwenden. Diese Pipeline-Parallelisierung lässt sich in Abhängigkeit der Auslastung der einzelnen Worker ebenfalls dynamisieren.

Streaming: Wird der Datenreduktionsschritt auf den *VisHost* verlagert, kann auf Kommunikation zwischen *Workern* vollständig verzichtet werden. Allerdings wird dann pro *Worker* ein *Data Channel* zum *VisHost* benötigt. Da die Teilergebnisse durch unterschiedliche Berechnungslast zu verschiedenen Zeitpunkten am *VisHost* eintreffen, besteht die Möglichkeit bereits vorliegende Daten schon einmal zu visualisieren. Der Anwender kann somit verfolgen, wie sich das Endergebnis nach und nach aufbaut. Dieser Vorgang wird als Streaming bezeichnet. Weitere Streaming-Ansätze, die sich in Kombination mit *Multiresolution* und *Progressive Meshes* ergeben, werden ausführlich in Kapitel 5 behandelt.

In Viracocha sind als *Scheduler Commands* lediglich das *Master-Worker*-Konzept und das Streaming implementiert. Pipeline-Parallelisierung existiert ansatzweise für einige spezielle

Algorithmen, wobei keine Dynamisierung durchgeführt wird. Der hierarchische *Master* wurde bisher nicht weiter verfolgt, da in Kapitel 4 weitergehende Dynamisierungsstrategien untersucht werden, die bereits in den unteren Schichten von Viracocha verankert sind und somit unabhängig vom implementierten Extraktionsalgorithmus arbeiten.

3.4.2 Topologie von Multi-Block-Datensätzen

Für die Vernetzung komplexer Strömungsgebiete kommen heutzutage vorwiegend unstrukturierte Gitter oder Multi-Block-Datenstrukturen (MB, *Multi Blocks*) zum Einsatz. Einen Überblick über verwendete Datengitter für die Strömungssimulation findet man zum Beispiel bei [HAMA97]. Besonders die weit verbreiteten und effizienten Multi-Grid-Strömungslöser, die Navier-Stokes-Gleichungen mit einer hohen Anzahl von Unbekannten mithilfe der Finite-Volumen-Methoden berechnen können, machen von MB-Datenstrukturen Gebrauch [GROS96]. Als Beispiel ist in Abbildung 16 der in dieser Arbeit häufig verwendete Vierventilmotor (vgl. Anhang A) dargestellt.

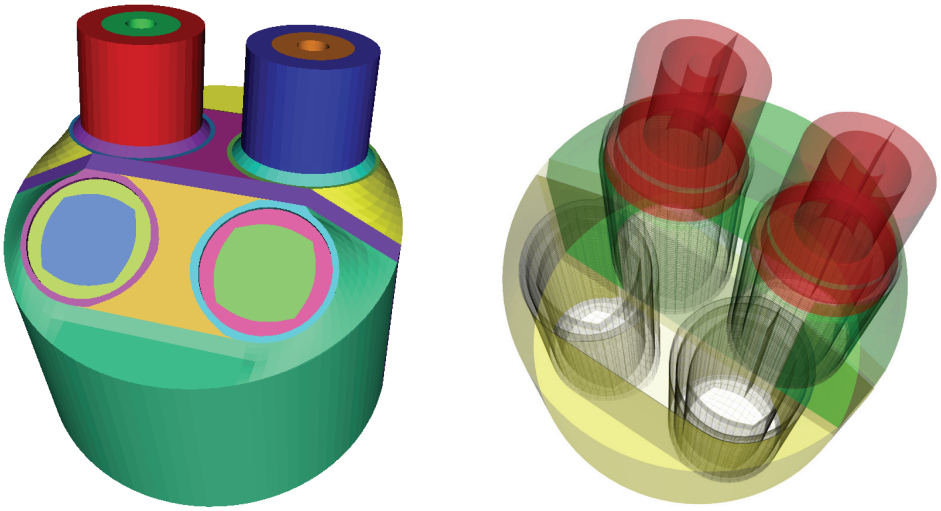

Abbildung 16: Multi-Block-Datensatz, Einzelblöcke farbkodiert (links), Skelettdarstellung mithilfe von transparent und als Drahtgitter dargestellten Blöcken (rechts)

Die Einzelblöcke bestehen aus rectilinearen Gittern. Dadurch ist die Nachbarschaftsbeziehung von Zellen innerhalb eines Blocks implizit durch die *i-j-k*-Koordinaten des Berechnungsraums, dem so genannten *C-Space*, gegeben (vgl. hierzu [SADA97]). Beim Übergang von einem Block zum nächsten fehlen allerdings diesbezügliche Informationen. Um diese Daten für ein schnelleres Postprocessing zur Verfügung stellen zu können, wurde für Viracocha eine weitere Datenstruktur definiert, die die vorliegende Multi-Block-Topologie ausführlich beschreibt. Die wesentlichen Datenfelder sind in Abbildung 17 dargestellt.

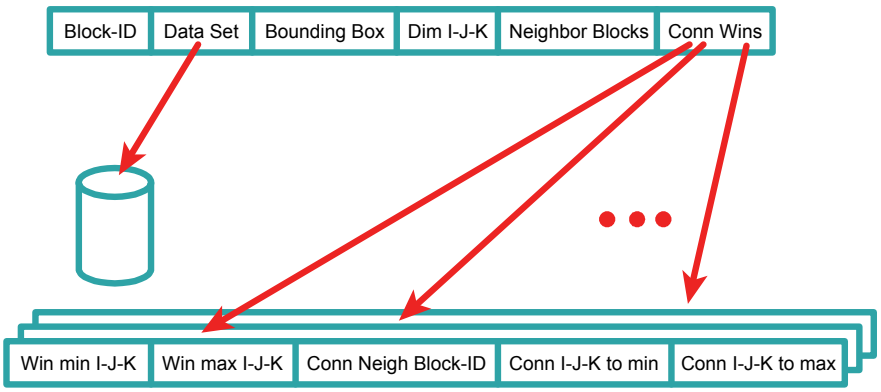

Abbildung 17: Datenstruktur für Multi-Block-Topologieinformation

Die in Viracocha integrierte MB-Topologiebeschreibung ist hierarchisch aufgebaut. Die feinste gespeicherte Nachbarschaftsbeziehung zwischen zwei Blöcken sind Zellverbindungen, wobei davon ausgegangen wird, dass die Gitterpunkte zwischen zwei Blöcken exakt übereinander liegen. Außenzellen eines MB-Blocks sind entweder mit einer Zelle eines benachbarten Blocks verbunden oder stellen eine Außenwand des untersuchten Strömungsfeldes dar. Für die direkte Bestimmung von Nachbarzellen über Blockgrenzen hinweg sind so genannte *Connection Windows* definiert. Statt jede Zellnachbarschafts-beziehung einzeln zu speichern, werden überlappende Bereiche zusammengefasst. Zur Definition werden die *i-j-k*-Indices von zwei diagonal gegenüberliegenden Eckpunkten herangezogen. Da die korrespondierenden Koordinaten beider Blöcke gespeichert werden, sind auch Verdrehungen, wie in Abbildung 18 beispielhaft dargestellt, abbildbar. Ein so definiertes Konnektivitätsfenster ist eindeutig, da nur angrenzende Zellen auf einer der möglichen sechs Seiten eines strukturierten Blockes berücksichtigt werden. Dadurch ist es aber auch möglich, dass mehrere Fenster zwischen Blöcken existieren.

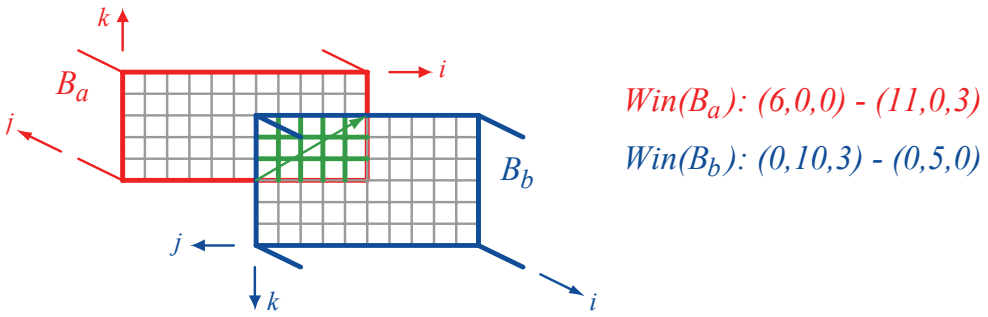

Abbildung 18: *Connection Windows* zwischen benachbarten Blöcken

Die Verbindungsfenster können nun benutzt werden, um den Nachbarblock und die dazu gehörige Nachbarzelle direkt zu bestimmen. Allerdings werden entsprechend der Definition nur Nachbarschaftsbeziehungen aufgelöst, wenn Zellen sich über Zellseiten berühren. Besteht lediglich eine gemeinsame Kante oder nur ein einziger gemeinsamer Zelleckpunkt, dann wird diese Nachbarschaftsbeziehung hierüber nicht erkannt.
In Abbildung 19 sind zwei der curvilinearen Blöcke des Motordatensatzes, die üblicherweise ineinander stecken, dargestellt. Da der äußere Block (rechter Block) den inneren (linker Block) mit seiner *Bottom*-Fläche einmal komplett umschließt und zudem die Startkoordinate dieser Fläche nicht mit einer Startkoordinate einer Fläche des inneren Blocks deckungsgleich ist, existieren insgesamt fünf Verbindungsfenster. Eine Besonderheit ist am äußeren Block zu erkennen. Die mit *Left* und mit *Right* gekennzeichneten Seitenflächen gehen eine vollständige Verbindung ein, der Block ist also mit sich selbst verbunden. Solche Fenster werden Selbstverbindungsfenster genannt.

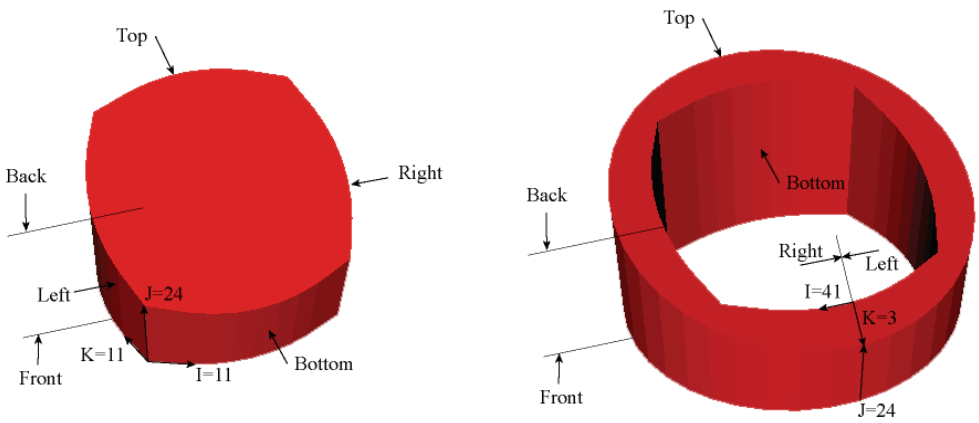

Abbildung 19: Definition zweier ineinander gesteckter Blöcke

Neben den *Connection Windows* enthält die MB-Topologiebeschreibung von Viracocha auch reine Blocktopologieinformationen. In einer einfachen Liste werden die verbundenen Blöcke aufgelistet, wobei wiederum Kanten- und Punktkonnektivitäten nicht berücksichtigt werden. Das Ergebnis kann man auch als Graphen darstellen. Der in Abbildung 16 präsentierte MB-Datensatz ergibt den in Abbildung 20 visualisierten Graphen. Über zyklische Kanten sind Blockverbindungen gekennzeichnet, die sich über Selbstverbindungsfenster ergeben.

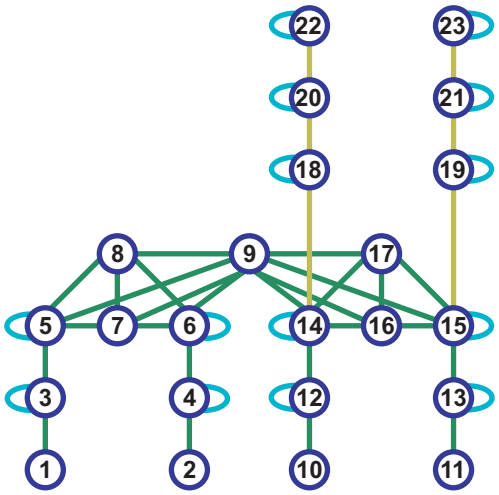

Abbildung 20: Topologiegraph des Multi-Block-Motordatensatzes

Der dritte Level von Topologieinformationen definiert sich über *Bounding-Boxen*. Da hierdurch die minimalen und maximalen Koordinaten entlang der kartesischen Raumachsen gespeichert werden, die von den Datenpunkten eines Blockes eingenommen werden, kann es ggf. zu recht groben Abschätzungen kommen. Soll beispielsweise derjenige Block gefunden werden, der eine vorgegebene Raumkoordinate beinhaltet, dann werden nur diejenigen Blöcke genauer untersucht, deren *Bounding-Boxen* bereits die Raumkoordinate umgeben.
Nicht jede Stufe der vorgestellten Topologiehierarchie muss für einen aktuell zu bearbeitenden Strömungsdatensatz vorliegen. Und nicht jeder Algorithmus lässt sich immer gleich direkt mit den auf einer oberen Stufe vorhandenen Informationen erfolgreich anwenden. Daher kann die Hierarchie von oben nach unten durchlaufen werden, bis ein Level Informationen zur Verfügung stellt, die zum Erfolg führen. Zusammenfassend existieren die folgenden vier Stufen, die für Nachbarschafts- und Positionssuche nacheinander angewendet werden können:

- Zellkonnektivität mithilfe von **Konnektivitätsfenstern**
- **Blocknummern** der benachbarten Blöcke
- ***Bounding-Box***-Information der einzelnen Blöcke
- **Keine Information** über Nachbarschaftsbeziehungen (Suche über Rest)

Viele Berechnungsverfahren benötigen sämtliche Datenblöcke für die Merkmalsextraktion. Andere Algorithmen, wie zum Beispiel Partikelverfolgungsverfahren, können jedoch durchaus von den Topologieinformationen profitieren. Hier lädt Viracocha zuerst die in einer Datei abgelegte Topologiebeschreibung und kann dann anhand der darin gespeicherten Angaben entscheiden, welche Datenblöcke tatsächlich benötigt werden. Nur diese werden auf Anfrage tatsächlich in den Hauptspeicher geladen (*Load-on-Demand*).

3.4.3 Partitionierung von Datensätzen

Bei Multi-Block-Datensätzen liegt jeder einzelne Block in einer eigenen Datei abgespeichert vor. Abhängig von Größe und Partitionierung der Datensätze, Berechnungskomplexität und Latenz des Dateisystems ist die Zeitersparnis durch verteiltes Laden von Teildaten erheblich. Das einfache Beispiel der parallelen Schnittflächenberechnung, dargestellt in Abbildung 21,

belegt den Sachverhalt. Im sequentiellen Fall (1 Prozessor) besteht das Verhältnis der Berechnung zum Laden der Daten insgesamt 1:10. Durch die Parallelisierung wird vor allem die Zeit des Ladevorgangs reduziert, während die Berechnungszeit recht schnell keinen Einfluss mehr auf die Gesamtlaufzeit hat.

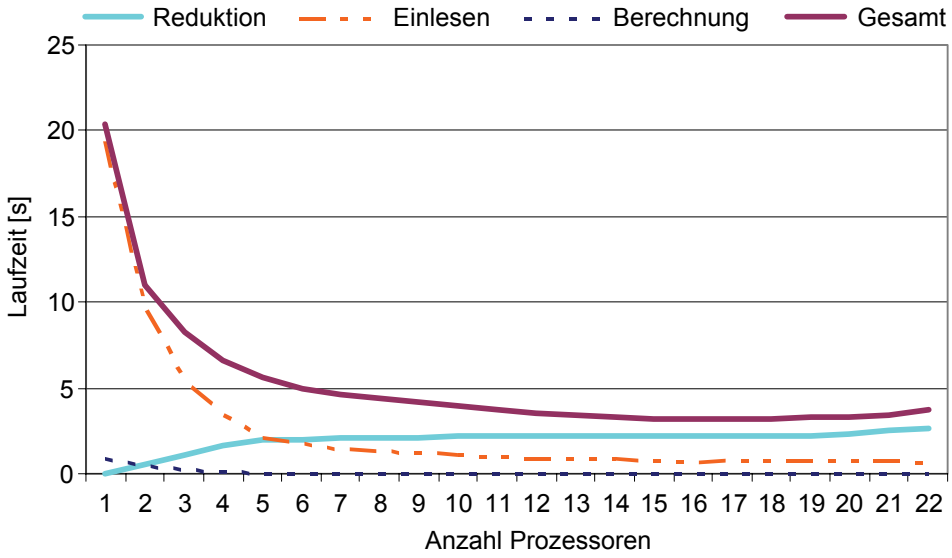

Abbildung 21: Parallele Schnittflächen-Berechnung, Multi-Block-Motordatensatz, ein Zeitschritt, SunFire-Cluster

Ab einer gewissen Anzahl von involvierten Prozessoren lässt sich allerdings auch beim Datenladen keine merkliche Beschleunigung mehr erreichen, da der verwendete Motordatensatz lediglich aus 23 Blöcken pro Zeitschritt besteht. Wie in Abbildung 21 ebenfalls erkennbar, nimmt die Zeit für das Zusammenfügen der Teilergebnisse (*Reduktion*) mit zunehmenden *Workern* zu. Ab 17 involvierten *Workern* steigt daher die Gesamtlaufzeit wieder an. Dieses Sättigungsverhalten ist ein typisches Merkmal für das parallele CFD-Postprocessing, da sequentielle Anteile dominieren und Kommunikation den weiteren Parallelisierungsgewinn zunichte macht. Der Zeitpunkt wird vor allem von Datensatzgesamtgröße, Datensatzzerlegung und Berechnungskomplexität bestimmt.
Statt für Datenparallelisierung nur auf den Einzelblöcken eines diskreten Zeitschrittes zu arbeiten, besteht bei instationären CFD-Datensätzen in der Regel auch die Möglichkeit gleichzeitig Berechnungen auf Daten der verschiedenen Zeitschritte durchzuführen. Ohne eine vorangegangene Partitionierung lassen sich so auch unstrukturierte Datensätze oder strukturierte Datensätze, die nur aus einem Block pro Zeitschritt bestehen (Bsp.: Verdichtungsstoßdatensatz, vgl. Anhang A) direkt einsetzen. Die Skalierbarkeit der Parallelisierung ist dann durch die Anzahl der Zeitschritte begrenzt. Im Fall stationärer Extraktionsverfahren pro Zeitschritt entfällt auch der Reduktionsschritt am Ende der Berechnungspipeline.
Für das Design von Algorithmen spielt die Behandlung von Datensätzen eine wesentliche Rolle. Im einfachsten Fall lädt jeder *Worker* direkt seine Daten, die er zur Berechnung benötigt. Dass ein somit möglicher gleichzeitiger Zugriff auf eine einzige Datei in Abhängigkeit des verwendeten Betriebssystems und der Eigenschaften des installierten Dateisystems ebenfalls zu einem Falschenhals werden kann, demonstriert Abbildung 22. Beim Einladen des mit 3 MBytes recht kleinen, strukturierten VTK-Beispieldatensatzes *Kitchen* (vgl. Anhang A) benötigt eine Onyx-2 (vgl. Anhang B) für den sequentiellen Zugriff im Durchschnitt 1,17 Sekunden. Greifen zwei Prozesse gleichzeitig auf den Datensatz zu, steigen die Zugriffszeiten pro Prozess auf Werte zwischen 1,22 und 1,34 Sekunden. Dieses Verhalten verschlimmert sich mit zunehmender Prozessanzahl. Bei 16 gleichzeitig zugreifenden Berechnungsknoten schwanken die üblichen Zugriffszeiten zwischen 3,9 und 32,3 Sekunden; gemessene Spitzenwerte lagen sogar bei über 45 Sekunden. D. h., selbst wenn

Worker genügend Speicher haben, kann das parallele Laden von Daten schnell ineffizient werden und die Berechnung verlangsamen.

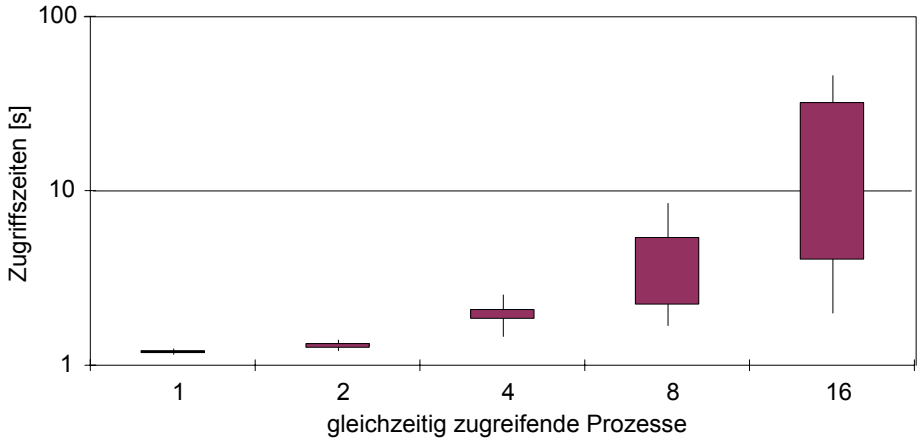

Abbildung 22: Zugriffszeiten mehrerer Prozesse einer SGI Onyx-2 auf dieselbe Datei

Sollen Datensätze bearbeitet werden, die nicht vollständig in den Speicher passen, müssen diese nicht unbedingt zuerst in einem Vorverarbeitungsschritt partitioniert werden. Dies betrifft vor allem unstrukturierte Gitter, die zunehmend für die Vernetzung komplexer Strömungsfelder verwendet werden, aber üblicherweise als eine einzige zusammenhängende Datenstruktur pro Zeitschritt vorliegen. Algorithmen, die gezielt Datenfragmente aus einem kompakten Datensatz in den Hauptspeicher laden, werden allgemein als *Out-of-Core*-Algorithmen bezeichnet. Neben der eigentlichen Datengröße spielen die Anzahl von Datenteilen, die in den Hauptspeicher passen, und die Anzahl von Datenteilen, die in einem Festplattenspeicherblock abgelegt sind, eine wichtige Rolle. Der letzte Aspekt ist deswegen wichtig, da bei der Anforderung eines einzigen Datums immer der komplette Festplattenblock geladen werden muss, in dem es liegt. Die beschriebene Korrelation wird durch so genannte *External-Memory*-Algorithmen berücksichtigt [VITT01].
Mit den schnell zunehmenden Größen von Simulationsrohdaten wächst die Bedeutung der *Out-of-Core*-Strategien auch für das Postprocessing stetig. Ein Überblick der verschiedenen hierbei verwendeten Ansätze wurde von Silva et al. in [SILV02] zusammengetragen.

3.4.4 Parallele Stromlinienberechnung

In der numerischen Integration werden Partikelpositionen, die als Stützpunkte einer Stromlinie dienen, hintereinander berechnet. Da jedoch für einen gegebenen diskreten Zeitschritt meistens mehrere Stromlinien gleichzeitig ermittelt werden sollen, bietet es sich an die zu berechnenden Stromlinien unter den *Workern* aufzuteilen. Somit liegt nach der Unterteilung, wie sie in Abschnitt 2.1 vorgenommen wurde, Berechnungsparallelität vor. Jeder dieser Berechnungsknoten erhält beispielsweise eine gleiche Anzahl von Saatpunkten zugewiesen, die nun nacheinander zur Stromlinienberechnung verwendet werden. Zum Einsammeln der einzelnen Stromlinien steht dann wieder das *Master-Worker*-Konzept zur Verfügung.

3.4.4.1 Laufzeitkomponenten von Viracocha

Die Stromlinienberechnung lässt sich auch einsetzen um die unterschiedlichen disjunkten Bestandteile einer Messung, die während der Laufzeit des parallelen Postprocessings mit Viracocha auftreten, zu bestimmen.
Für die Untersuchung dieser einzelnen Laufzeiten auf das Gesamtlaufzeitverhalten wurde ein kleiner rectilinearer Datensatz verwendet, der eine geschlossene Innenströmung in einer Küche simuliert. Dadurch lassen sich die speziellen Probleme, die mit der Behandlung von partitionierten Datensätzen verknüpft sind, ausblenden. Zudem wurde die Anzahl der zu

berechnenden Stromlinien recht hoch gewählt um Lastschwankungen während der Berechnung auszugleichen und somit akkurate Aussagen über die Interdependenz der verschiedenen Komponenten treffen zu können. Eine Beispielberechnung unter Verwendung des Beispieldatensatzes *Kitchen* ist in Abbildung 23 dargestellt.

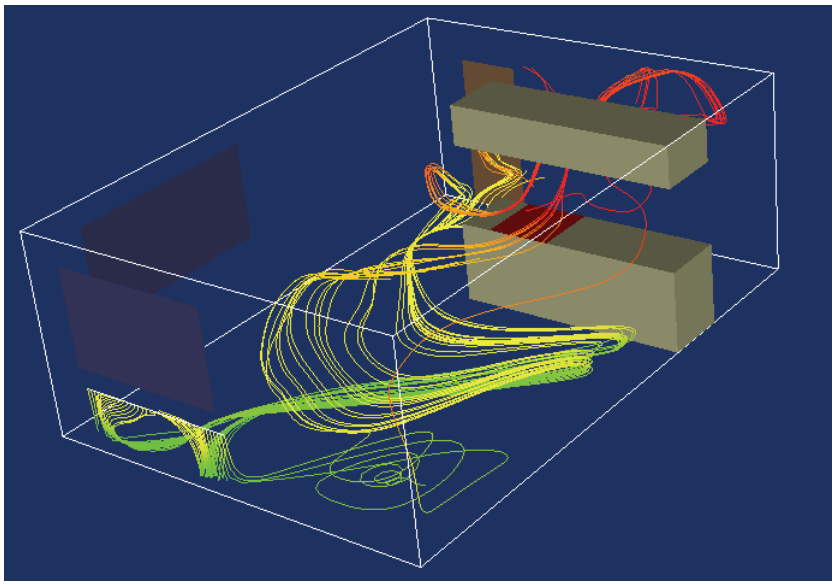

Abbildung 23: Luftzirkulation innerhalb einer Küche

Mit dem beschriebenen Testszenario wurden unterschiedliche Rechnersysteme unter Verwendung von Viracocha untersucht [GERN00]. Als Visualisierungsrechner kam primär eine SGI-Onyx zum Einsatz. Eine hpcLine von Fujitsu-Siemens, die über MPICH angebunden wurde, übernahm die parallele Stromlinienberechnung. Zum Vergleich wurden auch andere Kombinationen eingesetzt, wie zum Beispiel die jeweils wesentlich effizienteren nativen MPI-Versionen sowohl auf der Onyx, die dann auch die Berechnung übernahm, als auch auf der hpcLine (nun mit Visualisierung). Abgerundet wurde das Testfeld mit einem Intergraph-PC, der jedoch lediglich den sequentiellen Berechnungsfall abdeckte. Leistungsdaten der einzelnen Plattformen sind im Anhang B zusammengefasst.

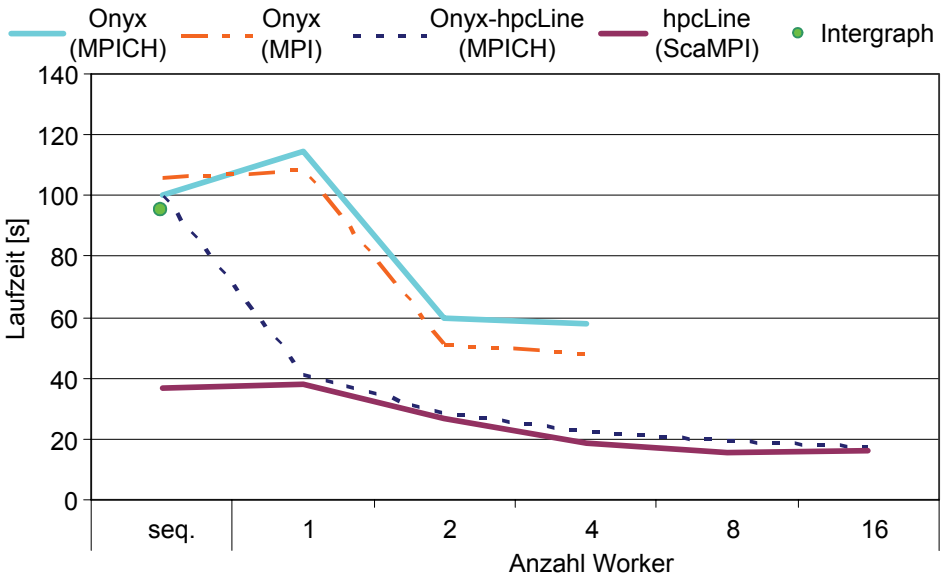

Abbildung 24: Laufzeiten für die Berechnung von 1800 Stromlinien

Ein Übersichtsergebnis für die Berechnung von 1800 Stromlinien ist in Abbildung 24 dargestellt. Wegen der deutlich höheren Prozessorgeschwindigkeiten berechnet der Linux-Cluster die Stromlinien wesentlich schneller als die SGI Onyx. Des Weiteren lässt sich für

alle erfassten Messungen, in denen mehrere *Worker* zum Einsatz kommen, Berechnungsbeschleunigungen feststellen. Wird lediglich ein *Worker* eingesetzt, kommt gegenüber dem sequentiellen Fall noch ein Offset für die anfallende Kommunikation mit *Scheduler* und *VisHost* hinzu.
Von besonderem Interesse ist das Laufzeitverhalten der Variante, in der sich die Onyx und die hpcLine die Arbeit aufteilen. Vom Absetzen des Berechnungsauftrags bis zur Visualisierung lassen sich grob fünf Abschnitte in der Berechnungspipeline identifizieren:

1. Einladen des CFD-Datensatzes
2. Berechnung der Stromlinien
3. Kommunikation zwischen den Komponenten des *WorkHosts*
4. Kommunikation zwischen *VisHost* und *WorkHost*
5. Aufbereiten der Daten für die Visualisierung

Die jeweiligen Anteile an der Gesamtlaufzeit sind in Abbildung 25 aufgetragen. Während die ersten drei Abschnitte parallel auf dem *WorkHost* ablaufen, sind die beiden letzteren serielle Pipelineschritte. Da jeder *Worker* jedoch die gleichen Daten laden muss, ist durch den ersten aufgelisteten Punkt (*Read*) kein *Speed-up* zu erwarten. Hingegen sollte der *Master-Worker*-Ansatz eine gute Effizienz beim reinen Berechnungsanteil (*Calc*) aufweisen. Beide Annahmen werden durch die Messungen bestätigt.

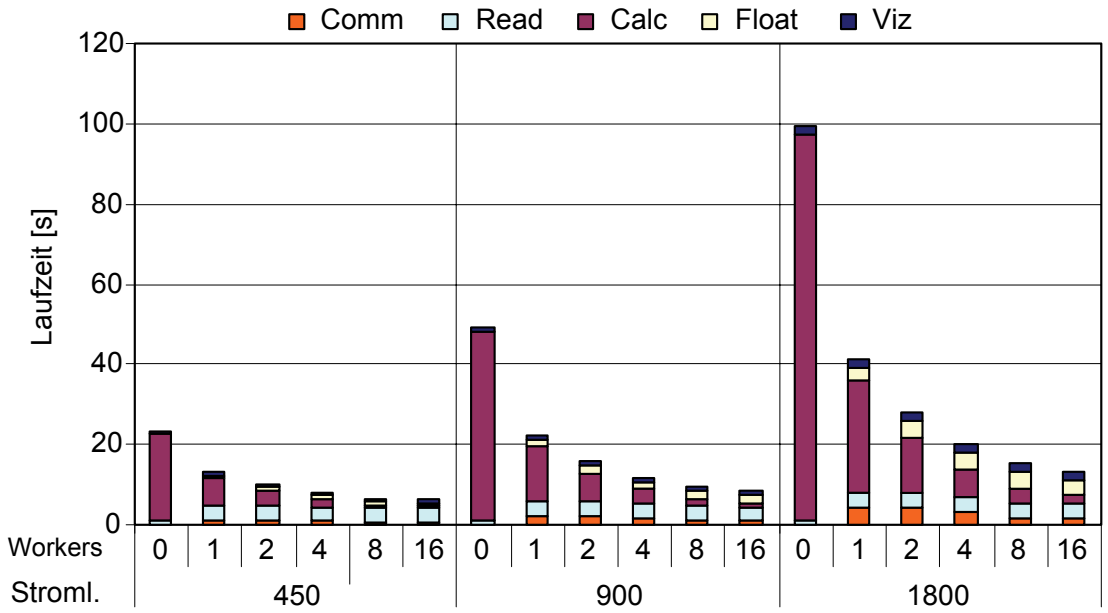

Abbildung 25: Kopplung von Onyx und hpcLine über MPICH

In der *Worker*-Spalte 0 ist der Fall aufgezeigt, in dem die Onyx die ganze Verarbeitungslast trägt. Erkennbar ist, dass sie durch eine bessere Anbindung an das Dateisystem zwar die Daten schneller laden kann, jedoch wegen der deutlich geringer getakteten Prozessoren wesentlich länger für die Extraktion benötigt.
In den anderen Spalten kommen die Zeiten für Kommunikation zwischen *Workern* (*Comm*) und die Kommunikation zwischen *VisHost* und *WorkHost* (*Float*) hinzu. Die wesentliche Last für *Float* entsteht dabei durch das Serialisieren, Versenden und Deserialisieren der Ergebnisdaten als Datenstrom von Fließkommazahlen. Erwartungsgemäß wächst dieser Anteil mit zunehmender Größe der Ergebnisdaten, bleibt aber in Relation zur *Worker*-Anzahl in etwa konstant. Die abschließende Datenaufbereitung auf der Onyx für die Visualisierung wird als *Viz* erfasst und ist ebenfalls abhängig von der Ergebnisgröße.
Neben der Serialisierung der Daten, die den Parallelisierungsgewinn schmälert, kommt noch ein Faktor hinzu, der im Diagramm als *Comm* bezeichnet wird. Hierunter subsumieren sich Zeiten, die durch Versenden und Zusammenfügen während der Datenaggregation anfallen.

Nicht zu vernachlässigen ist dabei die hier ebenfalls erfasste Zeit, in der ein *Worker* auf einen anderen warten muss, da der Reduktionsschritt vom *Master-Worker* nur sequentiell abgearbeitet werden kann.
Da zwei heterogene Systeme (Onyx und hpcLine) miteinander kommunizieren, kommt die plattformunabhängige *Message-Passing*-Variante MPICH zum Einsatz, die für den Datentransfer das TCP/IP-Protokoll über Fast Ethernet verwendet. Dieser Kommunikationsweg ist somit auch für den Datenaustausch zwischen den *Workern* vorgeschrieben. Würde man auf dem Linux-Cluster das schnellere ScaMPI einsetzen, ließe sich dieser Laufzeitanteil verringern. Die anderen Zeiten blieben davon unberührt. Bereits in Abbildung 24 ist ein leichter Vorteil der hpcLine im Standalone-Betrieb erkennbar. Die Aufsplittung der Einzelzeiten in Abbildung 26 erlaubt eine weitergehende Analyse. Tatsächlich konnte eine Verringerung der *WorkHost*-internen Kommunikation, die nun in *Reduce* und *Wait* unterteilt in das Diagramm eingetragen wurden, erreicht werden. Dieser Gewinn fällt jedoch bescheiden aus, da der *Wait*-Anteil zunehmend wächst, was auf Latenzen und Balancierungsprobleme hindeutet.

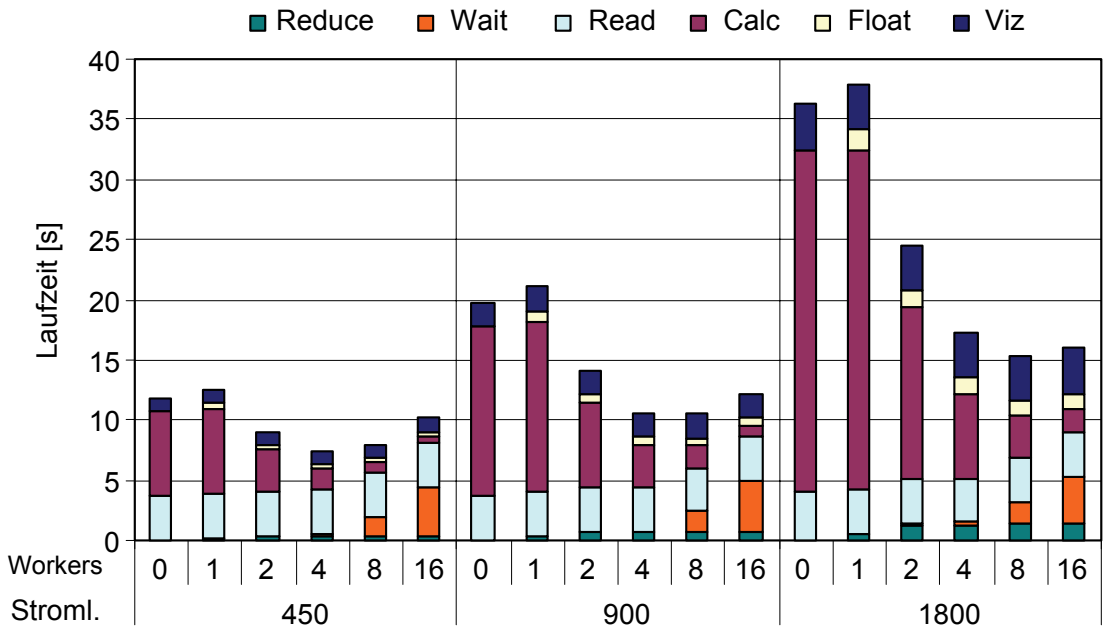

Abbildung 26: Parallelisierung auf der hpcLine mit ScaMPI

3.4.4.2 Die Verteilungsheuristik

Beim Küchendatensatz wurde davon ausgegangen, dass die Berechnungslast pro Stromlinie in etwa gleich ist. Bei vielen Strömungsdatensätzen kann jedoch nicht von solch einer Balanciertheit für das Postprocessing ausgegangen werden. Um eine ungleiche Verteilung der Extraktionslast zu untersuchen, wurde mit der in Abschnitt 3.4.4.1 beschriebenen Testumgebung der stationäre Nasendatensatz des Aerodynamischen Instituts der RWTH Aachen untersucht (siehe Abbildung 27) [GERN02]. Vollständig eingeladen, berechneten 1 bis 16 *Worker* 50 bzw. 100 Stromlinien. Für die Verteilung der Saatpunkte wurde eine Saatlinie definiert, auf der sich die einzelnen Saatpunkte äquidistant verteilten. Entsprechend der Anzahl der *Worker* wurde die Saatlinie sodann in Teilstücke zerlegt und den *Workern* zugewiesen.
Die Saatlinie wurde direkt am Eingang des zum Nasenloch führenden Einströmrohrs platziert. Zudem lag sie quer zur Hauptströmungsrichtung und parallel zur Nasenscheidewand. Ihre Länge entsprach dem Durchmesser des Querschnitts des anfänglich zylindrischen Einströmrohrs. Da es sich um eine Einatemsimulation handelt, konnte somit angenommen werden, dass sich die Stromlinien im gesamten Strömungsfeld entwickeln würden. Die Visualisierung der Naseninnenströmung durch die berechneten Stromlinien bestätigte diese Annahme.

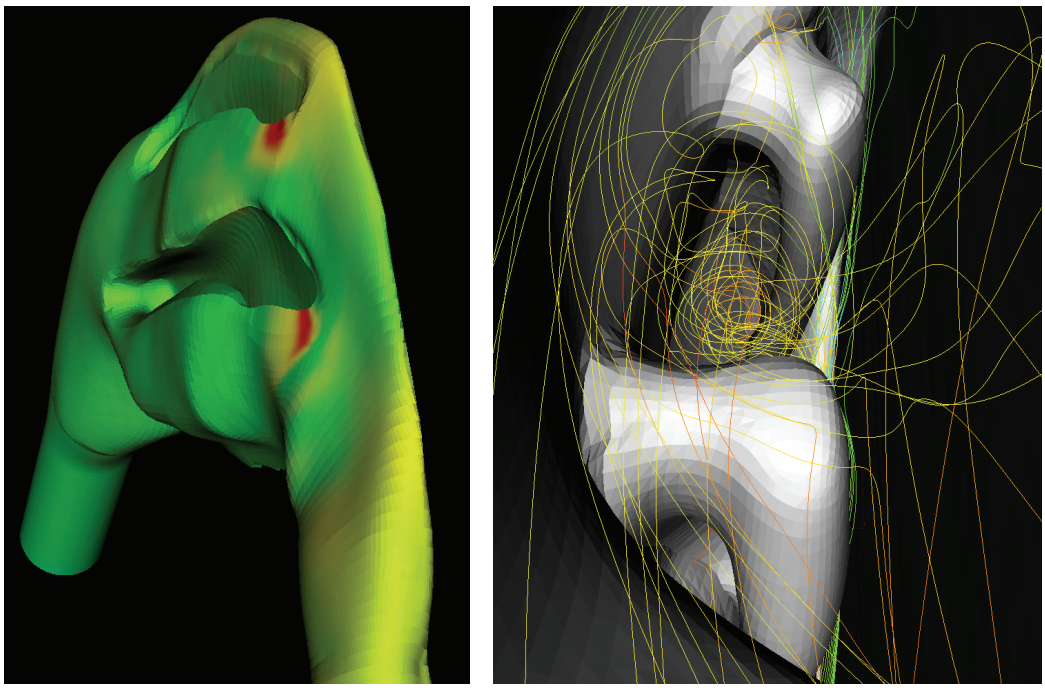

Abbildung 27: Nasendatensatz, Außenansicht (links) und Innenansicht mit Stromlinien (rechts)

Allerdings wurde, wie Abbildung 28 zeigt, für die reine Berechnungszeit nicht die erwartete Effizienz erreicht. Zudem traten auffällige Unregelmäßigkeiten bei der *WorkHost*-internen Kommunikation (*Comm*) auf. Die Erklärung findet sich in der ungleichmäßigen Lastverteilung unter den *Workern*. In Abbildung 29 sind die tatsächlich berechneten Datenmengen pro *Worker* aufgetragen. Es lässt sich erkennen, dass sich vor allem *Worker*, die vor der Mitte der Saatlinie mit Saatpunkten versorgt wurden, mit einer vielfachen Berechnungslast konfrontiert sehen.

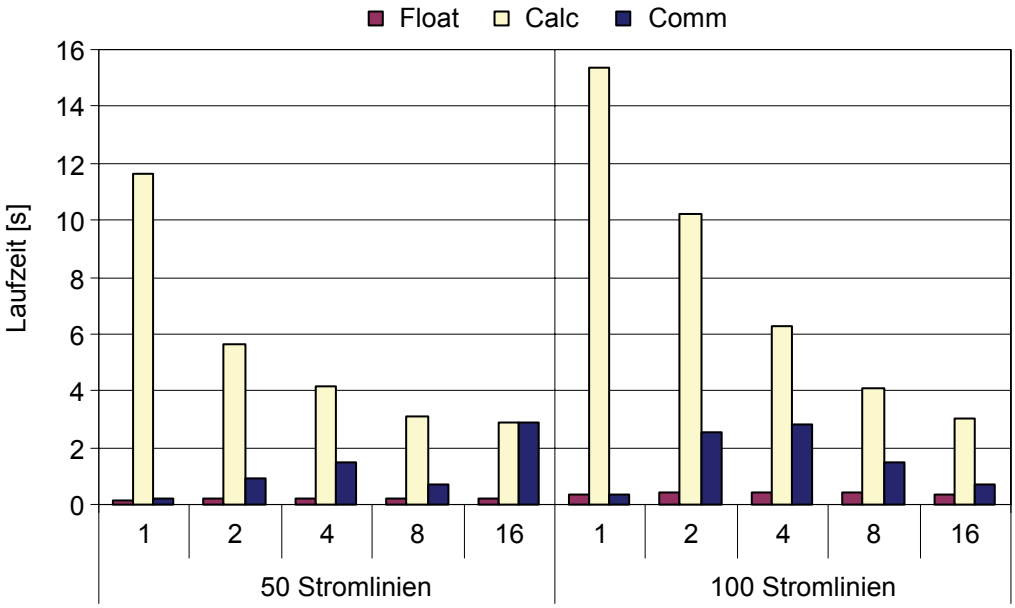

Abbildung 28: Laufzeiten zur Berechnung von Stromlinien auf der hpcLine (MPICH)

Die Ursache ist im speziellen Anwendungsfall begründet. Sobald Stromlinien am Rachen das Strömungsfeld der Nasenhöhle verlassen, wird ihre Berechnung beendet. Stromlinien, die mit hoher Geschwindigkeit und ohne auf große Hindernisse zu stoßen bis zum Rachen durchströmen, sind kurz und weisen dementsprechend wenige Stützpunkte auf. Aber gerade im oberen Bereich der Nasenhaupthöhle, wo sich eine der großen Nasenmuscheln befindet, verweilen die Stromlinien erheblich länger. Durch diese Strömungseigenschaft ist die Nase im

Übrigen überhaupt erst in der Lage ihre vielfältigen Aufgaben, wie Anwärmung, Anfeuchtung und Reinigung der Atemluft, wahrzunehmen [REIM01, GERN03].

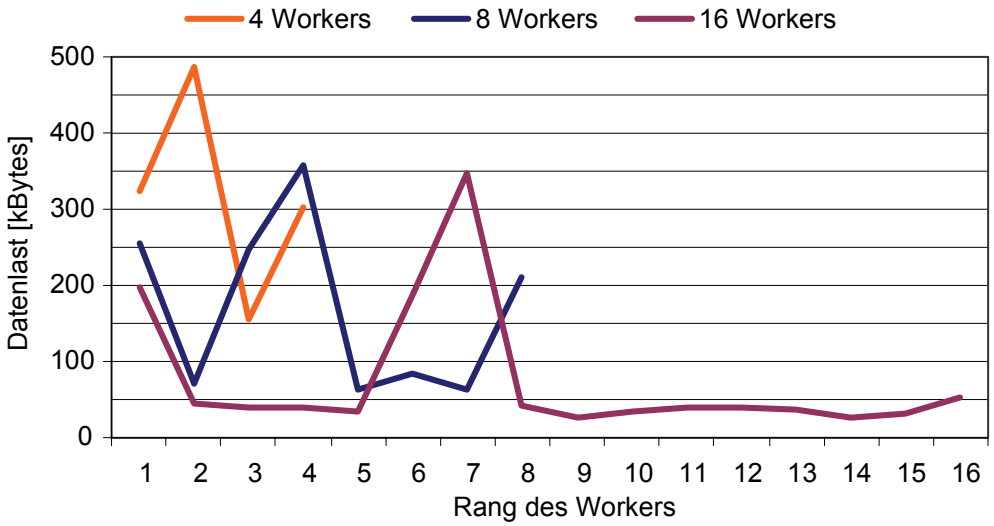

Abbildung 29: Anfallende Datenmengen für jeden einzelnen *Worker* bei der Gesamtberechnung von 50 Stromlinien

In der Regel weisen CFD-Datensätze eine ungleichmäßig verteilte Strömungsdynamik auf. Daher kommt es immer wieder zu ungleichmäßiger Berechnungslast. In allen hier durchgeführten Messungen wurde der *Master-Worker* vorher bestimmt, d. h. es wurde keine dynamische Zuweisung durch den *Scheduler* vorgenommen. Obwohl im ersten Diagramm der *Master-Worker* dennoch am wenigsten Zeit für die Berechnung braucht und somit ohne Verzögerung und parallel zur laufenden Berechnung anderer *Worker* die Teilergebnisse zusammenfügen kann, kommt es trotzdem zu Balancierungsproblemen. Der Grund liegt in der unterschiedlich langen Integration der beteiligten *Worker*. Während der *Master-Worker* ab acht *Workern* bereits weniger als eine Sekunde für die Stromlinienberechnung benötigt, läuft der Worker mit der längsten Berechnungszeit noch ganze 38 Sekunden. Einige exemplarische Messwerte sind in Tabelle 1 zusammengestellt.

Anzahl Prozesse	**1**	**2**	**3**	**8**	**16**	**21**
Master-Worker (am schnellsten)	128	17	2,5	0,75	0,51	0,48
Worker (längste Berechnung)	128	91	60	38	22	17
Gesamtlaufzeit [in Sekunden]	137	103	75	52	37	42

Tabelle 1: Vergleich *Master-Worker* und *Worker* mit längster Integrationszeit durch stückweise Verteilung der Saatlinie

Soll trotzdem eine statische Saatpunktverteilung vorgenommen werden, dann führt eine einfache Auffächerungsstrategie – hier als *Round Robin* bezeichnet – auch ohne Vorhersageansätze zu einer recht ausgeglichenen Arbeitslast. Dabei werden die Saatpunkte auf der Saatlinie jetzt der Reihe nach abwechselnd einem Berechnungsprozess zugewiesen. Durch diese Spreizung wird die Wahrscheinlichkeit größer, dass jeder *Worker* nun Stromlinien mit hoher und mit weniger großer Last berechnen muss. Die Zuordnung der Saatpunkte ist in Abbildung 30 dargestellt.
Mit dieser Verteilungsstrategie konnte tatsächlich eine Berechnungsbalancierung im Nasendatensatz erreicht werden. Dass der *Round-Robin*-Ansatz aber auch für Datensätze, in denen nicht direkt eine ungleiche Extraktionslast angenommen werden kann, sinnvoll ist, lässt sich anhand des turbulenten Einströmvorgangs im AIA-Motordatensatz belegen. Eine hohe Anzahl von Saatpunkten eliminiert wiederum sonstige störende Einflüsse und unterstreicht den Aussagewert der erzielten Ergebnisse.

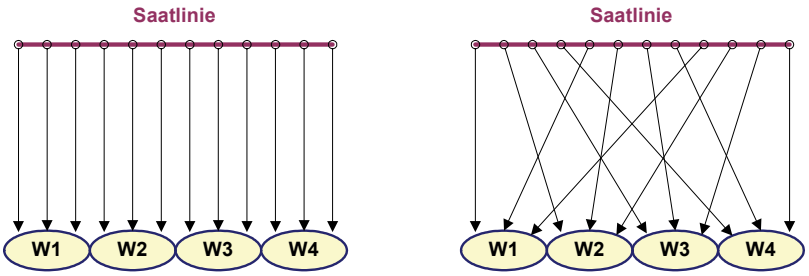

Abbildung 30: Gleichmäßige Saatpunktverteilung (links) und Verteilung per *Round Robin* (rechts) pro *Worker*

Eine erste Versuchsreihe im Zeitschritt 20 mit 5000 Saatpunkten lief auf der hpcLine[10] als Vertreter von *Distributed-Memory*-Clustern. Dass hier die Annahme einer besseren Skalierung per *Round Robin* tatsächlich zutrifft, kann beim Vergleich der beiden Diagramme in Abbildung 31 und Abbildung 32 erkannt werden. So erfordert z. B. der *Round-Robin*-Ansatz mit 6 *Workern* circa 40 Sekunden Laufzeit, während die stückweise Verteilung 60 Sekunden benötigt.

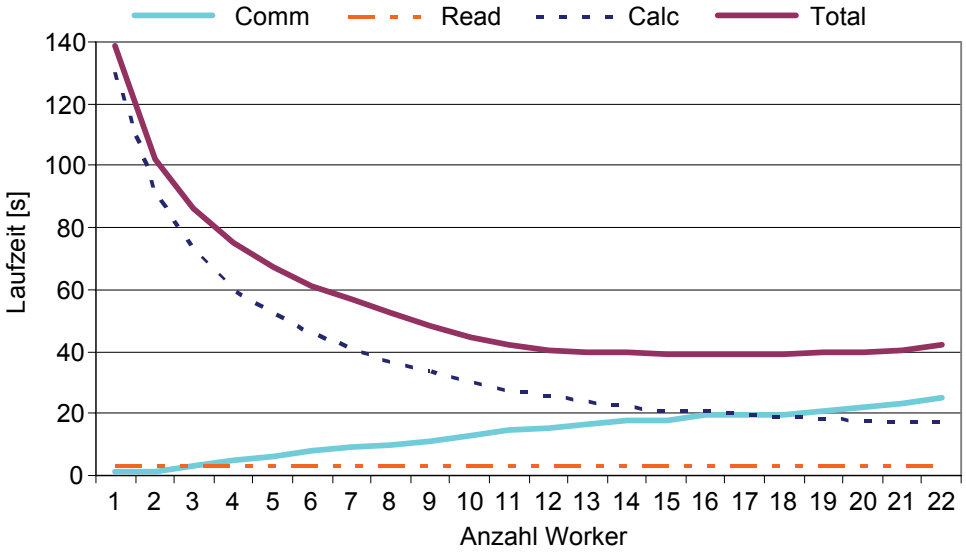

Abbildung 31: Stückweise Verteilung der Saatlinie, 5000 Saatpunkte, AIA-Motor, hpcLine

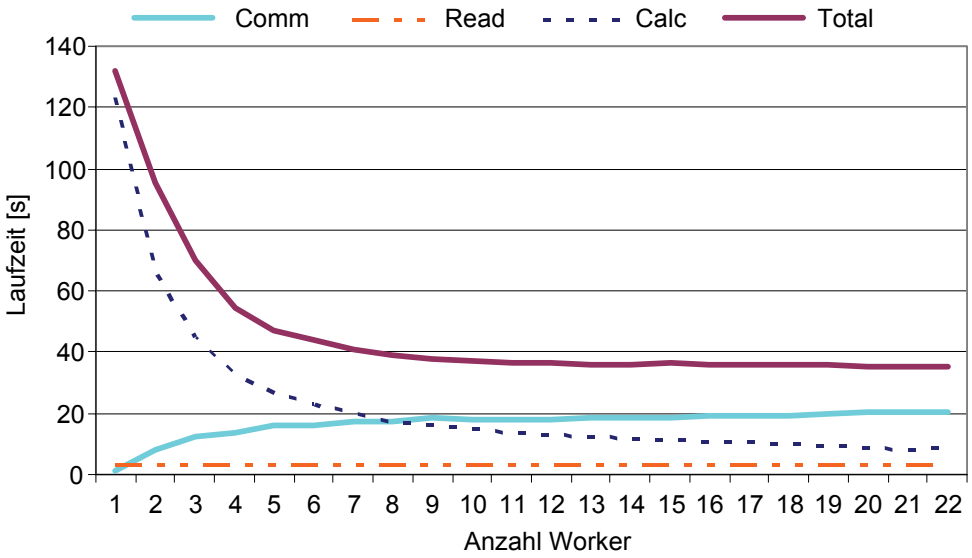

Abbildung 32: *Round-Robin*-Verteilung der 5000 Saatpunkte, AIA-Motor, hpcLine

[10] Hierbei handelte es sich um die mit 32 Pentium-III-800MHz-Prozessoren aufgerüstete Variante.

Neben der hpcLine kam in einer zweiten Messreihe diesmal auch der SunFire-Hochleistungsrechner zum Einsatz. Entgegen der hpcLine verwendet ein SunFire-Knoten einen großen gemeinsamen Hauptspeicher. Da bei der SunFire eine über *Shared Memory* kommunizierende MPI-Variante verwendet wird, sollte sich dies positiv auf die Datenkommunikation unter den *Workern* (als *Comm* erfasst) auswirken. Gemessen wurde wiederum die Berechnung von 5000 Stromlinien ohne und mit *Round-Robin*-Verteilung. Die Ergebnisse sind in Abbildung 33 und Abbildung 34 dargestellt.

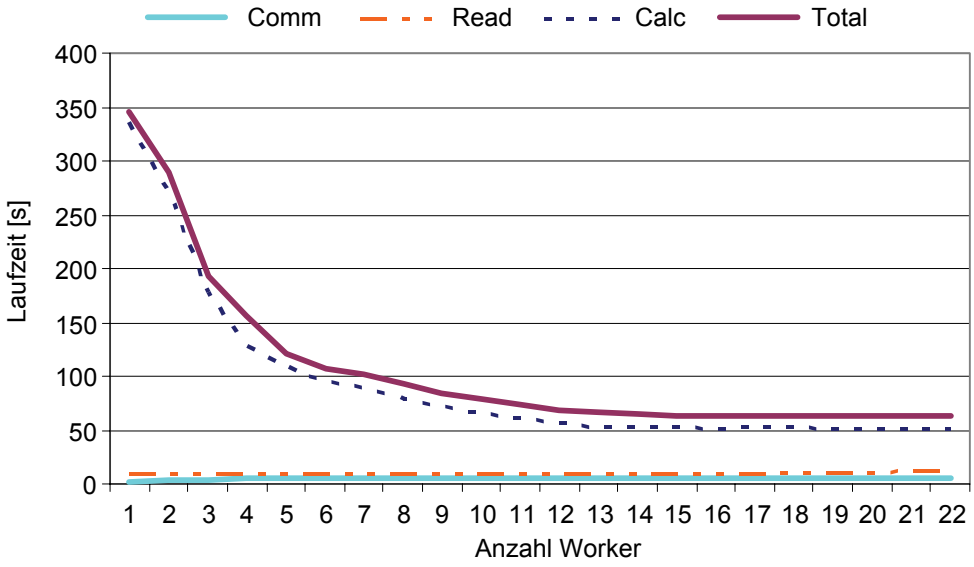

Abbildung 33: Stückweise Verteilung der Saatlinie, AIA-Motor, SunFire 6800

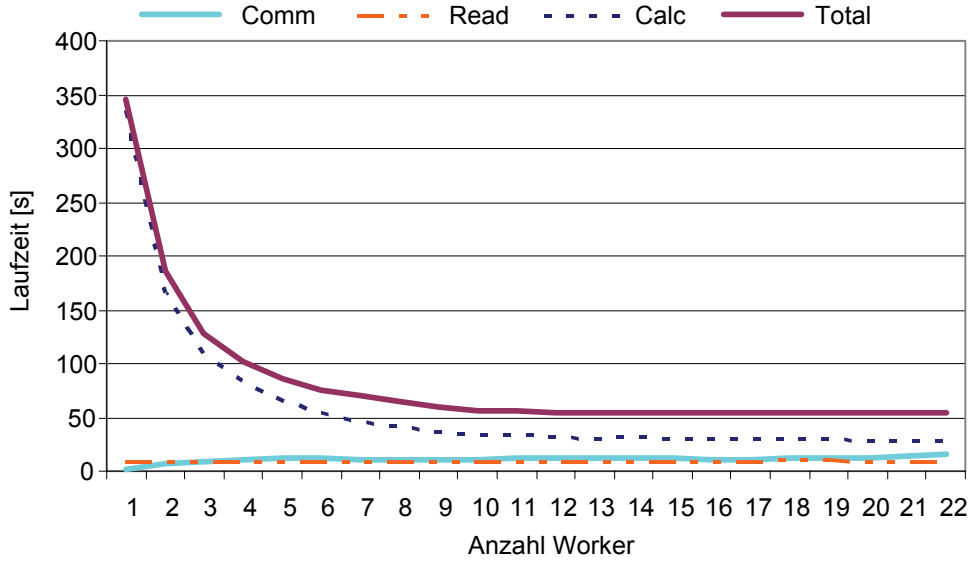

Abbildung 34: *Round-Robin*-Verteilung der Saatpunkte, AIA-Motor, SunFire 6800

Tatsächlich ist die *Worker*-Kommunikation nun weitestgehend gleich bleibend niedrig. Anders als bei der hpcLine hat die steigende Anzahl von *Workern* beim Empfang und Zusammenfügen der Teilergebnisse kaum noch Auswirkungen auf die gesamte Berechnungszeit. Hier zeigt sich, dass der Datenaustausch über *Shared Memory* wirklich erheblich schneller vonstatten geht.

Für eine genauere Bewertung der Leistungsfähigkeit der angewendeten Verteilungsheuristiken wurde der erreichte *Speed-up* berechnet. Die eigentlichen Integrationszeiten sind in Tabelle 2 und Tabelle 3 separat aufgelistet, da diese zum parallelen Programmanteil zugeordnet werden. In der Gesamtlaufzeit sind dann noch zusätzlich Ladezeiten und Kommunikation aufaddiert.

Anzahl Prozesse	1	2	4	8	16	22
Integrationszeit	336	271	145	83	53	42
Speed-up (Integration)	-	1,23	2,32	4,05	6,34	8,0
Gesamtlaufzeit	346	281	155	93	65	65
Speed-up (Gesamt)	-	1,23	2,23	3,72	5,32	5,32
Sequentieller Anteil	2,9 %	3,55 %	6,9 %	12 %	18,9 %	35,4 %

Tabelle 2: Erreichter *Speed-up* auf einer SunFire-6800, normale Saatpunktverteilung

Anzahl Prozesse	1	2	4	8	16	22
Integrationszeit	336	169	85	43	28	20
Speed-up (Integration)	-	1,99	3,95	7,81	12,0	16,8
Gesamtlaufzeit	346	180	101	64	54	54
Speed-up (Gesamt)	-	1,92	3,45	5,41	6,4	6,4
Sequentieller Anteil	2,9 %	6,1 %	15,8 %	24,7 %	48,1 %	62,9 %

Tabelle 3: Erreichter *Speed-up* auf SunFire-6800, *Round-Robin*-Saatpunktverteilung

Vergleicht man dagegen die *Speed-ups* der reinen Integrationszeit zwischen der normalen Saatpunkt-Verteilung (Tabelle 2) und dem *Round-Robin*-Ansatz (Tabelle 3), dann fällt sofort auf, dass der zweite Ansatz wesentlich effektiver ist.
Es lässt sich zudem feststellen, dass ab einem gewissen Punkt keine weitere Steigerung des *Speed-ups* (Gesamt) möglich ist. Allerdings erreicht der zweite Ansatz diese Sättigungsmarke mit 12 gegenüber 16 Prozessoren wesentlich früher und liegt dann mit 54 Sekunden Gesamtberechnungsdauer über 10 Sekunden unter der Bestmarke von 65 Sekunden, die der erste Ansatz erreicht.

3.4.5 Bahnlinienberechnung in Multi-Block-Datensätzen

Ist es noch vertretbar, bei kleineren MB-Datensätzen den benötigten Zeitschritt zur Stromlinienextraktion im Vorfeld komplett zu laden, würde dies einen erheblichen Aufwand bei der instationären Variante der Partikelverfolgung mittels Bahnlinienberechnung bedeuten. Hier müssten dann alle berücksichtigten Zeitschritte auf einmal in den Hauptspeicher geladen werden, obwohl üblicherweise nur ein sehr geringer Teil der eingeladenen Datenmengen tatsächlich benötigt wird.
Um das dynamische Nachladen von Daten zu bewerten, kam in einem weiteren Experiment wiederum der Multi-Block-Motordatensatz vom AIA zum Einsatz. Implementiert und evaluiert wurden unterschiedliche Integrationsverfahren für Partikelverfolgung. Als besonders effizient hat sich die adaptive Schrittweitenkontrolle nach Weller [WELL96] herausgestellt, die während der Integration mithilfe des Runge-Kutta-Verfahrens 4. Ordnung dessen einzelne Teilschritte für eine Fehlerabschätzung auswertet [GERN05].
Für das Laden benötigter Dateien kam eine Strategie zum Einsatz, die automatisch Datenblöcke nachlädt, wenn ein Algorithmus darin gespeicherte Datenzellen benötigt. Aus der Sicht des Algorithmus geschieht das Nachladen unbemerkt. Besonders wichtig ist hierbei, dass beim Verlassen einer Bahnlinie mithilfe der gespeicherten MB-Topologieinformationen neben der Identifizierung des Nachbarblocks zudem die potentiell betretene Zelle vorhergesagt werden kann. Dies reduziert den erheblichen Zeitaufwand für eine Zellsuche, wie er von Sadarjoen et al. in [SADA97] beschrieben wurde, deutlich.
Es wurden jeweils 8 Bahnlinien unter Verwendung unterschiedlicher Integrationsverfahren berechnet. Die Saatlinie war jeweils identisch und befand sich leicht unterhalb eines Einlassventils. Integriert wurde von Zeitschritt 14 bis 23. Die erzielten Ergebnisse sahen in Abhängigkeit von Integrationsverfahren und Integrationsparameter immer annähernd so aus wie in Abbildung 35 dargestellt.

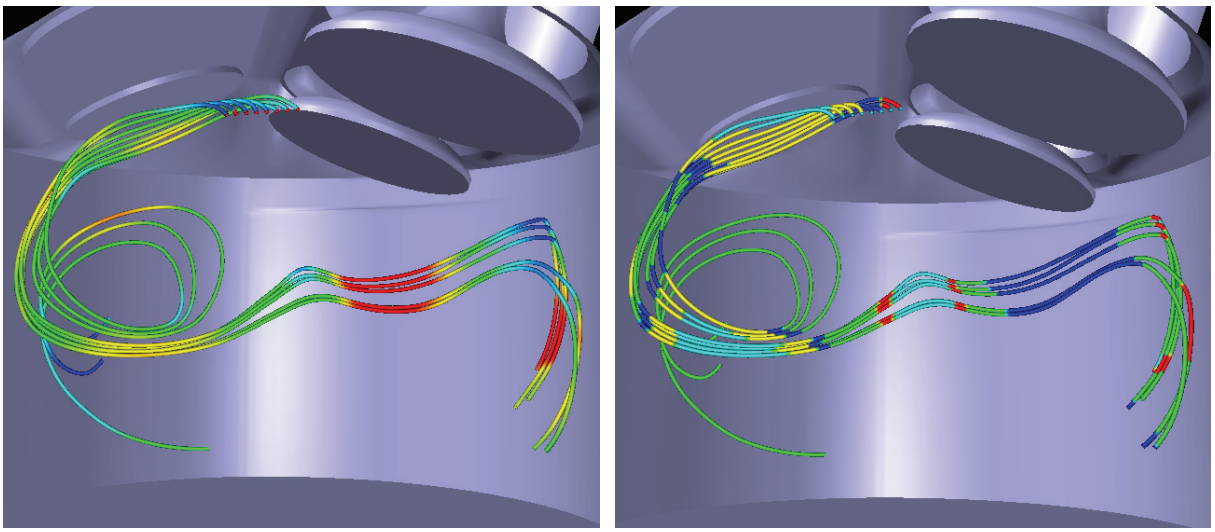

Abbildung 35: Bahnlinienberechnung von Zeitschritt 14 bis 23, Skalar-Mapping (links) und Farbkodierung der Verweilzeit in einem Einzelblock (rechts)

Im sequentiellen Fall wird eine Bahnlinie nach der anderen berechnet. Es wird mit dem ersten Saatpunkt begonnen und der Block, der diesen Saatpunkt enthält, geladen. Verlässt die Bahnlinie den aktuellen Block, wird derjenige Block nachgeladen, der nun vermutlich die aktuelle Partikelposition beinhaltet. Es kann in Sonderfällen vorkommen, dass die Annahme falsch ist und dann noch alternative Blöcke eingeladen werden müssen, um die Integration fortsetzen zu können. Ist man am Ende der Integrationszeit angelangt, beginnt der *Worker* mit der zweiten Bahnlinie. Die Wahrscheinlichkeit ist hoch, dass sie ebenfalls im gleichen Startblock wie die erste Bahnlinie beginnt. Nur wenn der Verlauf der zweiten Bahnlinie Blöcke durchläuft, die die erste nicht bereits geladen hat, wird dieser nachgeladen. Diese Prozedur gilt für alle nachfolgenden Bahnlinien. Je später eine Bahnlinie berechnet wird, umso höher ist die Wahrscheinlichkeit, dass sich der Datenblock schon im Speicher befindet. Das hat zur Folge, dass die Zeit, die für das Laden aufgebracht werden muss, deutlich geringer ausfällt. Diesen Sachverhalt der Ladelast verdeutlicht in Abbildung 36 die Kurve für einen *Worker*.

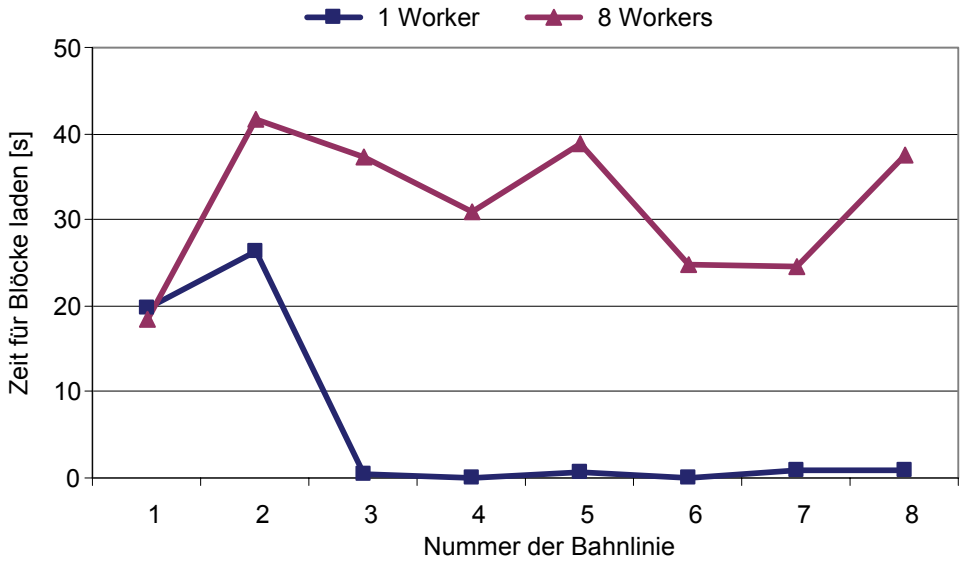

Abbildung 36: Benötigte Zeit zum Laden der angeforderten Blöcke beim Einsatz eines *Workers* bzw. von 8 *Workern*, wenn jeweils 8 Bahnlinien hintereinander berechnet werden

Werden jedoch 8 Bahnlinien von acht unabhängigen *Workern* bearbeitet, muss jeder *Worker* für die ihm zugeteilte Bahnlinie sämtliche Datenblöcke laden. Hier kann man keinen

Caching-Effekt erwarten. Die entstehende Last pro *Worker* ist ebenfalls in Abbildung 36 als zweite Kurve (8 *Workers*) aufgetragen.
Hier muss jeder *Worker* das notwendige Datenladen selbst übernehmen. Zwar wird diese Belastung durch Parallelisierung auf mehrere Prozesse verteilt, durch unterschiedliche Blockgrößen kommt es jedoch zu starken Balancierungsproblemen. Dass sich dies im vorliegenden Fall so negativ auswirkt, dass sich anfänglich kaum eine Lastverteilung beim Laden (*Load*) durch Parallelisierung erreichen lässt, ist im linken Diagramm der Abbildung 37 erkennbar. Bei 8 *Workern* tritt dann zwar die erhoffte Lastverringerung beim Laden auf, nun kommt es jedoch beim Anwenden des *Master-Worker*-Konzepts zu Balancierungsproblemen, sodass die Zeit für Kommunikation und Datenzusammenführung (*WorkHost Comm*) in die Höhe schnellt. Somit geht die eine Einsparung zu Lasten einer anderen Bearbeitungsgröße. Werden alle benötigten Zeitschritte im Voraus in den Hauptspeicher geladen, zeigt sich, dass die Berechnung und die Kommunikation innerhalb der *Work Group* durchaus balanciert vonstatten gehen (rechtes Diagramm, Abbildung 37).

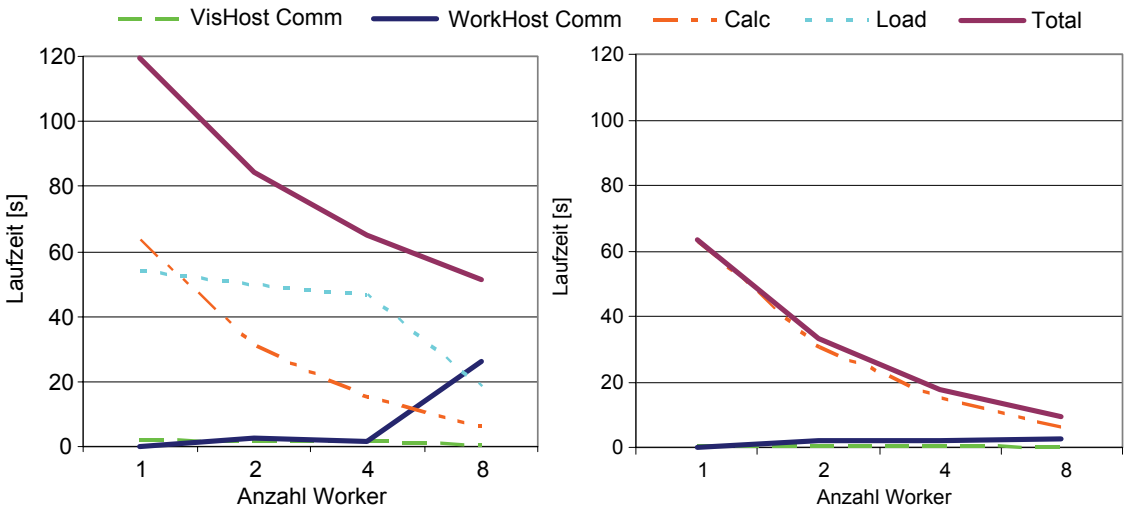

Abbildung 37: Benötigte Berechnungszeit, ohne (links) und mit (rechts) vorgeladenen Daten

In Abbildung 38 ist die Effizienz der Berechnung mit vorgeladenen Daten abgebildet. Mit Werten über 0,8 kann von einer guten Skalierbarkeit des angewendeten Bahnlinienalgorithmus gesprochen werden. Kommt jedoch das Laden-bei-Bedarf hinzu, bricht die Effizienz des Extraktionsverfahrens mit einem Wert unter 0,3 deutlich ein.

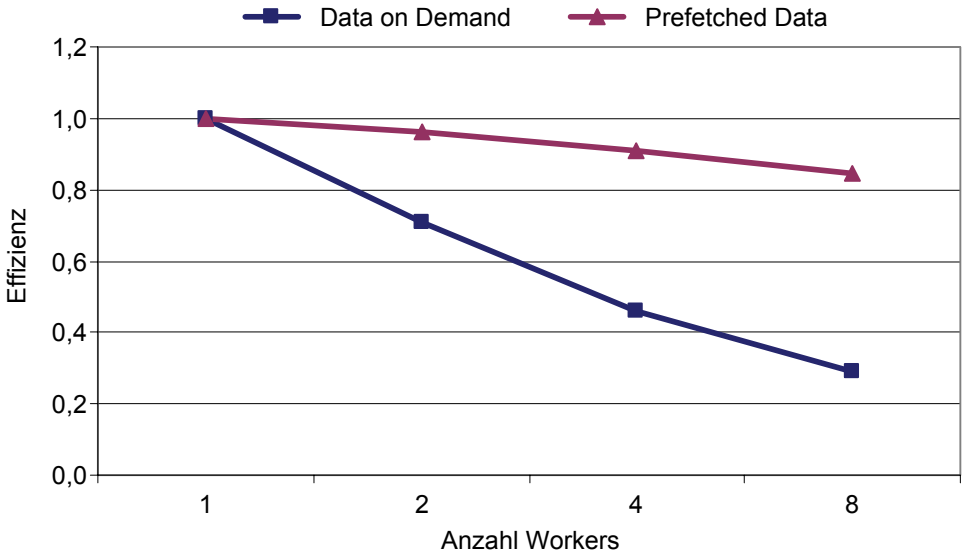

Abbildung 38: Effizienz unterschiedlicher Ladestrategien für die Bahnlinienberechnung

In der Prefetching-Messung wurden die kompletten Blöcke aller benötigten Zeitschritte lediglich zur Validierung des eigentlichen parallelen Integrationsansatzes im Vorfeld eingelesen. Natürlich wird dadurch die nicht unerhebliche Zeit, die hierfür benötigt wurde, schlicht ignoriert (vgl. auch Abbildung 22). Werden wirklich große Datensätze eingesetzt, kommt dieser Ansatz schon alleine wegen des unzureichenden Hauptspeichers nicht mehr in Frage. Daher ist es grundsätzlich der zu wählende Weg, Daten erst bei Bedarf wirklich in den Speicher zu holen. Aber trotz integriertem Multi-Block-Handling macht das Datenladen in der Regel die Effizienz, die durch optimiert parallelisierte Extraktionsalgorithmen erreicht werden kann, wieder zunichte. Dies war die wesentliche Motivation zur Entwicklung des in Viracocha integrierten Datenmanagements, das im Kapitel 4 ausführlich dargestellt wird.

3.4.6 Isoflächen und Wirbelregionen

Der von Lorensen und Cline als *Marching Cubes* [LORE87] vorgestellte Ansatz zur Isoflächenextraktion, entwickelt für reguläre Gitter, untersucht jede einzelne Zelle eines Datensatzes, ob sie aktiv ist. Bei n Zellen liegt somit die Zeitkomplexität bei $O(n)$. Der originale Algorithmus beinhaltet allerdings das Problem, dass nicht jede aktive Zelle eindeutige Konturen aufweist. Es kann zu Löchern (engl.: *Cracks*) in eigentlich geschlossenen Isoflächen kommen. Nielson und Hamann haben festgestellt, dass hiervon ausschließlich Zellen betroffen sind, die mindestens eine Seite aufweisen, auf der alle vier Kanten vom Isowert geschnitten werden. Zur Vermeidung von Löchern schlagen die Autoren in [NIEL91] vor, mithilfe der Bestimmung der Asymptoten der bilinearen Interpolationsfunktion, die sich für die jeweilige Zellenseite definieren lässt, eine eindeutige Triangulierung vorzunehmen. 2003 stellte Nielson in [NIEL03] seine Erweiterung auf die trilineare Interpolation vor.
Eine Isofläche kann somit alleine durch die isolierte Betrachtung der Einzelzellen eines Datensatzes bestimmt werden. Reine zellbasierte Ansätze lassen sich grundsätzlich sehr einfach durch Zerteilung der Datensätze und Zuweisung von Teildaten auf verschiedene Prozesse parallelisieren. Es müssen dann in einem abschließenden Schritt die parallel berechneten Teile der Isofläche wieder zusammengefügt werden. Diese Form der Parallelisierung ist sehr einfach mit dem Konzept des *Master-Workers* zu realisieren. Die einzelnen Datensatzblöcke werden auf die verschiedenen Knoten verteilt, die Isoflächen mit den Standardalgorithmen berechnet und dann zum *Master-Worker* geschickt. Dort werden alle Teilstücke zu einer einzigen Isoflächendatenstruktur zusammengesetzt.
Aus Zeitgründen entfällt hier jedoch üblicherweise das Entfernen doppelter Punkte an den Nahtstellen. Dies hat jedoch Einfluss auf die Qualität der Darstellung, denn üblicherweise werden Punktnormalen an den Eckpunkten aller Polygone berechnet, aus denen eine Isofläche besteht. Für einen dieser Punkte geschieht dies durch Mittelung der Flächennormalen aller Polygone, die ihn als gemeinsamen Punkt aufweisen. Durch die Interpolation der Punktnormalen während des Renderns kann eine gleichmäßige Beleuchtung erreicht und der diskrete Charakter des Ergebnisses verborgen werden. Teilen sich Polygone jedoch keine gemeinsamen Eckpunkte, werden sie nicht als zusammenhängende Oberflächenstruktur erkannt. Liegt ein größerer Krümmungswinkel zwischen diesen betroffenen Polygonen vor, sind deutlich Kanten wahrnehmbar.
Je mehr Prozesse parallel an einer Isofläche rechnen, umso mehr Kanten treten auf. Zudem können *Worker* zugewiesene Datenblöcke unterschiedlich verarbeiten. Zum einen lassen sich die einzelnen Blöcke vor der Isoflächenextraktion zu einem unstrukturierten Teildatensatz zusammenfügen, wobei wiederum auf die Eliminierung doppelter Punkte an den Blockgrenzen verzichtet wird. Die anschließende Isoflächenberechnung erzeugt dennoch eine korrekt verbundene Oberfläche. Das liegt daran, dass der Algorithmus beim Einfügen neuer Dreiecke zum Gesamtergebnis automatisch doppelte Isoflächenpunkte erkennt und Duplikate vermeidet.
Der andere Weg verwendet jeden Block direkt als Eingabedatum für die Isoflächenberechnung. Vor dem Verschicken zum *Master-Worker* müssen dann alle

Teilflächen zu einem kompakten Datenpaket zusammenfügt werden. Dabei werden allerdings lediglich Datenpakete hintereinander gehängt. Eine Entfernung der doppelten Kantenpunkte geschieht dadurch jedoch nicht.[11]
Das hier beschriebene Problem, dass Isoflächenfragmente ggf. keine topologische Beziehung zu Nachbarfragmenten aufweisen, macht sich nicht nur in der Darstellung von Kantenartefakten bemerkbar. Vielmehr wird die Datenstruktur aufgebläht und deren Optimierung wird verhindert, sodass die Renderleistung deutlich einbricht. Dies ist ein durchaus ernsthaftes Problem für die in Kapitel 5 beschriebenen Streamingansätze, wenn diese lediglich pro Zelle Isoflächenfragmente extrahieren und zum Visualisierungsrechner übertragen.
Die Behandlung von Wirbelregionen stellt sich für das Postprocessing weitestgehend identisch zur Isoflächenbehandlung dar. Der einzige Unterschied ist, dass das Skalarfeld, das die Werte für die Wirbelidentifikation beinhaltet, erst noch berechnet werden muss, wobei diese vorgeschaltete Verarbeitungsstufe recht zeitintensiv sein kann. Nach der Vorstufe lassen sich Wirbelregionen aber üblicherweise einfach als Isoflächen mit einem für das Wirbelverfahren angepassten Isowert extrahieren. Da die Berechnung der Quantität, die für die Wirbelbestimmung herangezogen wird, üblicherweise lokal in einer Zelle durchgeführt wird, sind somit Parallelisierungsansätze, wie sie bei der Isoflächenextraktion beschrieben wurden, anwendbar. Einen Überblick über die wichtigsten Extraktionsverfahren von Wirbelregionen liefert Martin Roth in [ROTH00].

3.4.7 Kritische Punkte

Das Suchen und Identifizieren von kritischen Punkten ist ebenfalls ein zellbasierter Ansatz [GLOB91]. Werden die Geschwindigkeitsgradienten mit gemischten Differenzen [SADA97] bestimmt, ist somit keine Nachbarschaftsinformation vonnöten. Dann lassen sich die Datenblöcke wie für die Parallelisierung der Isoflächenberechnung auf einzelne *Worker* verteilen. Darüber hinaus kann der Berechnungsalgorithmus sehr einfach weiter skaliert werden, indem auch innerhalb der Datenblöcke die Zellen unter *Workern* aufgeteilt werden. Probleme mit Nahtstellen, wie sie bei Isoflächen zu berücksichtigen sind, treten hier nicht auf.
Exemplarisch wurde die Extraktion von Topologieinformationen eines Vektorfelds am Nasendatensatz evaluiert [GERN04a]. Im Vordergrund stand hier der Vergleich von experimentell erfassten Strömungsergebnissen mithilfe der PIV-Analyse (*Particle Image Velocimetry*, PIV) [GHAR00, KONR01, BRUE00] und numerischen Simulationsergebnissen. In Analogie zum PIV-Experiment werden parallel zum Septum des Nasenmodells Schnitte gelegt, auf denen das Geschwindigkeitsfeld projiziert wird. Zur Vereinfachung der nachfolgenden Berechnungsschritte wird das auf diese Weise erzeugte zweidimensionale Strömungsfeld trianguliert.
Auch der Nasendatensatz ist ein Multi-Block-Gitter (siehe Anhang A). Somit lassen sich bereits Schnittberechnung und Triangulierung auf einfache Weise mit dem *Master-Worker*-Konzept parallel durchführen. Die im nächsten Bearbeitungsschritt anstehende Berechnung der kritischen Punkte kann direkt von den einzelnen *Workern* auf von ihnen berechneten Schnittteilen angeschlossen werden. Zur Bestimmung der Zellen, die einen kritischen Punkt beinhalten, kommt das Poincaré-Index-Verfahren zum Einsatz [TRIC01]. Die kritischen Punkte lassen sich dann direkt mithilfe der baryzentrischen Koordinaten bestimmen (vgl. [SADA94]). Ein Ausschnitt des Ergebnisses der Suche von Zellen, die einen kritischen Punkt beinhalten, ist in Abbildung 39 (links) dargestellt.
Direkt bei der Zellsuche können anhand des Poincaré-Index solche Zellen herausgefiltert werden, die einen Sattelpunkt beinhalten. Für die Extraktion des Topologiegraphens werden ausschließlich solche Punkte verwendet. Man berechnet das Eigenwertsystem des Geschwindigkeitsgradienten an dieser Position und verwendet die Eigenvektoren für die

[11] Die hier beschriebenen MB-Ansätze lassen sich auch auf die Extraktion von Schnittflächen übertragen.

Integration der Separationslinien [HELM91, LEEU99, LEEU00]. Ab hier reichen ggf. die Schnittdaten, die lokal bei einem *Worker* vorliegen, nicht mehr aus. Denn während der Separationslinienintegration treten die gleichen Problembereiche auf, wie sie bereits für die Parallelisierung der Stromlinienberechnung in Abschnitt 3.4.4 beschrieben wurden. Die Integrationlinien können aus den Datenblöcken, die dem einzelnen *Worker* zugewiesen sind, herauslaufen. Für den vorliegenden Fall eines Schnittes im Nasenmodell werden daher direkt nach der Triangulierung die von den einzelnen *Workern* berechneten Teile des zweidimensionalen Strömungsfeldes in einem temporären Bereich des Dateisystems abgelegt. Direkt vor der Separationslinienberechnung werden dann vom jeweiligen *Worker* alle Schnittteile eingeladen. Die Integration kann nun ohne zusätzliche Kommunikation parallel durchgeführt werden. Nach der Berechnung werden die Ergebnisse zum *Master-Worker* zum Einsammeln aller Ergebnisteile gesendet. Das Ergebnis ist in Abbildung 39 (rechts) zu sehen.

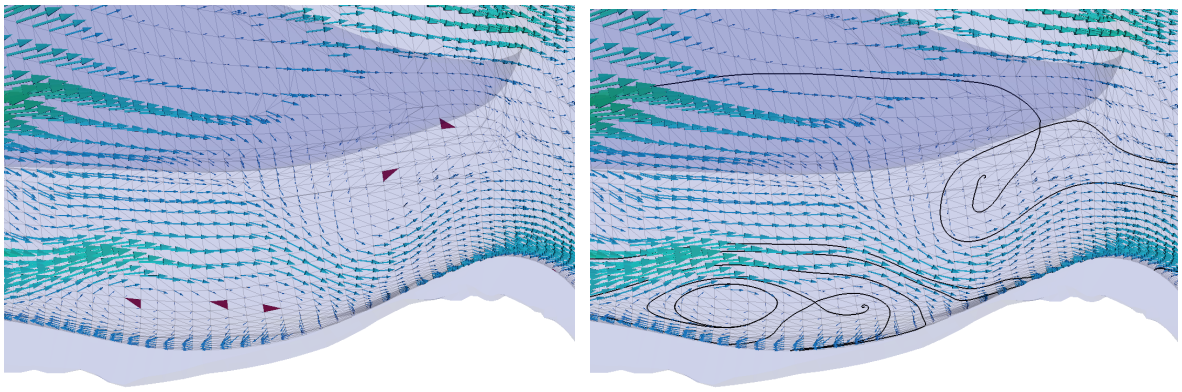

Abbildung 39: Ausschnitt unterhalb der unteren Nasenmuschel mit Darstellung von Geschwindigkeitsvektoren, Zellen mit kritischen Punkten (links) und Separationslinien (rechts)

Das beschriebene Beispiel im Nasendatensatz projiziert das Geschwindigkeitsfeld auf eine Ebene, sodass lediglich Algorithmen zur Behandlung von zweidimensionalen kritischen Punkten benötigt werden. Die parallele Bestimmung von dreidimensionalen kritischen Punkten wurde in weiteren Experimenten untersucht. Dabei kam der Algorithmus von Globus et al. [GLOB91] zum Einsatz, der auf strukturierten Gittern arbeitet und anhand der an den Eckpunkten einer Zelle definierten Geschwindigkeitsvektoren die aktuelle Zelle als möglichen Kandidaten für eingeschlossene kritische Punkte sucht. Kandidatenzellen werden per Bisektion unterteilt und die resultierenden Unterzellen erneut daraufhin untersucht, ob sie Kandidatenzellen sind. Dies kann iterativ fortgeführt werden bis keine Kandidatenzellen mehr gefunden werden oder eine gewisse Unterteilungstiefe erreicht wird. Die übrig gebliebenen Kandidaten werden dann verwendet um per Newton-Iteration oder einfacher Mittelpunktbestimmung den vorliegenden kritischen Punkt zu berechnen. Anhand des dort vorliegenden Geschwindigkeitsgradiententensors werden sie in einem abschließenden Schritt klassifiziert. Das Ergebnis kann mithilfe von Symbolen, die das Eigensystem des Geschwindigkeitsgradiententensors repräsentieren, dargestellt werden. Einen Eindruck von der Topologieeinteilung des Vektorfelds ist durch die zusätzliche Berechnung von Separationslinien möglich, wie das Beispiel in Abbildung 40 zeigt.
Wegen des zellbasierten Ansatzes und der zum Teil erheblichen Berechnungslast für Bisektion und Klassifizierung war zu vermuten, dass sich ein zusätzlicher deutlicher *Speed-up* einstellen würde, wenn mehrere Prozesse gleichzeitig einen einzigen Datenblock gemeinsam bearbeiten würden, indem sie sich die Zellen untereinander aufteilen. Wegen der ungleichen Verteilung von möglichen kritischen Punkten innerhalb eines Datensatzes besteht jedoch die Gefahr der Unbalanciertheit der Prozessoren und somit eines Einbruchs der Skalierbarkeit. Daher wurde nun OpenMP (vgl. Abschnitt 2.2) anstelle von MPI verwendet, um eine Lastverteilung der Berechnung durch die dynamische Hinzunahme von Prozessoren zu erreichen.

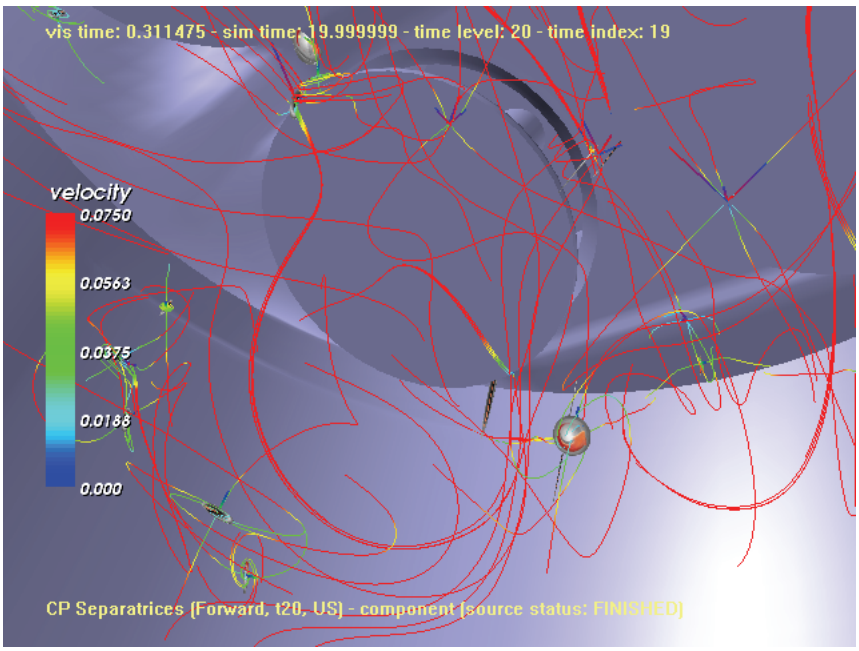

Abbildung 40: Kritische Punkte, dargestellt als Eigensystemsymbole, und zusätzlich berechnete Separationslinien im Motordatensatz

Da OpenMP auf SMP-Knoten beschränkt ist, wurden Balancierungsstrategien lediglich auf dem SunFire-Cluster evaluiert. Die folgenden Tests fanden auf einem E6800-Knoten mit 48 Prozessoren statt. Angewendet wurden die Turbine sowie der Verdichtungsstoßdatensatz. Der erste ist ein MB-Datensatz, während der letztere jeweils einen rectilinearen Einzelblock pro Zeitschritt aufweist (Datensatzeigenschaften: vgl. Anhang A).
Die Daten wurden im Vorfeld komplett geladen, um den hierdurch zu erwartenden Einfluss auf die Untersuchung zu vermeiden. Die Berechnung des Eigensystems für die Klassifizierung von kritischen Punkten musste in eine kritische Sektion gekapselt werden, da die Bestimmung des Geschwindigkeitsgradienten in VTK nicht *thread-safe* ist. Des Weiteren wurde die Suche per Bisektion innerhalb einer Zelle abgebrochen, sobald in dieser Zelle ein erster kritischer Punkt identifiziert wurde. Dadurch wird die Last durch Bisektion, die zum Teil in hunderten von Punkten münden kann, und für die anschließende Klassifizierung mithilfe der Eigenwertanalyse deutlich reduziert ohne einen nennenswerten Einfluss auf die Aussagekraft des erzielten Gesamtergebnisses zu haben.

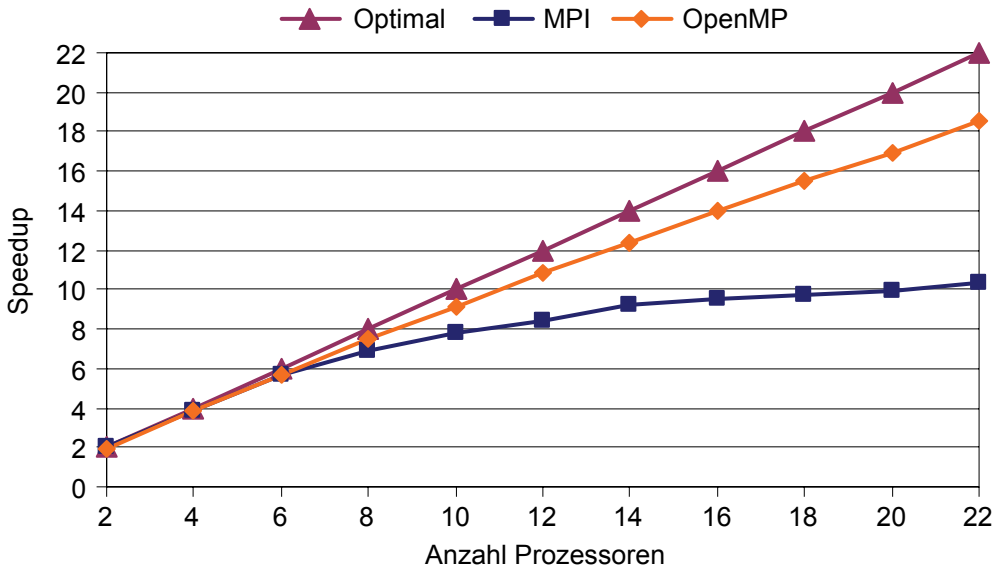

Abbildung 41: Erreichter *Speed-up* für die Berechnung von kritischen Punkten unter Verwendung von MPI und OpenMP, Turbine, Zeitschritt 10 bis 15

Die nachfolgenden OpenMP-Messungen zeigen Ergebnisse unter Verwendung eines einzigen *Workers* von Viracocha. In einem ersten Versuch wurden mit OpenMP die Blöcke der Turbine aus den Zeitschritten 10 bis 15 verteilt, was zu einer Bearbeitungslast von 864 Blöcken führt. Vergleicht man den *Speed-up* der OpenMP-Messung mit Ergebnissen, die sich durch Parallelisierung mit MPI erzielen lassen, kann festgestellt werden, dass die Erwartung einer besseren Lastverteilung erfüllt wurde. Abbildung 41 fasst die Ergebnisse zusammen.
In einem nächsten Versuch wurde *Nested OpenMP* eingesetzt, indem in einer inneren Schleife zusätzlich die Zellen des aktuell zu bearbeitenden Blocks auf weitere *Threads* aufgeteilt wurden. Die Turbine besitzt 144 Blöcke pro Zeitschritt mit insgesamt 2.373.600 Zellen, sodass die mittlere Blockgröße circa 16.000 Zellen umfasst. Dabei schwankt die tatsächliche Blockgröße deutlich. Ebenso ist die Verteilung der kritischen Punkte sehr ungleichmäßig, was ebenfalls zu Balancierungsproblemen führt. Das hauptsächliche Problem ist jedoch die Anzahl der Zellen, die ein *Thread* zugeordnet bekommt. Wird diese zu gering, steigt der Anteil der *Thread*-Koordinierung, und die Effizienz der verschachtelten Parallelisierung bricht ein. Dies kann deutlich in Abbildung 42 abgelesen werden, die einen Ausschnitt aus den erzielten Ergebnissen präsentiert. In der ersten Schleife wurde den *Top-Level-Threads* immer ein Block nach dem anderen (*Scheduler*-Parameter: [*dynamic*,1]) zugewiesen. Die Aufteilung der Zellen in der zweiten Schleife fand automatisch mit der *Scheduler*-Strategie [*guided*] statt (vgl. Abschnitt 2.2).

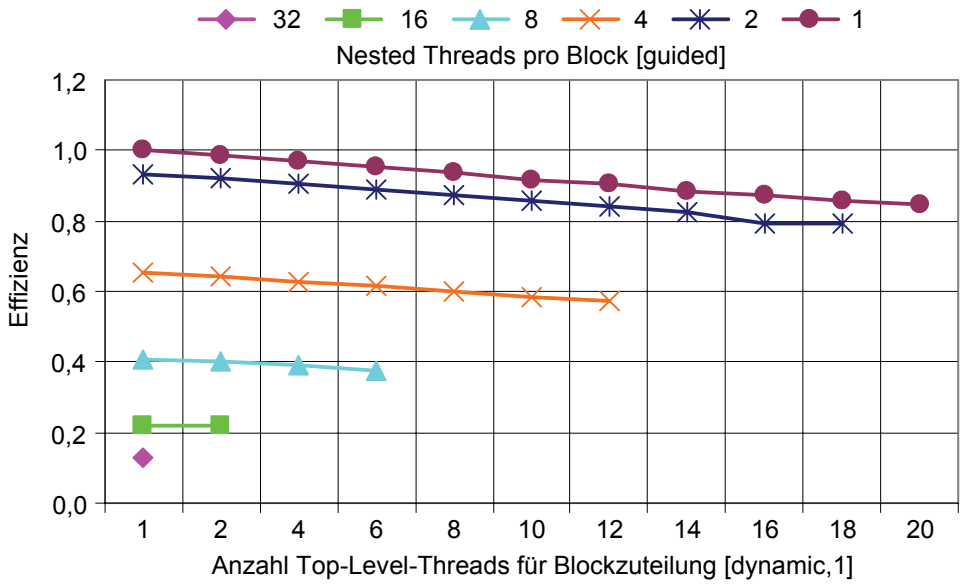

Abbildung 42: Berechnung von kritischen Punkten im Turbinendatensatz, Zeitschritt 10 bis 15, mithilfe von *Nested OpenMP*

Die Blöcke des Verdichtungsstoßdatensatzes sind mit der jeweils konstanten Anzahl von 1.950.399 Zellen deutlich größer. In Abbildung 43 ist erkennbar, dass nun dasselbe Experiment unter Verwendung dieses Datensatzes von Zeitschritt 300 bis 329 wesentlich bessere Ergebnisse liefert. Der leichte Einbruch bei 14 *Top-Level-Threads* und die anschließende Verbesserung ist durch die Anzahl von nur insgesamt 30 zur Verfügung stehenden Blöcken zu erklären. Die Effizienz der inneren Verteilungsschleife fällt erst mit der Verwendung aller im Cluster-Knoten vorhanden 48 Prozessoren unter 0,8.
Eine optimale Zellzuteilung, wenig Kommunikation und die günstige Verteilung der relativ wenigen kritischen Punkte begünstigen das gute Resultat. Die Effizienzschwankungen in Abbildung 43 sind hauptsächlich Balancierungsproblemen in der Berechnungslast zuzuschreiben. Von 92,29 Sekunden im sequentiellen Fall konnte die Laufzeit mit 6 *Top-Level-Threads* und 8 Zellschleifen-*Threads* auf 2,19 Sekunden gesenkt werden. Diese Kombination, die alle 48 Prozessoren des Cluster-Knotens verwendet, liefert daher mit einem Wert von 42,14 den besten *Speed-up* in der Messung.

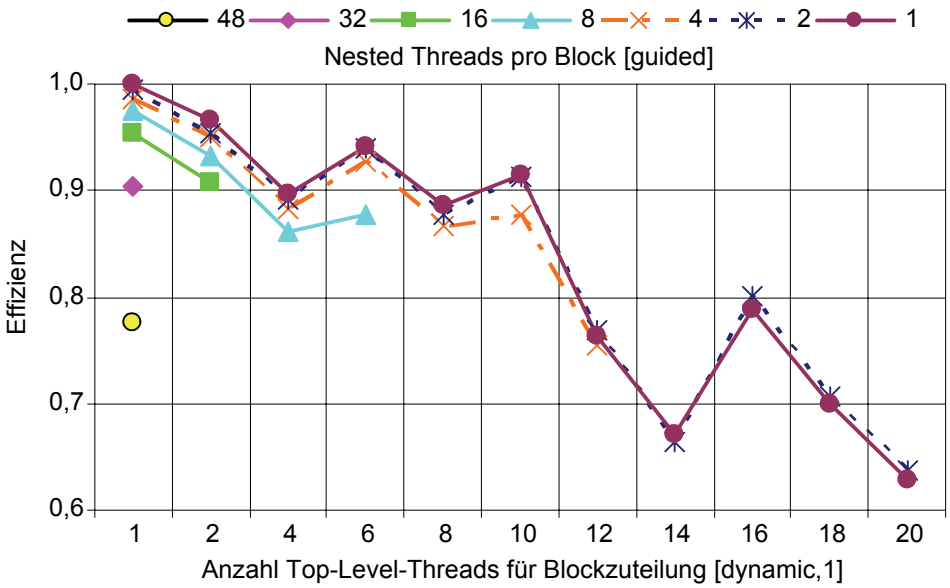

Abbildung 43: Berechnung von kritischen Punkten in einem Verdichtungsstoß, Zeitschritt 300 bis 329, mithilfe von *Nested OpenMP*

Die Annahme, dass sich für die Berechnung von kritischen Punkten ein hoher *Speed-up* durch den Einsatz möglichst vieler Prozessoren erreichen lässt, konnte eindeutig bestätigt werden. Die Messungen belegen zudem, dass der Mehrwert geschachtelter Parallelisierung mit der Größe der Datensätze steigt. Der hier dargestellte Extraktionsalgorithmus lässt sich auch auf andere Zelltypen erweitern, sodass *Nested OpenMP* dann ebenfalls für große unstrukturierte Datensätze zur Verfügung stehen würde.

4 Das Viracocha Data Management System

Viracocha bietet mit den Ansätzen zur Parallelisierung bereits weit reichende Methoden für das CFD-Postprocessing an. Die bisher dargestellten Ergebnisse zeigen aber auch, dass immer wieder Balancierungsprobleme auftreten, die vor allem durch das während der laufenden Berechnung erforderliche Nachladen von Daten herrühren. Einige Extraktionsalgorithmen erreichen durch den Einsatz speziell angepasster Strategien durchaus verbesserte Berechnungsbalancierungen. Diese stehen jedoch nicht automatisch auch für andere oder für neu zu entwickelnde Algorithmen zur Verfügung.
Daher wurde das *Viracocha Data Management System* (VDMS) entwickelt, das als *Middleware* fest in Viracocha integriert wurde. Dadurch steht es allen Algorithmen zur Verfügung, wobei die Verwendung wiederum transparent stattfindet, d. h., dass der Algorithmus i. Allg. die Existenz des VDMS nicht kennt und somit auch sein Verhalten nicht kontrollieren muss.
Das Hauptziel ist das Verwalten der benötigten Daten. Der wichtigste Aspekt ist dabei die Speicherhierarchie (vgl. Abbildung 44). Dauerhaft auf einem entfernten File-Server abgelegt, müssen die Datensätze zuerst einmal in den lokalen Hauptspeicher geladen werden. Von da aus werden die Daten bei Bedarf in den prozessornahen, häufig mehrere Stufen umfassenden Cache kopiert. Dabei unterscheiden sich Zugriffszeiten und Speichergröße erheblich. So benötigt der wenige kByte-große Level-1-Cache (*On-Chip-Cache*) in heutigen Hochleistungsrechnern weniger als eine Nanosekunde bis das angeforderte Datum dem Prozessor zur Verfügung steht. Diese Latenzzeit verzehnfacht sich beim Nachladen aus dem Level-2-Cache und verhundertfacht sich, wenn sie dort nicht vorliegen und daher vom Hauptspeicher angefordert werden müssen.

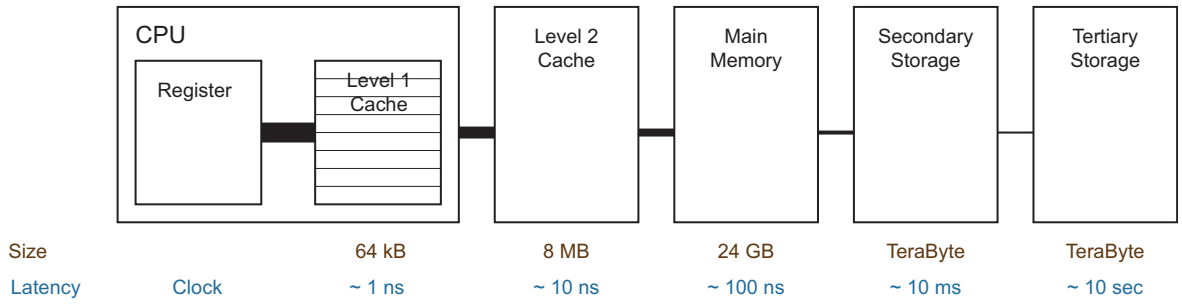

Abbildung 44: Speicherhierarchie, Speichergrößen und Latenzen (Beispiel: Sun-Fire 6800)

Das Problem, das für prozessornahe Caches beschrieben wurde, gilt genauso für die unterschiedlich langen Zugriffszeiten auf Hauptspeicher, Dateisystemen (sekundäre Speichersysteme) und Archivsystemen (tertiäre Speichersysteme). Das VDMS beschäftigt sich daher vor allem mit dem Laden der Daten von entfernten Dateisystemen. Folgende Hauptstrategien wurden realisiert:

Caching: Bereits geladene Daten werden, solange ausreichend Platz zur Verfügung steht, in applikationsnahen Speichern gehalten. Bei erneuter Anfrage des Datums wird es nicht vom entfernt liegenden Dateisystem geladen sondern direkt aus dem Cache geholt.

Prefetching: Daten werden bereits vor der Anfrage in den Cache geladen. Wird das Datum dann tatsächlich angefordert, kann der Cache das Datum sofort liefern. Der Geschwindigkeitsvorteil wird dadurch erreicht, dass die Daten im Hintergrund parallel zur laufenden Berechnung geladen werden. Der Algorithmus muss im günstigsten Fall daher nicht mehr während einer Datenanfrage seine Berechnung unterbrechen. Prefetching-Strategien basieren hauptsächlich auf räumlichen Nachbarschafts-

beziehungen. Eine im Rahmen des CFD-Postprocessings neue Strategie ist die Verwendung von Wahrscheinlichkeitsvorhersagen basierend auf Markov-Ketten.

Ladestrategien: In Abhängigkeit von Datenstrukturen und verfügbaren Hardware-Ressourcen lassen sich optimierte Ladestrategien registrieren und ausführen. Liegen zum Beispiel parallele Dateisysteme vor, können entsprechende Ladestrategien definiert werden, die diese Technik ausnutzen.

***Data Service*:** Dieser Ansatz verfolgt Lastbalancierung durch die optimale Zuordnung von Daten an berechnende *Worker*. Dabei kennt eine zentrale Instanz den Inhalt aller Caches, sodass ein nachfragender *Worker* Daten für die Verarbeitung zugewiesen bekommt, die, wenn möglich, bereits in einem der Caches vorliegt.

Da das VDMS unabhängig vom eigentlichen Anwendungsgebiet ist und somit kein Wissen über Visualisierung und Computergraphik aufweist, werden zunächst ergänzend zu Kapitel 2 wichtige Vorarbeiten aufgezeigt, die speziell im Bereich effizienter Datenverwaltung geleistet wurden. Anschließend wird das Design des VDMS beschrieben, bevor auf die einzelnen Strategien bezüglich Caching (Abschnitt 4.3), Prefetching (Abschnitt 4.4) und Datenladen (Abschnitt 4.5) eingegangen wird. Dieses Kapitel wird mit der Beschreibung von *Data-Service*-Objekten (Abschnitt 4.6) und einer ausführlichen Analyse des VDMS beim Einsatz von CFD-Extraktionsalgorithmen (Abschnitt 4.7) abgeschlossen.

4.1 Parallele Datenmanagementsysteme

Neben den bisher vorgestellten Visualisierungsprogrammen, die eine höhere Performanz vorwiegend durch Lastverteilung auf Mehrprozessormaschinen zu erreichen versuchen, haben sich andere Forschungsrichtungen verstärkt mit der Verwaltung und dem Zugriff auf große Datensätze beschäftigt. Die hier entwickelten Ansätze stehen in der Regel als *Middelware* zur allgemeinen Benutzung in anderen Anwendungen zur Verfügung. Da das VDMS ebenfalls als *Middleware* ausgelegt ist und Teilaspekte der hier vorgestellten Systeme abdeckt, werden die wichtigsten Ansätze nun kurz skizziert.

Einen Mittelweg zwischen Datenbank und parallelem Dateisystem geht das *Meta-Data Management System* (MDMS). Es platziert sich zwischen der Anwendung und einem hierarchischen Dateisystem (*Hierarchical Storage System*, HSS). Benötigt die Anwendung ein Datum, dann wählt das MDMS eine passende I/O-Technik aus und schlägt diese als bevorzugte Zugriffsmethode vor. Die Applikation greift sodann direkt auf das HSS zu, wobei es ihr überlassen bleibt tatsächlich die vorgeschlagene Methode zu verwenden. Entwickelt wurde das System an der Northwestern University von Choudhary et al. [CHOU99], um eine einheitliche Schnittstelle zwischen datenintensiven Anwendungen und einem HSS zu bilden. Als optimierte I/O-Techniken sind u. a. *Data Sieving* und *Collective I/O* vorgesehen.

Data Sieving reduziert Latenzzeiten, die durch die parallele Anfrage kleiner, nicht zusammenhängender Datenstücke anfallen. Statt lauter kleine Einzelzugriffe zuzulassen, versucht dieser Ansatz die Anfragen zu wenigen großen Zugriffen zu bündeln. Dabei werden auch Daten, die eigentlich zwischen den angefragten Datenblöcken liegen in den Systempuffer eingelesen. Dort werden die eigentlich gewünschten Teildaten extrahiert und an die anfragenden Prozesse weitergeleitet. Wollen dagegen mehrere Prozesse gemeinsam einen großen Datenblock bearbeiten, bietet sich *Collective I/O* an. In diesem Fall wird wiederum nur ein einziger, großer Dateizugriff durchgeführt, wodurch erneut Latenzzeiten eingespart werden. Anschließend werden die Prozesse mit ihren Teildaten versorgt, wobei die Datenzuteilung auf verschiedenen Systemebenen stattfinden kann. Zum einen kommt hierfür der eigentliche Client [ROSA93] in Frage, zum anderen können diese Aufgabe aber auch I/O-Server [SEAM95] und gar das Festplattensystem [KOTZ01] übernehmen.

Metadaten, wie sie für die Verwaltung von Datenreferenzen im MDMS benötigt werden, kommen auch im *Storage Resource Broker* (SRB) zum Einsatz. Diese *Middleware* wurde von

Baru et al. in [BARU98] vorgestellt und sieht für Datenanfragen eine einheitliche SQL-ähnliche Sprache vor. Die Struktur der Daten kann dabei völlig heterogen und beliebig im System verteilt sein. Das SRB sorgt dafür, dass es für die Anwendung transparent ist, ob sich die Daten auf einem Dateisystem, in einer Datenbank oder auf einem Archivsystem befinden.
Mit der Verabschiedung von MPI-2 hat das MPI-Forum mit MPI-I/O auch eine Schnittstelle für paralleles I/O als festen Bestandteil der Spezifikation definiert. Eine der Implementationen ist ROMIO, das auf einer portablen parallelen I/O-Schnittstelle mit dem Namen ADIO (*Abstract Device Interface for I/O*) aufsetzt [THAK99a]. ADIO kapselt im Prinzip hardware-spezifische Implementierungsdetails und bietet nach außen hin einige Basisfunktionen an, die nun von ROMIO verwendet werden, um die von MPI-I/O vorgeschriebenen Schnittstellen-funktionen zu realisieren.
Der von No et al. in [NO03] beschriebene *Scientific Data Manager* (SDM) stellt sich als Kombination von Datenbank und parallelem I/O dar. Zum Benutzer hin existiert eine einheitliche Schnittstelle, über die auf die Daten zugegriffen werden kann. Innerhalb des Systems kommt ein paralleles Dateisystem für die Speicherung zum Einsatz. Hier wird nun auf das beschriebene MPI-I/O aufgebaut, sodass die verschiedenen darin enthaltenen I/O-Optimierungen zur Verfügung stehen. Beispielsweise kommen verstärkt kollektive I/O-Funktionen zur Anwendung. Die Datenbank dagegen nimmt applikationsspezifische Metadaten auf. Dies passiert jedoch für den Anwender transparent im Hintergrund, sodass sich für ihn die Benutzung dieses Systems erheblich vereinfacht.
Damit die Verfahren, die auf parallelem I/O basieren, überhaupt effektiv einsetzbar sind, müssen parallele Dateisysteme vorhanden sein. Eines der am weitesten verbreiteten parallelen Dateisysteme ist das *Global Parallel File System* (GPFS) von IBM [SCHM02]. Diese findet sowohl in Supercomputern als auch in Linux-Clustern Anwendung.

4.2 Aufbau des VDMS

Beim Design lagen einige wesentliche Anforderungen zugrunde. Die weitestgehende Entscheidung war die Kapselung des Managements von der eigentlichen Berechnung, d. h., das VDMS sollte weder Wissen über den zur Anwendung kommenden Algorithmus noch über die verwendeten Daten, also Informationen über Organisation und Datenformat, besitzen. Damit kann sich das VDMS als ergänzende Komponente in der mittleren Schicht von Viracocha ansiedeln. Nach außen stellt es Schnittstellen zur Verfügung und beeinflusst die Funktionalität der *Worker*.
Da das VDMS ein integrativer Bestandteil von Viracocha ist, läuft es somit auch nur auf der *WorkHost*-Seite. Das Visualisierungs-*Frontend* benötigt keinerlei Wissen über das Datenmanagement, sodass es transparent vom eigentlichen Auftraggeber stattfinden kann. Dennoch bietet das VDMS Dienste für den *VisHost* an, um beispielsweise bereits berechnete Daten erneut abzufragen. Das Verhalten des Datenmanagements wird in der Startphase des *WorkHosts* durch Einstellungen in Konfigurationsdateien bestimmt. Da sein Einsatz jedoch optional ist, besteht die Möglichkeit ihn mit der Berechnungsanforderung zu deaktivieren oder das voreingestellte Verhalten zu verändern.
Das VDMS ist kein isoliertes, monolithisches Modul, sondern ist der verteilten Struktur von Viracocha angepasst. Es teilt sich im Wesentlichen in die beiden folgenden Haupt-komponenten auf:

- **ein Server**
- **mehrere *Proxies***

Das Server-Objekt übernimmt zentrale Verwaltungsaufgaben, während die *Proxy*-Variante jedem *Worker* zugewiesen wird. Die Kommunikation untereinander baut dabei auf die bereits existierende Kommunikationsschicht von Viracocha auf. Für die Kommunikation zwischen den verteilten Komponenten des VDMS wurden lediglich weitere Kommunikationsendpunkte

hinzugefügt. Weit reichender ist dagegen die zusätzliche Funktionalität, die in die Kommunikationsschicht integriert wurde, um auch paralleles I/O zu ermöglichen.
In der Anwendungsschicht kann über Schnittstellen direkt mit dem Datenmanager kommuniziert werden. Weitere Implementierungsklassen, um die Funktionsfähigkeit des VDMS zu ermöglichen, sind nicht notwendig. Da das VDMS jedoch von den tatsächlich zum Einsatz kommenden Datentypen unabhängig ist, müssen auf der Anwendungsschicht zumindest die verwendeten Daten konkretisiert werden. Dafür bietet das VDMS spezielle Datenzugriffsklassen als Basisklassen an, von denen konkrete auf den jeweiligen Datentyp angepasste Zugriffsklassen abgeleitet werden.
Die Integration des VDMS in die verteilte Struktur von Viracocha wird in der folgenden Abbildung 45 nochmals verdeutlicht.

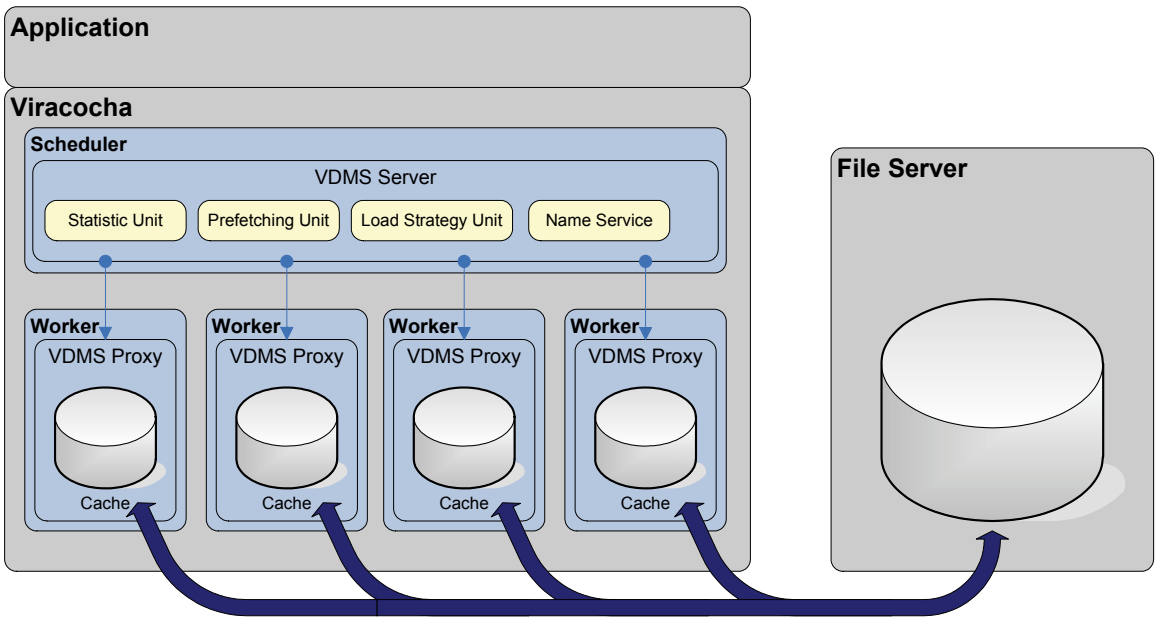

Abbildung 45: Integration der Datenmanagementkomponenten in die Organisationsschicht von Viracocha

4.2.1 Server

Der Server, als einer von zwei Hauptkomponenten des Datenmanagementsystems, ist beim *Scheduler* angesiedelt und verwaltet alle globalen Informationen des Datenmanagers. Vergleicht man die Aufgaben des Servers mit denen des *Schedulers*, so lassen sich identische Aufgabenbereiche identifizieren; wobei sie sich beim Server auf die Daten und beim *Scheduler* auf die Berechnungen beziehen.
In der Regel läuft der Server als Teil des *Schedulers*, kann aber auch als selbstständiger Prozess initiiert werden. Wie der *Scheduler* kann der VDMS-Server als zentrale Instanz schnell zum Falschenhals werden. Daher wurde bei seiner Entwicklung besonders auf Stabilität und Performanz geachtet. Vor allem läuft er als nicht blockierender Prozess, sodass auch andere wichtige Dienste verzögerungsfrei ihre notwendigen Verwaltungsarbeiten durchführen können.
Wegen der engen Anbindung an die zentrale Instanz des VDMS und an den *Scheduler* ist der Server in der Lage auch direkt vom *VisHost* an ihn gestellte Anfragen zu bearbeiten. Dabei kann es sich um ganz allgemeine Informationen über den Zustand des VDMS handeln, wie beispielsweise der momentanen Auslastung und aktuell verwendeter Strategien. Aber auch der Zugriff auf den zentralen Namensservice ist möglich, worüber sich Informationen über zurzeit bearbeitete Daten erfragen lassen.
Insbesondere besteht die Möglichkeit bereits berechnete Daten, die sich beim *WorkHost* abspeichern lassen, ohne erneuten Berechnungsaufwand nochmals abfragen zu können. Umgekehrt lassen sich vom *VisHost* aus Daten an den Server schicken. Dies macht dann Sinn,

wenn ein Kommando beauftragt werden soll mit für den *WorkHost* nicht erreichbaren Daten zu arbeiten. Dies können temporär bestimmte Ausgangsdaten oder auf lokalen Platten befindliche Daten sein.
Die ohne Berechnungsaufwand zu bedienenden Anfragen eines *VisHosts* sind lediglich eine erweiterte Funktionalität des Servers. Seine Hauptaufgabe ist die zentrale Koordination des Datenmanagements während einer Berechnung. Jeder rechnende *Worker* bekommt eine *Proxy*-Komponente zugewiesen, die dort jeweils als Stellvertreter (engl.: *Proxy*) des Datenmanagers tätig wird. Der Server dient im Hintergrund mit zentral verwalteten Namensdiensten, kann das Behandeln der Daten in Statistiken zusammenfassen und hilft mit Strategieentscheidungen aus.

4.2.2 Proxy

Die dem *Worker* zugewiesene *Proxy*-Komponente dient als Schnittstelle zwischen Kommando und Datenverwaltung. Datenanfragen des laufenden Kommandos werden direkt vom *Proxy* entgegengenommen. Die Bearbeitung läuft lokal auf dem jeweiligen Knoten und ohne Wissen über andere *Proxies*. Sollen dennoch Informationen oder Daten zwischen *Proxies* ausgetauscht werden, so kann dies nur unter Anweisung des Servers geschehen.
Aber auch das Kommando bleibt weitestgehend im Unklaren darüber was mit seiner Datenanfrage passiert. Der *Proxy* ist eine *Black Box*, die die Anfrage entgegennimmt und daraufhin das Datum zur Verfügung stellt. Woher die Daten kommen und welche Strategien verwendet wurden, liegt außerhalb der Kontrolle des Kommandos. Dies hat allerdings auch den Vorteil, dass Optimierungen der Datenbereitstellung alleinige Aufgabe des *Proxies* ist.
Andererseits sind die Kommandos nicht verpflichtet die *Proxy*-Schnittstelle zum Laden von Daten zu verwenden. Bei Bedarf können auch am Datenmanagement vorbei eigene Daten geladen und verändert werden. Damit lassen sich auch speziellere Anforderungen, die vom VDMS nicht erfüllt werden können, realisieren.

4.2.3 Dienste

Das VDMS bietet mehrere Dienste an, die sowohl beim Server als auch bei den *Proxies* abgerufen werden können. Die Implementierung und die zu erwartenden Ergebnisse sind dann jedoch unterschiedlich.

Namensdienst: Er bildet logische Namen, die in Viracocha als Datenreferenzen bezeichnet werden, auf Ressourcenadressen ab. Diese Abbildung ist notwendig, da der einfache Dateiname als Referenz häufig ungenügend ist. Besonders in heterogenen Umgebungen ist die Pfadangabe nicht notwendigerweise eindeutig. Verschiedene Datenplatten und Netzlaufwerke, vor allem unter Windows-Betriebssystemrechnern, führen dazu, dass ein und dieselbe Datei unterschiedlich aufgelöst werden muss. Aber nicht nur der Dateiname spezifiziert eine Datei. Datenformate und Dateiabhängigkeiten zum Beispiel in Form von Block- und Zeitinformationen werden durch weitere Metadaten spezifiziert. Damit in Viracocha dennoch ein einfaches Handling z. B. auch von großen auf mehreren Dateiverzeichnissen verteilten CFD-Datensätzen möglich ist, bekommen Datenobjekte eine eindeutige Referenz zugewiesen.
Der Namensdienst dient nun alleinig der einfachen knotenübergreifenden Referenzierung von Datenobjekten. Dabei werden keinerlei weitergehende Strukturinformationen abgelegt. Ändert sich die Adresse einer Ressource, muss auch eine neue Referenz vergeben werden. Persistenz ist ebenfalls nicht implementiert. Datenreferenzen sind nur zur Laufzeit des Systems gültig und werden nach Beendigung zerstört.
Die Zuweisung und Verwaltung der Datenreferenzen erfolgt im VDMS-Server. Damit ist die Eindeutigkeit im System gewährleistet. Der *Proxy*-Namensdienst dagegen muss dafür sorgen, dass die richtige Abbildung von Datenobjekt und Referenz stattfindet. Hierfür muss bei jeder Anfrage beim *Proxy*-Namensdienst eine Kommunikation mit

dem zentralen Server-Namensdienst aufgebaut werden. Wird für eine unbekannte Ressource der Namensdienst bemüht, generiert er automatisch eine neue Referenz.

Statistik-Einheit: Diese sammelt Daten über das Verhalten des Datenmanagers. Hierfür werden auf jedem Knoten, also für *Proxies* und Server, Messwerte über Bandbreiten, Cache-Trefferraten, Laufzeiten etc. festgehalten. Damit kann während der Bearbeitung einer Datenanfrage jederzeit nachvollzogen werden, wie effizient das VDMS wirklich arbeitet. Diese statistischen Erhebungen sind aber nicht nur für die Auswertung durch den Entwickler gedacht. Vielmehr werden sie für interne Komponenten wie z. B. die Prefetching-Einheit benötigt, um dann akkurate Strategieentscheidungen treffen zu können.

Strategie-Einheit: Als zentrales Element des Datenmanagers gilt der Cache. Um ihn zu befüllen existiert diese selbständige Einheit, die mehrere Strategien zum direkten Laden von Daten in den Cache verwaltet.

Prefetching-Einheit: Als zweite Einheit zum Befüllen der Caches versucht sie mithilfe der Statistik-Einheit sinnvolle Daten im Voraus in den Cache zu laden.

Abschließend sei vermerkt, dass ein explizites Zeitmanagement nicht implementiert wurde. Die einzigen Zeitinformationen, die bisher vom VDMS abgefragt werden, sind Zeitstempel von auf der Festplatte abgelegten Dateien; es wird also auf die zentrale Zeitvergabe des Dateisystems zurückgegriffen.

4.3 Caching

Das Prinzip des Cachings baut auf die Lokalität von Daten [FROE96]. Dabei besitzen Daten sowohl temporale als auch räumliche Lokalität. Bei der temporalen Lokalität geht man davon aus, dass das Datum auch in Zukunft mit hoher Wahrscheinlichkeit wieder benötigt wird. Ein bedeutendes Maß dieser Eigenschaft ist die temporale Distanz. Diese gibt an wie viele anderweitige Zugriffe stattgefunden haben, bis das Datum erneut angefordert wurde. Bei der räumlichen Lokalität wird dagegen die Nachbarschaft von Daten bewertet. Dabei geht man davon aus, dass beim Laden eines Datenblocks mit hoher Wahrscheinlichkeit auch der nachfolgende Datenblock angefordert wird.
Braucht ein Algorithmus ein Datum, schaut das System zuerst im Cache nach. Wurde das Datum dort gefunden, liegt ein so genannter *Cache Hit* vor. Andererseits spricht man von einem *Cache Miss*, der ein unverzügliches Nachladen des Datums bewirkt. Um die Effizienz des Cachings zu bewerten, kann die Trefferrate (engl.: *Hit Rate*) ermittelt werden. Dabei werden die erfolgreichen Cache-Anfragen im Verhältnis zur Gesamtanfrage bewertet:

$$Hit\ Rate = \frac{\#\ Cache\ Hits}{\#\ Cache\ Requests} \tag{1}$$

Da sich in der Regel nur bereits explizit angeforderte Daten im Cache befinden, führt der Zugriff auf ein neues Element immer zu einem *Cache Miss*. Dieser Umstand wird auch als Kaltstart-*Miss* oder als *Compulsory Miss* bezeichnet [WIEL96]. Da Caches nur eine begrenzte Größe aufweisen, sind sie schnell mit Datenelementen aufgefüllt. Weitere Datenanforderungen, die nicht vom Cache bedient werden können, führen dazu, dass die neu geladenen Daten andere im Cache befindlichen Elemente verdrängen. Soll erneut auf ein nun entferntes Element zugegriffen werden, dann kommt es zwangsläufig zu einem weiteren *Cache Miss*, der in Abgrenzung zum Kaltstart-*Miss* als Kapazitäts-*Miss* bezeichnet wird.

4.3.1 Cache-Stufen des VDMS

Das *Viracocha Data Management System* verfügt über zwei Cache-Stufen, aus denen Anfragen nach Daten bedient werden können. Sie werden als primärer und sekundärer Cache bezeichnet. Jeder Knoten von Viracocha besitzt diese Cache-Hierarchie, wobei wiederum die

Caches des Servers gegenüber den *Proxie*-Caches andere Aufgaben wahrnehmen müssen. So werden hieraus vor allem Datenanfragen des *VisHosts* beantwortet und die *Proxies* bei Bedarf mit zusätzlichen Berechnungsdaten versorgt. Die beiden Caches eines *Proxies* dagegen stehen in erster Linie dem Kommando zur Verfügung. Daten, die für die Extraktion benötigt werden, werden hier zum schnelleren Zugriff abgelegt. Da Algorithmen Daten von anderen *Workern* anfragen können, werden diese ebenfalls aus den *Proxy*-Caches genommen.

4.3.1.1 Primärer Cache

Der primäre Cache wird im Hauptspeicher angelegt und verwaltet immer einen so genannten *Data Handler* und die zugehörige Datenreferenz. Dadurch vereinfacht sich die Schnittstelle des Caches erheblich. Der *Data Handler* kapselt zwei weitere Objekte (siehe Abbildung 46), welche die eigentlichen Daten genauer spezifizieren. Das sind zum einen das *Data Item*, das die tatsächlichen Daten hält, und das *Data-Access*-Objekt, über das der Zugriff auf die Daten erfolgt.

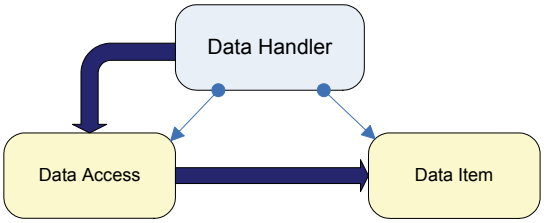

Abbildung 46: Der *Data Handler* kapselt Datenblock und Datenzugriffsobjekt; Zugriffs- (dünne Pfeile) und Datenflussrichtung (dicke Pfeile)

Die beiden Objekte des *Data Handlers* sind bereits in der Organisationsschicht von Viracocha definiert. Dabei ist das *Data Item* als templatisiertes Container-Objekt ausgelegt, sodass es das Halten von Daten beliebigen Typs erlaubt. Innerhalb des VDMS wird daher nur mit diesen Containern gearbeitet. Sollen jedoch konkrete Operationen auf den Daten durchgeführt werden, muss das *Data-Access*-Objekt verwendet werden. In der Organisationsschicht ist es lediglich als Schnittstelle definiert. Erst in der Algorithmusschicht erfolgt die Implementierung der Datenbehandlungsmethoden. Da hier das Wissen über tatsächlich verwendete Daten vorliegt, wird die Anforderung an das VDMS erfüllt, dass dieses beliebige Daten und Datentypen verwalten kann.
Die Speichergröße des primären Caches lässt sich durch Benutzerangaben konfigurieren. Es wird dabei versucht innerhalb der angegebenen Grenzen zu bleiben. Allerdings ist dies nicht immer möglich. Benötigt ein Algorithmus mehr Daten als in den Cache passen, darf der Cache die vom Algorithmus bereits verwendeten Daten nicht löschen, da einmal von ihm angefordert ein direkter Zugriff besteht. Der Cache wird bei Speicherzugriffen aber nicht mehr konsultiert. Ein eigenmächtiges Entfernen durch den Cache würde daher einen illegalen Speicherzugriff nach sich ziehen. Aus diesem Grund muss der Datenmanager trotz überschrittener oberer Schranke den Speicher und somit die Daten zur Verfügung stellen. Dieser Ansatz ist vertretbar, da der Algorithmus auch ohne Einbeziehung des VDMS diesen Speicherbedarf hätte.

4.3.1.2 Sekundärer Cache

Der sekundäre Cache-Speicher, der optional verwendet werden kann, wird auf den lokalen Festplatten des jeweiligen Berechnungsknotens eingerichtet. Die Cache-Daten werden dabei als Dateien abgelegt. Die Cache-Struktur ist beliebig organisierbar, indem nach Bedarf Dateiverzeichnisse angelegt werden können. Es muss auch keine maximale Cache-Größe spezifiziert werden, die sich dann durch das Fassungsvermögen der Platten ergibt.
Sinn macht die Verwendung des sekundären Caches, wenn Daten verwendet werden, die jedes Mal erst über ein Netzwerk transportiert werden müssen und dies zu erheblichen Latenzen führen kann. Werden daher Daten durch Ersetzungsstrategien aus dem primären

Cache entfernt, werden diese erst einmal im nachgelagerten sekundären Cache abgelegt. Erfolgt eine Anfrage nach einem Datenblock, die nicht bereits durch die erste Cache-Stufe bedient werden kann, schaut der Datenmanager in dem sekundären Cache nach, ob nicht ggf. hier noch lokale Kopien vorliegen. Ist dies der Fall, wird zum Laden nicht der lange Weg durch die Speicherhierarchie bis zur dauerhaften Speicherposition gegangen, sondern sich direkt aus dem sekundären Cache bedient.

4.3.1.3 Namens-Cache

Neben den reinen Daten-Caches existiert noch ein am Namensservice angesiedelter Namens-Cache. *Data Items* werden im gesamten VDMS durch eindeutige Datenreferenzen identifiziert. Da Datenreferenzen jedoch vom Server vergeben und verwaltet werden, die Anfragen aber in der Regel beim *Proxy* eingehen, müssen ständig Anfragen zum Auflösen von Namen in Referenzen vom *Proxy* an den Server gestellt werden. Es hat sich gezeigt, dass die Anzahl der Namensanfragen, besonders mit zunehmender Anzahl von *Proxies*, sehr hoch sein kann. Dadurch wird der Server erheblich belastet.
Namen ändern sich während einer Arbeitssitzung üblicherweise nur selten. Da liegt es nahe Anfragen nach Namensauflösungen ebenfalls bei den *Proxies* zu cachen. Erst wenn der lokale Namens-Cache keine Auflösung anbieten kann, wird die Anfrage an den Server weitergeleitet. Um das Aufblähen des Caches mit alten Informationen zu vermeiden, haben die gespeicherten Datenreferenzen ein Verfallsdatum. Nach einer gewissen Verweildauer werden sie daher aus dem Namens-Cache gelöscht. Eine weitere Anfrage muss dann wieder vom Server beantwortet werden. Diese neue Auflösung kann dann erneut lokal gecacht werden.

4.3.2 Cache-Ersetzungsstrategien

Der Cache-Puffer hat nur eine geringe Größe und muss regelmäßig gecachte Daten ersetzen. Die Strategie, nach der entschieden wird, welche Datenblöcke entfernt werden sollen, heißt Ersetzungsstrategie.
Dabei besteht die Einschränkung, dass Daten nur dann dem Ersetzungsalgorithmus unterworfen sind, wenn das Berechnungskommando diese explizit als nicht mehr benötigt markiert hat. Dies geschieht über eine *HandBack*-Methode des *Proxies*, worüber das Kommando die Freigabe für das angegebene *Data Item* signalisiert. Alle nicht als verwendet markierten *Data Handler* werden bei Platzbedarf der aktuell aktiven Ersetzungsstrategie übergeben. Die implementierten Strategien werden im Folgenden kurz beschrieben.

Least Recently Used (LRU): Diese wohl einfachste, aber wegen ihrer Effizienz sehr häufig eingesetzte Strategie versucht gute Ergebnisse durch die Ausnutzung von temporaler Lokalität zu erreichen. Es wird immer dasjenige Element aus dem Cache entfernt, das am längsten von allen Elementen nicht mehr verwendet wurde.
Sowohl der primäre als auch der sekundäre Cache greifen auf den einfachen LRU-Ansatz zurück. Das Alter eines Datenblocks wird im sekundären Cache anhand des vom Dateisystem vergebenen Dateidatums ermittelt. Der primäre Cache muss dagegen einen eigenen Zeitstempel generieren oder strikt eine Sortierung nach der LRU-Ordnung vornehmen.

Least Frequently Used (LFU): Unter gewissen Umständen kann ein bereits häufig genutztes Element über einen gewissen Zeitraum unbenutzt bleiben und läuft bei der LRU-Strategie sodann Gefahr entfernt zu werden. Der LFU-Ansatz berücksichtigt die Häufigkeit der Benutzung, indem zusätzlich ein LFU-Referenzzähler eingeführt wird. Jedes Cache-Element besitzt einen eigenen Zähler, der jeweils bei einem Zugriff um eins erhöht wird. Ist der Cache voll, wird das Element mit dem geringsten Referenzzähler entfernt.
Da manche Blöcke eine Zeit lang recht häufig verwendet werden können und sodann hohe Referenzzähler aufweisen, die sie vor der Entfernung schützen, auch wenn die

weitere Verwendung unwahrscheinlich ist, kann man den Referenzzähler einer Alterung auszusetzen [ROBI90, WILL93]. Ab der Überschreitung eines gewissen Alters wird der Referenzzähler jeweils halbiert, wodurch die betroffenen Elemente nach einer gewissen Zeit wieder eher für den Ersetzungsfall in Frage kommen.

Frequency Based Replacement (FBR): Das FBR-Verfahren [ROBI90] vereinigt LRU und LFU, wobei die jeweiligen Nachteile behoben werden. Auch das FBR verwaltet eine LRU-Liste von gecachten Elementen. Diese Liste wird nun jedoch in drei hintereinander liegende Partitionen mit festen Längen aufgeteilt (vgl. Abbildung 47).

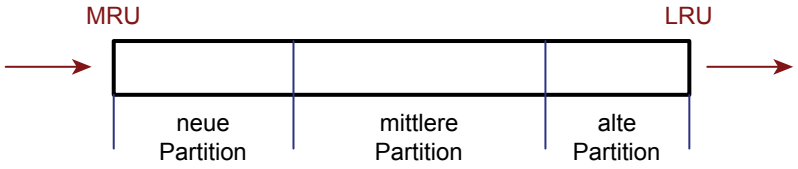

Abbildung 47: Partitionierung der FBR-Liste

Alle neu referenzierten oder erneut referenzierten Elemente (engl.: *Most Recently Used*, MRU) werden als erstes Element in die so genannte neue Partition eingefügt. Der Unterschied der Partitionen liegt in der Behandlung der Referenzzähler. Wird ein Element in der neuen Partition erneut angefragt, dann erfolgt keine Erhöhung. Die Referenzzähler der in den anderen Partitionen befindlichen Elemente werden jedoch weiterhin um eins erhöht. Ist die Liste aufgefüllt und es besteht der Bedarf Platz für ein neues Element zu schaffen, werden alle Elemente in der alten Partition untersucht und derjenige Datenblock mit dem niedrigsten Referenzzähler gelöscht.

PFBR: Diese neue Caching-Strategie basiert auf dem FBR-Ansatz und soll sich durch einen modifizierten Frequenzansatz noch besser an Datenstrukturen und Kommandos im Bereich des CFD-Postprocessings anpassen. Der Referenzzähler beginnt bei neu eingefügten Elementen nicht wie beim FBR bei eins, sondern mit einer zugewiesenen Priorität. Diese Priorität variiert in Abhängigkeit der Laufzeit des Kommandos. Finden die Anfragen an den Cache von einem erst gerade gestarteten Kommando statt, werden die neuen Cache-Elemente mit einer hohen Priorität versehen. Daten, die zu einem späteren Zeitpunkt neu angefragt werden, werden dem Cache mit einer niedrigeren Priorität zugewiesen. Der Sinn dieser variierenden Prioritäten liegt darin, dass zu Beginn eines Kommandos angefragte Blöcke länger im Cache verbleiben können als Blöcke, die erst zum Schluss angefragt werden. Diese späten Blöcke haben nämlich eher die Chance vom erst während der Laufzeit des Kommandos zum Zuge kommenden Prefetching profitieren zu können.

SIZE: Dieser ebenfalls nicht in der Literatur beschriebene Cache-Ansatz verfolgt die Strategie immer den größten Block zu ersetzen. Dadurch wird durch eine einzige Ersetzung immer die maximal mögliche Menge an Speicher freigegeben und somit versucht insgesamt mit einer minimalen Anzahl von Ersetzungen auszukommen. Dabei wird allerdings die Beziehung zwischen Verwendung und Datum völlig ignoriert. Neben dem LRU-Cache ist der SIZE-Cache die zweite Strategie, die auch für den sekundären Cache implementiert wurde.

Random Replacement (RAND): Entfernt wird bei dieser Strategie ein zufällig ausgewählter Block. Diese Strategie dient als untere Schwelle zur Bewertung anderer Ersetzungsstrategien. Unterschreitet diese den hier angegebenen unteren Performanzwert, dann wurden Annahmen über Datenlokalität fälschlich und zu Lasten der Applikation getroffen.

Der Erfolg der einzelnen Ersetzungsstrategien hängt ganz erheblich vom Einsatzbereich sowie von der Struktur und Größe der verwendeten Daten ab. So kommen LRU-Ersetzungsstrategien vorwiegend in Dateisystemen zum Einsatz. Bei Web-Servern schneiden

LFU und FBR dagegen besser ab [ARLI96]. Weitere wichtige Faktoren stellen die Größe und die Aufteilung des Caches dar. Diese Parameter sind in der Regel konfigurierbar und bestimmen häufig über Erfolg und Misserfolg einer Strategie. In den folgenden Abschnitten werden die implementierten Caching-Mechanismen evaluiert. Dabei kam die SIZE-Strategie nur für Messungen, die sich auf den sekundären Cache bezogen, zum Einsatz.

4.3.3 Evaluierung von Caching

Zur Bewertung des Cachings wird die in Gleichung 1 definierte Trefferrate herangezogen. Statt dieser Größe kommt häufig auch der entgegen gesetzte Wert zum Einsatz, welcher die Rate der *Cache Misses* angibt:

$$Miss\ Rate = \frac{\#\ Cache\ Misses}{\#\ Cache\ Requests} \tag{2}$$

Der Erfolg eines Caching-Verfahrens ist normalerweise unabhängig von der eingesetzten Hardware. Das Gleiche gilt für die Verwendung des sekundären Caches. Um die Verfahren zu testen, wurde ein Teil der Multi-Block-Motordaten des AIA verwendet und anhand zweier Ablaufprotokolle evaluiert:

Random: Dieses erste Protokoll führt 250 zufällige Extraktionsbefehle aus. Die Zeitschritte werden ebenfalls nach Zufallsprinzip bestimmt. Angesichts der anfallenden 255 MB Gesamtdatenmenge ist die gesetzte Cache-Größe von 32 MB entsprechend gering gewählt, sodass eine erhöhte Anzahl von *Cache Misses* erwartet werden kann. Da sich hier keine Zugriffsmuster bilden sollten, ist davon auszugehen, dass dieses Ablaufprotokoll den schlechtesten Fall für alle Caching-Strategien darstellt.

Linear: Hier werden ebenfalls 250 Extraktionskommandos abgesetzt. Diesmal liegt jedoch eine lineare Zeitfolge vor, die sich insgesamt siebenmal wiederholt. Die Cache-Größe ist in etwa doppelt so groß, sodass nun einige komplette Zeitschritte in den Cache passen sollten.

Zusätzlich zum Caching wurde *One Block Look-ahead* als System-Prefetching (vgl. Abschnitt 4.4.1.2) aktiviert. Dies sollte nicht gegen die Vergleichbarkeit der verschiedenen Caching-Strategien sprechen. Vielmehr besteht nun noch zusätzlich die Möglichkeit die Annahme zu verifizieren, dass PFBR eigentlich vom Prefetching profitieren sollte. Im Gegensatz zum Caching ist das Prefetching durchaus von der Geschwindigkeit des eingesetzten Systems abhängig, sodass alle Tests auf einer SunFire-6800 durchgeführt wurden. Das Ergebnis beider Testdurchläufe ist in Abbildung 48 dargestellt.

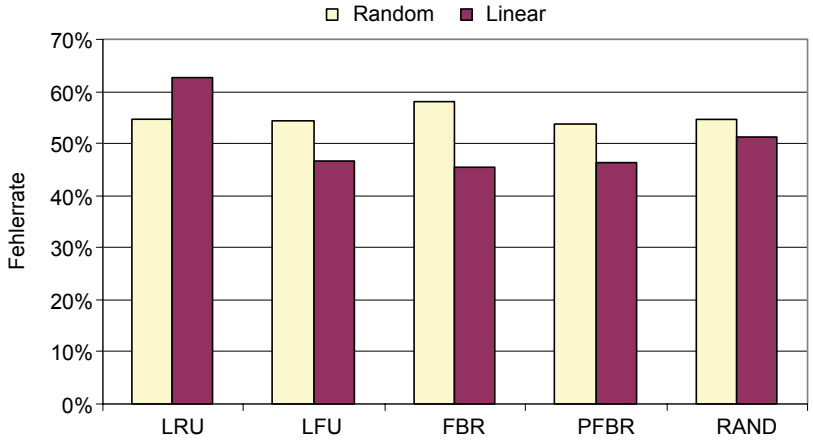

Abbildung 48: Fehlerraten der unterschiedlichen Ersetzungsstrategien

Dabei ist zu berücksichtigen, dass das RAND-Caching als obere Schranke für die Fehlerrate zu bewerten ist. Strategien, die höhere Fehlerraten aufweisen, sind daher schlechter als quasi

gar keine Strategie. In der *Random*-Testreihe kann sich keines der anderen Caching-Verfahren richtig absetzen, FBR liegt sogar über dieser Schranke. Der Grund liegt vorrangig in der relativ kleinen Cache-Größe und der zufälligen Ablaufreihenfolge des Testprotokolls. Dadurch kommt es zu häufigen Ersetzungen. Interessant ist allerdings, dass der priorisierte FBR gegenüber dem einfachen FBR deutlich besser abschneidet. Dies deutet darauf hin, dass sich hier, wie erwartet, eine höhere Anzahl von nützlichen Prefetches eingestellt hat.
Beim Testlauf mit der linearen Abfolge dominieren die frequenzbasierten Ansätze. Besonders der FBR kann überzeugen, wobei die priorisierende Variante keine Vorteile mehr aufweist und etwas schlechter abschneidet. Dass bei einer zyklischen Abarbeitung von CFD-Extraktionskommandos bevorzugt kürzlich verwendete Blöcke im Cache gehalten werden, erweist sich beim LRU-Ansatz als schädlich. Durch die hierfür ungünstig gewählte Cache-Größe kommt es nun dazu, dass ein Block, nachdem er gerade erst entfernt wurde, erneut angefordert wird. Dies lässt sich an der hohen Anzahl von Ersetzungen in Abbildung 49 erkennen.

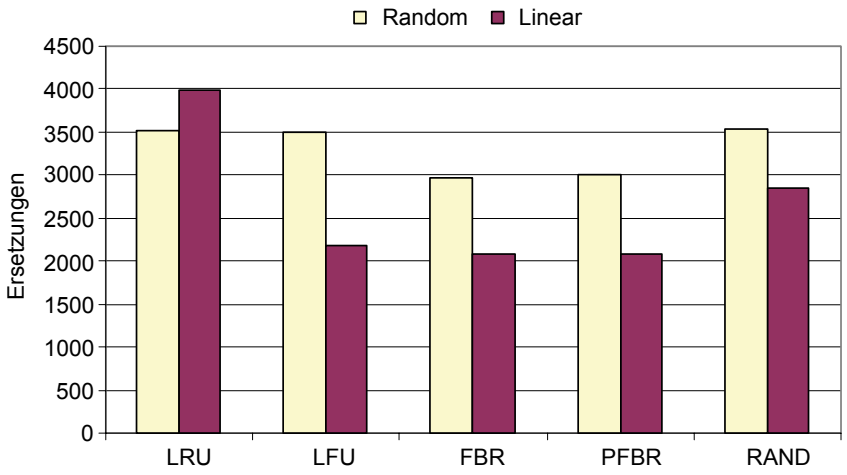

Abbildung 49: Häufigkeit der Blockersetzungen

Die frequenzbasierten Ansätze (LFU, FBR, PFBR) kommen dagegen alle mit relativ wenigen Ersetzungen aus. Obwohl diese Familie von Strategien bei einer fest vorgegebenen linearen Abfolge von Kommandos nicht optimal ist, verweilen die für Treffer sorgenden Blöcke ausreichend lange im Cache.
Für den sekundären Cache sind nur LRU und SIZE als Ersetzungsstrategien implementiert. Da aber hier der Datenbestand und die Trefferrate maßgeblich von der Ersetzungsstrategie des primären Caches abhängen, wurden die beiden für den sekundären Cache vorgesehenen Strategien in Verbindung mit den exemplarischen Strategien FBR und LFU für die erste Cache-Stufe getestet. Der primäre Cache ist wiederum ausreichend klein gehalten, sodass tatsächlich Ersetzungsstrategien für die Auslagerung in die zweite Cache-Stufe sorgten. Auch der sekundäre Cache wies eine Größe auf, die zum Einsatz der zu untersuchenden Ersetzungsstrategien führte. Zum Vergleich wurde dem primären LFU-Cache in einer weiteren Messreihe ein doppelt so großer sekundärer Cache nachgeschaltet. Die Messprotokolle waren für alle drei Testreihen identisch. Die erfassten Ergebnisse sind in Abbildung 50 dargestellt.
Es sind kaum Unterschiede zwischen SIZE und LRU auszumachen. Die in der ersten und zweiten Cache-Stufe zum Einsatz kommenden Ersetzungsverfahren ergänzen sich nicht optimal, sodass es unrelevant ist, welches der beiden Verfahren im sekundären Cache verwendet wird. Eigentlich müsste SIZE durch die Entfernung des jeweils größten Blocks eine höhere Trefferrate aufweisen. Aber durch die Verwendung des Motordatensatzes mit seinen vielen, aber recht kleinen Datenblöcken fällt dies bei den durchgeführten Messreihen nicht ins Gewicht.

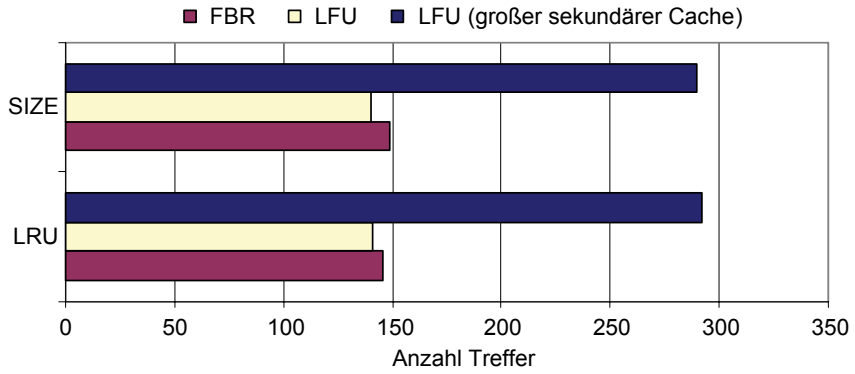

Abbildung 50: Cache-Treffer im sekundären Cache in Abhängigkeit zur angewendeten Strategie der ersten Cache-Stufe

4.4 Prefetching

Beim Caching werden Daten erst dann in den Cache geladen, wenn ein *Cache Miss* aufgetreten ist. Bis das Datum geladen wurde wartet die Applikation. Um diesen Missstand zu beheben, laden Prefetching-Ansätze schon vor der eigentlichen Anfrage die benötigten Daten. Werden sie dann angefragt, liegen sie bereits im Cache vor und die Anwendung kann ohne Performanzverlust weiterarbeiten. Effektives Prefetching muss zur rechten Zeit ausgeführt werden, wirklich benötigte Daten laden und nicht zu viel Overhead verursachen [WIEL96].
Wird das Prefetching durch Software-Komponenten realisiert, also z. B. vom Betriebssystem oder von der Applikation übernommen, dann werden Prefetching-Mechanismen als nebenläufige Prozesse implementiert um den Berechnungsprozess rechtzeitig mit Daten versorgen zu können. Zwar kann der Berechnungsalgorithmus einen expliziten *Fetch*-Befehl absetzen, jedoch blockiert der Prefetching-Prozess nicht die weitere Ausführung der Berechnung. Der Prefetching-Prozess lädt die Daten nebenläufig in den Cache, die somit schon vorliegen können, wenn der Berechnungs-Prozess zum ersten Mal die Daten benötigt. Die Berechnung kann also kontinuierlich ohne unterbrechendes Warten auf zu ladenden Daten durcharbeiten.
Es kann jedoch auch zu unerwünschten Nebeneffekten kommen. Einer dieser Effekte tritt auf, wenn ein prefetchter Datenblock zwar in den Cache geladen aber nicht sofort abgerufen wurde. Wird er dann tatsächlich angefordert, kann er durch Ersetzungsstrategien des Caches schon wieder entfernt worden sein. Ein weitaus gravierender Effekt ist allerdings, dass durch das Einlagern eines prefetchten Blocks ein bereits gecachter Datenblock aus dem Cache entfernt wird, der aber dann direkt angefragt wird. Dieser Effekt heiß Cache-Verschmutzung (engl.: *Cache Pollution*). Ein weiterer Effekt wirkt sich auf das Speichersystem, also z. B. Festplatte oder Hauptspeicher, aus. Wegen der intensiveren Datenvorhaltung werden diese Systeme nun wesentlich stärker frequentiert. Wenn die Bandbreite dieser Komponenten nicht entsprechend ausgelegt ist, kommt es schnell zu einer Überlastung des Speichersystems.
Für die Bewertung des Prefetchings existieren einige Kennwerte [WIEL96]. So gibt die Prefetch-Trefferrate die Anzahl von Datenblöcken an, die nach einem *Fetch*-Befehl auch tatsächlich verwendet wurden. Im Verhältnis zur Gesamtanzahl der Prefetches gesetzt, erhält man die Effizenz. Im Gegensatz dazu sind nutzlose Prefetches diejenigen, die zwar geladen aber nicht angefordert oder zu spät angefordert wurden, also bereits wieder aus dem Cache entfernt wurden. Ein Maß, ob ein *Fetch*-Befehl rechtzeitig oder zu spät abgesetzt wurde, ist die Prefetch-Distanz. Dabei werden die zwischenzeitlichen Ladeschritte nach einem *Fetch*-Befehl angegeben, bis der Datenblock auch tatsächlich angefordert wird. Letztendlich interessiert vor allem die Reduzierung von *Cache Misses*, die durch Prefetching erreicht wurde. Dies wird durch die Deckung (engl.: *Prefetch Coverage*) angegeben und berechnet sich wie folgt:

$$Prefetch\ Coverage = \frac{\#\ Used\ Prefetches}{\#\ Used\ Prefetches + \#\ Cache\ Misses} \quad (3)$$

Neben all diesen Einzelbewertungen des Prefetching-Verhaltens ist nach Gerlhof und Kemper [GERL94] das einzige wirklich aussagekräftige Kriterium allerdings die Laufzeitersparnis. Alle anderen Maßstäbe können zwar eine einzelne Prefetching-Strategie genauer analysieren, durch die große Anzahl von Beeinflussungsfaktoren können sie aber nur bedingt zur Bewertung des wirklichen Nutzens herangezogen werden.

4.4.1 Prefetching-Strategien des VDMS

Die beiden im VDMS integrierten Cache-Stufen werden durch eine Prefetching-Komponente ergänzt, die ebenfalls für den Server und jeden *Proxy* als selbstständige Instanz existiert. Prefetching-Anforderungen sind nicht blockierend, da die Prefetching-Komponente in einem eigenen *Thread* läuft.
Der Prefetcher unterscheidet zwischen Code-Prefetching und System-Prefetching. Jeder Ansatz kann Datenreferenzen zum Prefetchen vorschlagen. Hierfür hält der Prefetcher zwei Listen parat, je eine für die benutzergenerierte Anfrage und für automatische Anfragen, in denen er die jeweiligen Datenreferenzen einsortiert (vgl. Abbildung 51).

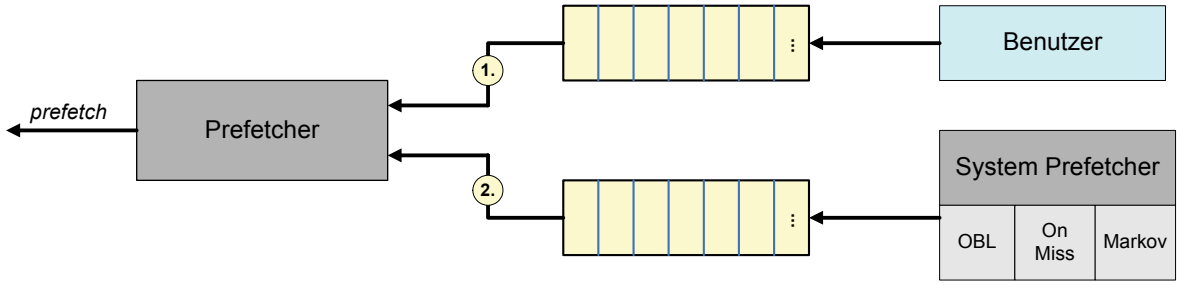

Abbildung 51: Arbeitsweise des Prefetchers

Die Bearbeitung der Listen ist unterschiedlich priorisiert. Da davon ausgegangen wird, dass benutzerdefinierte Anfragen wesentlich genauere Informationen über den verwendeten Datensatz und Algorithmus zugrunde liegen, werden zuerst diese Prefetch-Aufforderungen befolgt. Erst wenn keine Referenzen mehr in der Liste für die Code-Prefetches vorliegen, wird die Liste der System-Prefetches abgearbeitet. Eine Prefetching-Anforderung führt nicht immer zwangsweise zum Laden von Daten. Vielmehr wird zuerst überprüft, ob die vom Prefetcher vorgeschlagene Datenreferenz nicht bereits im Cache vorliegt. Erst wenn dies nicht der Fall ist, werden die Daten in den Cache geladen.

4.4.1.1 Code-Prefetching

Wird direkt aus der Applikation heraus das Prefetching initiiert, dann spricht man von einem Code-Prefetching oder auch software-initiierten Prefetching [WIEL96]. Das Anstoßen des Prefetchings wird innerhalb des Codes durch den nicht-blockierenden Aufruf der *Fetch*-Operation ausgelöst. Hier können neben dem zu ladenden Block weitere Informationen über Art und Verwendung angegeben werden. In der Praxis ist jedoch das code-basierte *Prefetch Scheduling*, also die günstige Platzierung des *Fetch*-Befehls relativ zum Lese- bzw. Schreibzugriff, nur eingeschränkt optimiert durchführbar, da Ausführzeiten zwischen *Fetch*- und Ladebefehl genauso variieren können wie Speicherlatenzen und daher zur Kompilierungszeit nicht vorhersagbar sind. Daher sollte zwischen *Fetch*-Befehl und Datenverwendung eine ausreichende Zeitspanne eingeplant werden. Ansonsten könnte es dennoch zu Wartezeiten kommen.

4.4.1.2 System-Prefetching

In Kontrast zum Code-Prefetching steht das System-Prefetching, das ohne Wissen oder Kontrolle des Anwendungs-Codes voraussichtlich benötigte Daten bereitstellt. Dafür kommt es nicht ohne anzuwendende Strategie aus, da keinerlei Verhaltensweisen und Zugriffsmuster der Applikation bekannt sind. Diese werden beim Starten des Systems voreingestellt, können aber auch zur Laufzeit geändert werden. Da die benötigten Informationen für die einzelnen Strategien unabhängig von der verwendeten Methode ermittelt werden, besteht sogar die Möglichkeit während des Ablaufes einer Berechnung die Strategie zu wechseln. Neben der Auswahl der Strategie besteht aber vom Algorithmus aus keine weitere Einflussmöglichkeit.
Einige geläufige Prefetching-Strategien werden nun vorgestellt:

One Block Look-ahead: Der OBL-Ansatz verwendet die räumliche Lokalität eines angefragten Datums. Wird Block *b* angefordert, lädt OBL automatisch zusätzlich noch den Block *b* + 1 nach. Diese Strategie ist einfach und schnell (Aufwand: *O*(1)) und passt gut zu Datensätzen bzw. Postprocessing-Algorithmen, die sequentiell abgearbeitet werden können. Dies ist bei vielen im Bereich des CFD-Postprocessing vorkommenden Verfahren der Fall.

Prefetch on Miss: Dieser Ansatz ist eine Variation von OBL. Hier wird nur dann der Nachfolgeblock *b* + 1 geladen, wenn der aktuell angefragte Block *b* nicht bereits im Cache vorliegt. Wird die Anfrage vom Cache bedient, dann erfolgt auch kein Prefetching.

Tagged Prefetching: Jeder Block erhält ein Maskierungs-Bit. Wird nun ein Block durch einen *Fetch*-Befehl geladen oder ein durch die Prefetching-Strategie geladener Block zum ersten Mal verwendet, dann wird dieses Flag gesetzt und der Nachbarblock geladen.

Strided Prefetching: Für gewisse Datenstrukturen ist das Laden des räumlich nächsten Blocks ineffizient. Lässt sich ein besseres Ergebnis erzielen, wenn eine größere Schrittweite ausgewählt wird, dann bezeichnet man die Strategie als *Strided Prefetching*. Die Bestimmung der Schrittweite kann zum einen von der Applikation vorgenommen werden und zum anderen implizit aus der Analyse der Zugriffsmuster erfolgen [WIEL96].

Die ersten drei hier vorgestellten Prefetching-Strategien nutzen räumliche Lokalität von Daten aus. Sie berücksichtigen die direkten Nachbarn und gehören daher zur Klasse des sequentiellen Prefetchings. Das Identifizieren der benötigten Blockreihenfolge ist jedoch für Komponenten, die auf der Organisationsschicht von Viracocha angesiedelt sind, nur schwer bis gar nicht möglich. Z. B. besitzen instationäre CFD-Multi-Block-Datensätze eine Topologie, die durch Dateibezeichnungen nur unzureichend bezeichnet wird. Zur Initialisierung kann daher von außen eine Nachbarschaftsbeziehung zwischen zu ladenden Datenblöcken vorgegeben werden.

4.4.1.3 Markov-Prefetching

Ebenfalls zum System-Prefetching zählt eine in das VDMS integrierte Strategie, die auf Wahrscheinlichkeitsgraphen basiert. Diese geben an, mit welcher Wahrscheinlichkeit Blöcke als nächstes benötigt werden. Einer dieser Ansätze wurde z. B. von Curewitz et al. in [CURE93] vorgestellt und basiert auf Analysen im Bereich der Datenkompression. Ähnliche Ansätze verfolgen auch andere Autoren [KROE96, DOUG99]. Der Einsatz für das CFD-Postprocessing ist dagegen neu.
Die Vorhersagen werden dabei basierend auf Wahrscheinlichkeiten getroffen, die sich durch bisher abgefragte Werte berechnen lassen. Dieser Ansatz wird auch als Markov-Prefetching bezeichnet. Hierfür definiert man einen Markov-*Predictor i*-ter Ordnung. Dieser durchsucht die bisherige Datensequenz, die so genannte Markov-Kette, nach Fragmenten, die der Reihenfolge der *i* letzten Anfragen entspricht. Aufgrund der Daten, die den gefundenen Fragmenten folgen, wird eine Vorhersage für den aktuellen Zeitpunkt getroffen. Die

einfachste Variante der ersten Ordnung ermittelt das nächste Datum lediglich auf der Grundlage der letzten Anfrage. Abbildung 52 zeigt ein Beispiel.

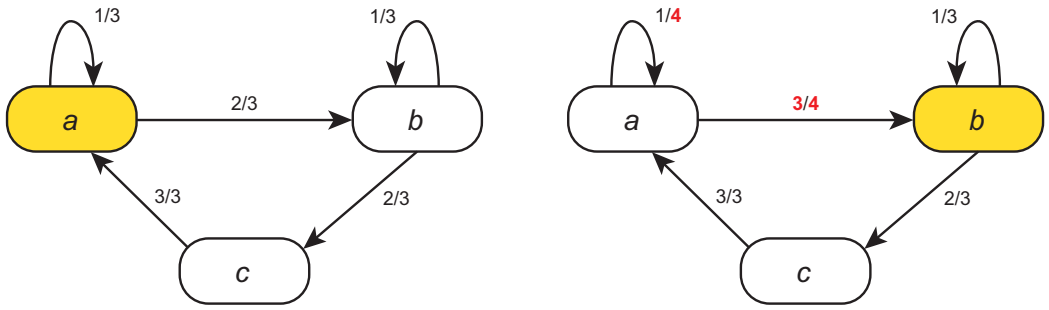

Abbildung 52: Wahrscheinlichkeitsgraph 1. Ordnung der Sequenz {*c,a,a,b,c,a,b,b,c,a*} (links), und nach der darauf folgenden Anfrage von *b* (rechts)

Greift man lediglich auf einen Markov-*Predictor* 1. Ordnung zurück, dann kann man sich das zeitaufwändige Durchsuchen der Abfragesequenz ersparen. Hierfür merkt man sich, wie häufig ein Datum in der Sequenz bereits abgefragt wurde. Das letzte Datum wird dabei nicht mitgezählt. Zudem hält man Nachfolgerzähler, die angeben, wie häufig der jeweilige Nachfolger bereits folgte. Die aktuellen Wahrscheinlichkeiten ergeben sich einfach durch die Division von Nachfolgerzähler durch Datumsabfragehäufigkeit. Steht das nächste Datum fest, werden der eigene Häufigkeitszähler und der Index des tatsächlich aufgetretenen Nachfolgers jeweils um eins erhöht. Für die Beispiel-Sequenz erhält man den in Abbildung 52 (links) dargestellten Wahrscheinlichkeitsgraphen mit entsprechenden Kantengewichtungen.
Im VDMS ist die Konstruktion des Wahrscheinlichkeitsgraphen als Baum realisiert und beruht wiederum auf Informationen aus der Statistik-Einheit. Da es sehr zeitaufwändig werden kann, immer die komplette Markov-Kette von neuem zu durchlaufen, zudem bereits Vorhersagen erster Ordnung recht viel Speicher benötigen können, lässt das VDMS nur eine maximale Kantenanzahl pro Knoten zu. Wird diese überschritten, wird der Baum komplett neu aufgebaut. Hierfür wählt man für die Neukonstruktion ein Fenster der letzten *w* Anfragen aus der Zugriffssequenz aus. Dabei sollte allerdings die Fenstergröße *w* deutlich größer als die maximal zulässige Kantenanzahl gewählt werden. Ansonsten kann es vorkommen, dass der Baum häufig neu aufgebaut werden muss.
Grundsätzlich sollte der Graph jedoch eine möglichst lange Lebenszeit besitzen. Dies hängt mit der Verweildauer von Blöcken im Cache zusammen. Aktuell angefragte Blöcke werden nach dem Laden im Cache gehalten. Nur diejenigen Blöcke, die schon einmal angefragt wurden, werden für eine Wahrscheinlichkeitsvorhersage verwendet. Kommt es nun zu einem *Fetch*-Befehl, dann macht dieser nur Sinn, wenn der ausgewählte Block nicht mehr im Cache verweilt. D. h. also, dass die Lebensdauer des Graphknoten für diese Datenreferenz deutlich länger sein muss als die Verweildauer des Blockes im Cache.
Da der Wahrscheinlichkeitsgraph zur Laufzeit konstruiert wird, kann der Markov-Prefetcher in der Anfangsphase keine ausreichend aussagekräftigen Vorhersagen liefern. Daher wird in dieser so genannte Lernphase zusätzlich auf den OBL-Prefetcher zurückgegriffen. Das Markov-Prefetching wird dabei wie üblich durchgeführt. Kann der Markov-Prefetcher aufgrund fehlender Information keinen Nachfolger bestimmen, kommt automatisch die OBL-Strategie zum Einsatz. Alternativ dazu lässt sich der Wahrscheinlichkeitsgraph beim Start des Prefetchings initialisieren. Damit entfällt die Lernphase. Dieses Wissen kann allerdings nicht vom VDMS selbst sondern höchstens von der Anwendungsschicht kommen.
Basierend auf dem Ergebnis des Markov-*Predictors* muss nicht zwangsläufig nur derjenige Block mit der höchsten Nachfolgewahrscheinlichkeit geladen werden. Vielmehr lassen sich durchaus unterschiedliche Prefetching-Strategien entwickeln. Beispielsweise können alle Daten mit den *k* höchsten Wahrscheinlichkeiten selektiert oder aufgrund der Wahrscheinlichkeitsverteilung Daten zufällig ausgewählt werden.

4.4.2 Evaluierung von Prefetching

Für die Bewertung des Prefetchings diente wiederum eine zufällige Aufrufreihenfolge von Befehlen und Zeitschritten, sowie eine zufällige Kommandoreihe mit einer linearen Zeitfolge (vgl. Abschnitt 4.3.3). Als Datensatz diente erneut der Multi-Block-Motordatensatz. Aktiviert wurde lediglich das System-Prefetching. An dieser Stelle wurde Code-Prefetching hingegen nicht untersucht, da dessen Effizienz ausschließlich durch den Algorithmusentwickler und nicht durch Strategien des VDMS bestimmt wird.
Es zeigte sich, dass wohl die Anzahl der Kommandos noch zu gering war, bzw. der Cache ausreichend groß gewählt wurde, sodass der Markov-Prefetcher 1. Ordnung keinen einzigen Block im Voraus einlud und lediglich bereits gecachte Daten vorschlug. In Kombination mit OBL bestand dieses Problem nicht, sodass diese Variante sogar für die *Random*-Messreihe mit einer Deckung von 39% und einer Reduzierung der *Cache Misses* um 40% das beste Ergebnis lieferte. Bei der zweiten Messreihe mit verstärkt linearer Berechnungsreihenfolge konnten die sequentiellen Prefetching-Strategien OBL und *Prefetching on Miss* besser abschneiden, da sie exaktere Vorhersagen gerade für Isoflächen- und Wirbelextraktionen, die eine statische Datenanforderung aufweisen, liefern können. Wegen der etwas konservativen Vorladestrategie von *Prefetching on Miss*, ist dieser immer etwas schlechter als OBL. Im *Random*-Fall fällt der Unterschied zu OBL, der hier sogar doppelt so viele Cache-Fehler eliminieren kann, recht deutlich aus. Die Messergebnisse sind in Abbildung 53 zusammengefasst.

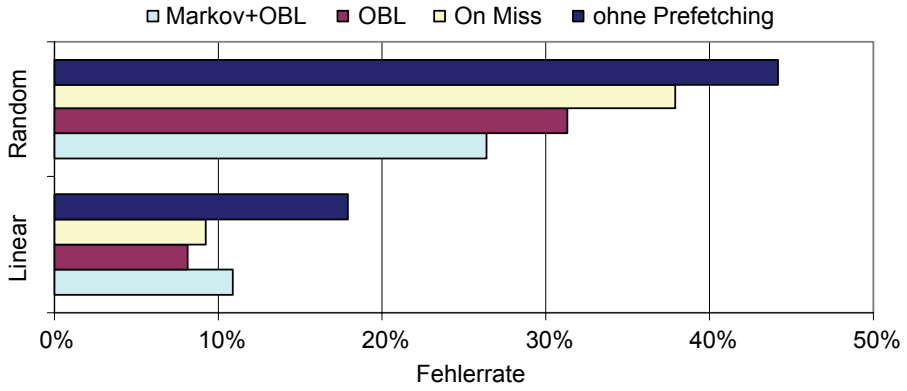

Abbildung 53: Cache-Fehler bei Verwendung von Prefetching-Strategien

Um auch den Markov-Prefetcher auf seine Leistungsfähigkeit hin zu überprüfen, wurde eine dritte Messreihe aufgestellt, bei der 60% der verwendeten Berechnungsbefehle aus Stromlinienextraktionen bestand. Zudem wurde nun der Cache klein gehalten. Hier ließ sich nachweisen, dass sich Markov-Ansätze vor allem beim Einsatz von Partikelverfolgung lohnen. Gerade bei Stromlinien kann der Markov-Prefetcher, nachdem er sich an die Struktur der Datenanfragen angepasst hat, wesentlich bessere Ergebnisse liefern. Die beiden sequentiellen Ansätze erlangen hier höchstens zufällig einmal einen Treffer. Die implementierten Markov-Ansätze für die Partikelverfolgung werden ausführlich in Abschnitt 4.7.1.2 dargestellt.

4.5 Optimierte Ladestrategien

Die bisher beschriebenen Ansätze Caching und Prefetching sorgen dafür, dass sich der Algorithmus aus dem Cache schnell mit benötigten Daten versorgen kann. Aber auch das Befüllen der Caches kann in Abhängigkeit der Hardware und der Systemarchitektur optimiert werden. Dafür werden vom VDMS angepasste Ladestrategien angeboten, die sich aber von System zu System unterscheiden können.
Jeder der *Proxies* hat mehrere Ladestrategien zur Auswahl, mit denen Daten in den primären Cache geladen werden können. Die Entscheidung, welche Strategie verwendet werden soll,

obliegt wieder dem Server. Das hat zur Folge, dass *Proxies* erst auf Antwort warten müssen, bevor sie das Laden fortsetzen können. Auf der anderen Seite weiß der Server als zentrale Instanz genauestens Bescheid über den Inhalt aller *Proxy*-Caches, was durch die Wahl geeigneter Strategien ggf. zu sehr verkürzten Ladezeiten führen kann. Zur Laufzeit kann er somit jederzeit die Strategie wechseln, was als adaptive Strategieentscheidung bezeichnet wird.
Für die Auswahl einer geeigneten Ladestrategie verwendet der Server die so genannte Tauglichkeitsfunktion (engl.: *Fitness Function*). Ausgewählt wird diejenige Strategie, die den höchsten Wert erhält. Die Tauglichkeitsfunktion greift wiederum auf einen Satz von Bewertungskriterien zurück, die die Performanz einer Strategie beurteilen. Ein einfaches Maß ist die durchschnittliche Laufzeit, die alle Phasen einer Strategie erfasst, aber von der Größe der geladenen Daten abhängt. Daher ist es sinnvoller stattdessen die durchschnittliche Bandbreite als Maß zu verwenden. Konkret bedeutet dies, dass die geladene Datenmenge durch die Strategielaufzeit dividiert wird. Weitere Performanzmaße sind denkbar: Anzahl der fehlgeschlagenen Strategieausführungen, explizite Anforderungen durch den Benutzer oder Kommunikations-Overhead zwischen Server und *Proxies*.
Hat sich das Auswahlverfahren eingependelt, führt dies in der Regel dazu, dass nur noch selten eine Strategie geändert wird. Erst wenn der aktuelle Ansatz eine gewisse Bandbreite unterschreiten sollte, bekommen andere Strategien die Gelegenheit eingesetzt zu werden. Daher würde ein gelegentliches Ausprobieren vorhandener Strategien möglicherweise zu einer verbesserten Ausnutzung von Dateisystemen, die durch Umwelteinflüsse schwankende Leistungsdaten aufweisen, führen.

4.5.1 Implementierte Ladestrategien

Es wurde vier Strategien implementiert. Nicht alle stehen immer zur Verfügung. Sind Vorbedingungen nicht erfüllt, werden sie aus der Menge der zu verwendenden Strategien ausgeschlossen. Eine Vorbedingung kann z. B. der direkte Zugriff auf eine zu ladende Datei sein. Ist die Datei lediglich auf einer lokalen Platte eines anderen Knotens vorhanden und somit das direkte Laden nicht möglich, so würden entsprechend betroffene Strategien ausscheiden. Die Bedingungen können sich jedoch zur Laufzeit ändern, sodass nicht berücksichtigte Strategien wieder Gültigkeit erlangen können und bei einer Ladeaktion dann doch noch zum Einsatz kommen.

Load Local: Wird eine Datei angefordert, die im Zugriffsbereich des Prozesses liegt, also auf dem Dateisystem gespeichert ist oder sich im sekundären Cache befindet, dann wird die *LoadData*-Methode des zum Datentyp gehörenden *Data-Access*-Objekts aufgerufen. Somit liegt die Verantwortung für den Ladevorgang beim Entwickler der Zugriffsmethode. Da üblicherweise während des Einlesens von Daten der Aufbau zugehöriger Datenstrukturen vorgenommen wird, ist der hierfür benötigte Zeitaufwand mit in der von *Load Local* erfassten Gesamtzeit enthalten.

Load Manual: Hier übernimmt das VDMS zuerst selber das Laden der Datei in Form einer Byte-Sequenz. Somit entfällt vorerst das Erzeugen von zusätzlichen Datenstrukturen, sodass diese Methode eingesetzt werden kann, um die tatsächliche Zeit für das Datenladen und somit die Dateisysteme und deren Caching-Verhalten zu evaluieren. Aber auch der Einfluss von unterschiedlich großen Dateiblöcken auf das Ladeverhalten lässt sich mit dieser Strategie besser untersuchen.
Danach kann die Byte-Sequenz wiederum dem *Data-Access*-Objekt übergeben werden, um daraus Datenstrukturen zu rekonstruieren und entsprechend zu befüllen.

Transfer Data: Anhand dieser Strategie sollen Daten, die schon in einem Cache eines anderen Knotens vermutet werden, in den eigenen kopiert werden. Daten zwischen verteilten Caches auszutauschen, ist zentraler Bestandteil kooperativer Caches [CORT97, DAHL94]. Liegen diese dort nicht vor, werden sie von der Ladestrategie direkt vom Dateisystem geholt. Da der anfragende *Proxy* die Adresse des Partner-*Proxys* erst

beim Server erfragen muss, dann die Daten vom Partner serialisiert und verschickt werden, um abschließend vom Ziel-*Proxy* wieder deserialisiert werden zu müssen, wird diese Ladestrategie nur dann zum Einsatz kommen, wenn sie trotz dieses Overheads noch immer schneller ist als das direkte Laden vom Dateisystem. Zur Beantwortung dieser Fragestellung spielt die eingesetzte Systemarchitektur eine wesentliche Rolle. So kommt z. B. der Datenaustausch über *Shared Memory* der *Transfer-Data*-Strategie zugute.

Load Collective: Diese Strategie lässt sich einsetzen, wenn alle Prozesse einer *Work Group* zusammen auf dasselbe Datum zugreifen müssen. Hierfür muss das Betriebssystem bzw. die Kommunikationsschnittstelle den gemeinsamen Zugriff unterstützen. Die MPI-Implementierung der Kommunikationsschicht bietet solche Methoden mittels MPI-I/O (vgl. Abschnitt 4.1).

In einer ersten Phase müssen sich alle beteiligten *Proxies* beim Server anmelden, der den gemeinsamen Datenzugriff koordiniert. Erst wenn sich alle *Proxies* einer *Work Group* gemeldet haben, quittiert der Server die Registrierung mit einem Erlaubnis-Flag. Die Koordinationsphase, die einen zusätzlichen Aufwand und Wartezeit bedeutet, sorgt dafür, dass alle Gruppenmitglieder tatsächlich zugreifen wollen und auch dieselbe Datei adressiert haben. Bei einem Fehler kann der Server mit einem Abbruch der Strategie reagieren.

Erst wenn der *Proxy* sein Erlaubnis-Flag erhalten hat, beginnt er mit der zweiten Phase und startet den Ladevorgang mittels Methoden der Kommunikationsschicht. Da hier keinerlei Informationen über Datenformate vorliegen, wird wiederum eine rohe Byte-Sequenz zurückgeliefert. Diese ist dann erneut durch Verwendung des passenden *Data-Access*-Objekts in die gewünschte Datenstruktur umzuwandeln.

4.5.2 Evaluierung von Ladestrategien

Der Erfolg des Cachings (und weitestgehend auch des Prefetchings) hängt überwiegend von Datensatz, Berechnung und Ersetzungsstrategie ab. Dagegen sind die Ergebnisse der Ladestrategien ganz wesentlich durch die zugrunde liegende System-Hardware beeinflusst. Die alles bestimmende Bewertungsgröße ist die Bandbreite, die eine Ladestrategie erreicht. Diese setzt sich nicht nur aus dem Durchsatz des Dateisystems zusammen, sondern berücksichtigt zusätzlich die Laufzeiten, die zum Konvertieren von Daten sowie zur Serialisierung und Deserialisierung von Transferdatenströmen benötigt werden. Des Weiteren wird die Bandbreite durch den Koordinationsaufwand, der zwischen Datenmanager-Komponenten entsteht, vermindert. Die erreichten Bandbreiten auf den drei getesteten Systemen sind in Abbildung 54 dargestellt.

Der Festplattenzugriff auf die binären Datensätze erfolgte jedes Mal über NFS, sodass hier vergleichbare Voraussetzungen gegeben waren. Deutlich erkennbar ist, dass die SunFire-6800 mit sämtlichen Ladestrategien wesentlich höhere Bandbreiten erreicht. Besonders auffällig ist dies bei den beiden direkt auf die Festplatte zugreifenden Strategien *Load Local* und *Load Manual*. Der Einbruch des kollektiven Zugriffs über ROMIO (um einen Faktor 4 langsamer) war zu erwarten, da in [THAK99b] darauf hingewiesen wird, dass ROMIO über NFS weniger performant ist als über parallele Dateisysteme.

Die Auswirkung unterschiedlicher Datenformate wird in Abbildung 55 deutlich. Kommen ASCII-Datensätze zum Einsatz, steigt zum einen die Dateigröße erheblich an, was sich in der Ladezeit niederschlägt. Zum anderen kommt nun aber noch die Zeit hinzu, die benötigt wird um ASCII-Daten in das binäre Speicherformat zu konvertieren. Durch diese Faktoren sind die Bandbreiteneinbrüche bei den auf Festplatten zugreifenden Verfahren drastisch.

Hiervon kann die *Data-Transfer*-Strategie, die keine Festplattenzugriffe durchführt, sondern nur noch Daten zwischen Knoten versendet, profitieren. Hier entfällt die zeitraubende ASCII-Konvertierung, sodass nur die im Speicher vorliegenden Datenstrukturen serialisiert und deserialisiert werden müssen. Die Messungen für den ASCII- und für den binären Fall sind

somit identisch. Bei der SunFire-6800 wurden durch den *Shared-Memory*-Transfer bessere Übertragungsraten erwartet. Bei genauer Analyse kann allerdings festgestellt werden, dass nicht die Datenübertragung sondern die Koordinierung der *Proxies* über den Server der eigentliche Flaschenhals ist (siehe Abbildung 56). 90 Prozent der Laufzeit gehen somit für Kontaktieren, Verbindungsaufbau und Kommunikation verloren.

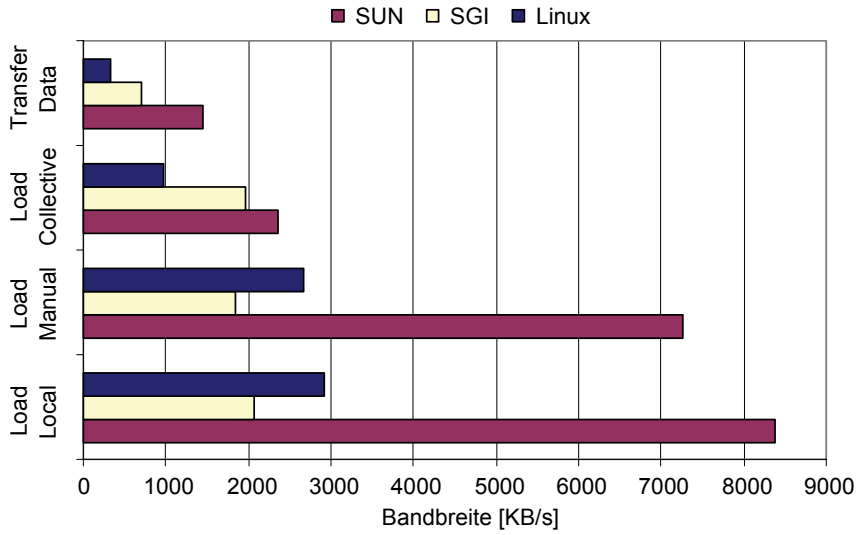

Abbildung 54: Durchschnittliche Bandbreite verschiedener Ladestrategien mit Binärdaten

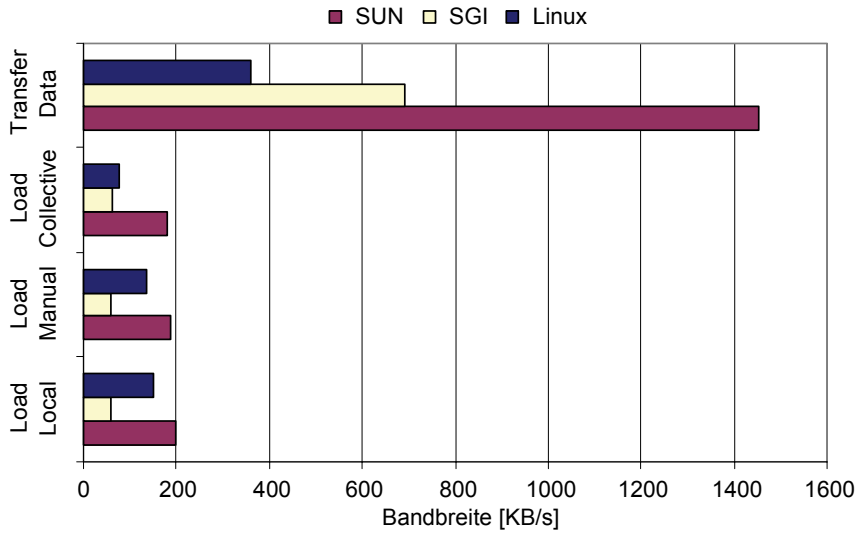

Abbildung 55: Durchschnittliche Bandbreiten verschiedener Ladestrategien mit ASCII-Daten

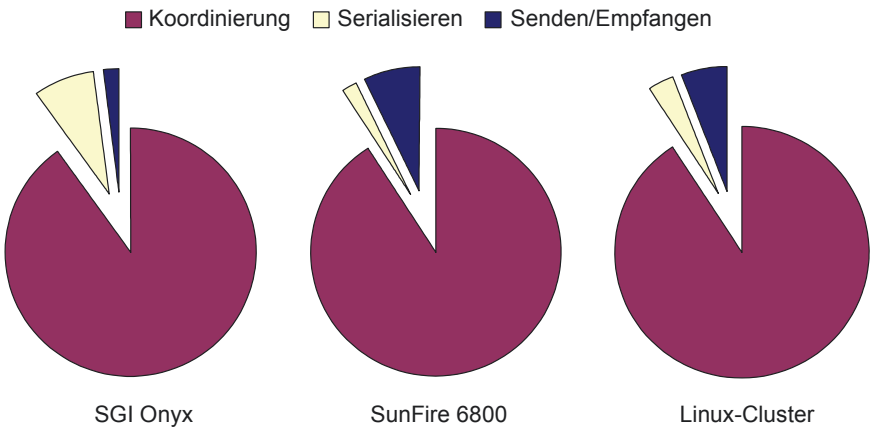

Abbildung 56: Anteile beim Transfer von Daten zwischen Knoten

4.6 Data Service

Die statische Blockzuweisung leidet in der Regel an Lastbalancierungsproblemen. Zwar könnten bei der Zuteilung noch unterschiedliche Blockgrößen berücksichtigt werden, da aber z. B. bei der Isoflächenextraktion in Abhängigkeit des gegebenen Isowertes die Berechnungslast pro Block erheblich variieren kann, bietet sich eine dynamische Blockverteilung an. Der *Worker* fragt in seiner Bearbeitungsschleife daher jeweils vor der Berechnung die Referenz des als nächstes zu bearbeitenden Blocks beim *Scheduler Command* an. Dessen *Data-Service*-Objekt, das eine Liste der noch zu bearbeitenden Blöcke verwaltet, erkundigt sich via Datenmanager-Server über den Inhalt der einzelnen *Proxy*-Caches. Liegen beim anfragenden *Worker* entsprechende Daten vor, werden diese zur Bearbeitung zugewiesen. Erst wenn keine gecachten Daten verwendet werden können, werden sonstige unbearbeitete Blöcke zugewiesen. Der grobe Ablauf aus Sicht des Algorithmus ist als Pseudocode in Listing 2 dargestellt.

```
While (DataService::DataBlocksUnprocessed())
{
      Block b;
      b = DataService::GetNextBlock ();
      ExtractData (b);
}
```

Listing 2: Pseudocode für dynamische Merkmalsextraktion

Eine geeignete Methode zur Bewertung von Lastbalancierungen ist die Standardabweichung der Einzellaufzeiten aller *Worker*. Sie ist folgendermaßen definiert:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{4}$$

Mit einem Erwartungswert μ und den n Einzelwerten x_i erhält man ein recht gutes Maß für die Streuung.
Um den *Data Service* entsprechend beurteilen zu können, wurde die Standardabweichung beim Einsatz von 8 *Workern* berechnet. Dabei kamen zum einen die Isoflächen- und zum anderen die Wirbelmantelextraktion, jeweils mit und ohne verwendetem *Data Service*, zum Einsatz. Die Werte sind in Abbildung 57 aufgetragen.

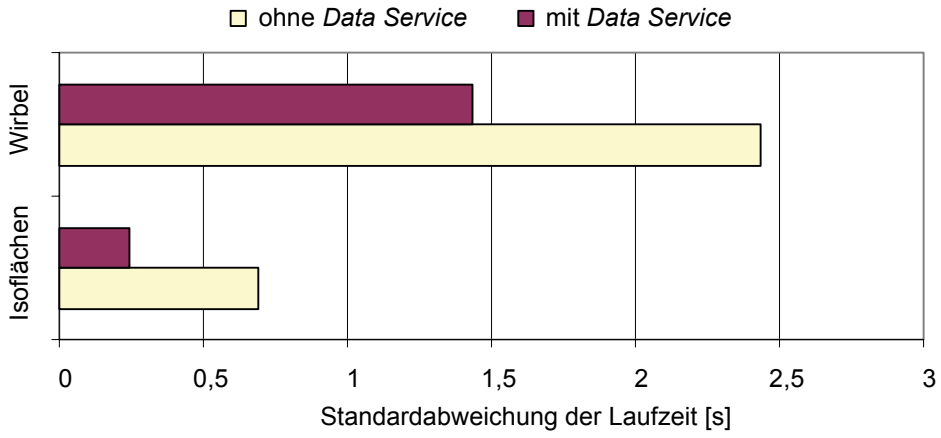

Abbildung 57: Die Standardabweichung bei Isoflächen- und Wirbelextraktion sinkt bei dynamischer Verteilung merklich

Es ist ersichtlich, dass der *Data Service* die Standardabweichung deutlich senkt. Das Ergebnis ist eine ausgeglichenere Berechnungslast zwischen den *Workern*. Als Kosten muss allerdings der Mehraufwand an Kommunikation erwähnt werden.

Trotz der Vorteile einer besseren Lastverteilung und einer optimalen Ausnutzung lokaler Caches, treten Probleme bei gleichzeitiger Verwendung von Prefetching-Strategien auf. Einfache Prefetching-Ansätze, wie Code-Prefetching und OBL, schlagen bei einer dynamischen Verteilungsstrategie fehl, da nun keine sinnvolle Vorhersage mehr über den als nächstes zugewiesenen Block getroffen werden kann. Daher wird man üblicherweise Prefetching und *Data Service* nicht kombiniert einsetzen.

4.7 VDMS-optimierte CFD-Algorithmen

In diesem Abschnitt soll nun aufgezeigt werden wie effizient die beschriebenen Funktionalitäten des *Viracocha Data Management Systems* im Vergleich zu einzelnen CFD-Extraktionsverfahren sind. Für die Evaluierung wurden exemplarisch Isoflächen, Wirbelregionen, sowie Strom- und Bahnlinien ausgewählt. Als Testdatensätze wurden die instationären Multi-Block-Datensätze *Engine* und *Propfan* sowie der ebenfalls in mehreren Zeitschritten vorliegende rectilineare Datensatz *Shock* eingesetzt (vgl. Anhang A). Die in Anhang B beschriebene Hardware diente auch hier wiederum als Systemarchitekturgrundlage.

4.7.1 Isoflächenextraktion mit dem VDMS

Bevor auf das Verhalten der Isoflächenextraktion unter Verwendung unterschiedlicher VDMS-Funktionalitäten ausführlich eingegangen wird, werden zuvor zwei applikationsgesteuerte Aspekte genauer dargestellt: Code-Prefetching und Intervallbäume. In den meisten Fällen besteht zwar keine Notwendigkeit Extraktionsalgorithmen für den Einsatz des VDMS zu modifizieren. Einige spezielle Funktionen lassen sich jedoch nur von außen steuern, da sie algorithmus- oder datensatzspezifisches Wissen voraussetzen. Neben den im Folgenden beschriebenen Ansätzen gehört zum Beispiel auch der im vorangegangenen Abschnitt beschriebene *Data Service* hierzu.

4.7.1.1 Code-Prefetching

Auch das System-Prefetching kann wesentlich durch die Anwendungsschicht beeinflusst werden. Werden einem Knoten vor dem Start der Berechnung die Datenblöcke statisch zugewiesen, kann eine Reihenfolge der Datenanforderung lokal bestimmt und sodann für Prefetching ausgenutzt werden. Die Datenreihenfolge lässt sich dem System für OBL-Prefetching übergeben. Das optimale System-Prefetching wird anschließend automatisch vom VDMS durchgeführt, sodass der eigentliche Berechnungsalgorithmus nicht verändert werden muss. Alternativ kann die Anwendung aber auch durch Code-Prefetching den Zeitpunkt zum Auslösen von *Fetch*-Befehlen selbst bestimmen. Ein entsprechender Pseudocode für Isoflächenextraktion ist in Listing 3 angedeutet.

```
For All (DataBlocks b)
{
      LoadBlock (b);
      Prefetch  (b + 1)
      ExtractIsosurface (b);
}
```

Listing 3: Pseudocode für die Verwendung von Code-Prefetching und Isoflächenextraktion

Sowohl das System-Prefetching als auch das Code-Prefetching können nur dann erfolgreich sein, wenn der Berechnungsaufwand ausreichend ist, um im Hintergrund bereits den nächsten Block in den Cache zu laden. Daher wird das Vereinen der extrahierten Isoflächenfragmente direkt nach der Bearbeitung eines einzelnen Blocks durchgeführt, was die Dauer eines Verarbeitungsschrittes erhöht. Erst dann beginnt der nächste Durchlauf der Berechnungsschleife.

4.7.1.2 Intervallbäume und Datensatzzerlegung

Eine weitere Optimierung lässt sich durch Intervallbäume [CIGN96, BERG00] erreichen. Solche Datenstrukturen geben an, welche Skalarbereiche durch einen Block abgedeckt werden. Somit erhält man eine einfache Heuristik, ob für einen gegebenen Isowert überhaupt Isoflächen in einem Datenblock auftreten können. Liegt der Isowert innerhalb des abgedeckten Zahlenbereichs, wird der Block, in Analogie zur aktiven Zelle, als aktiver Block bezeichnet.
Mithilfe von Intervallbäumen kann nun bereits vor der Zuteilung entschieden werden, ob der Block überhaupt Ergebnisse liefert. Von der Vorselektierung kann sowohl die dynamische als auch die statische Blockverteilung profitieren. Die Ladelast kann dadurch erheblich gesenkt werden, wie am Beispiel des Turbinen-Datensatzes des DLR in Abbildung 58 zu erkennen ist. Bei entsprechender Wahl des Isowertes müssen Großteile der Datenblöcke erst gar nicht geladen werden, wodurch sich die Berechnung beschleunigt.

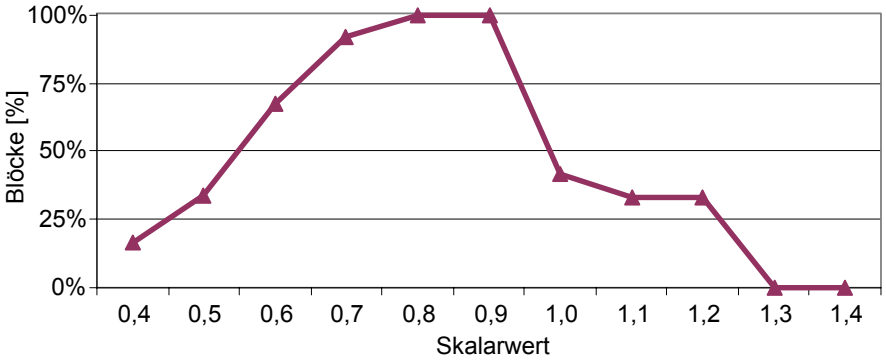

Abbildung 58: Aktive Blöcke des Druckskalarfelds im Turbinendatensatz

Dass sich das Verhalten des Datenmanagements mit der Partitionierung und der unterschiedlichen Größen der Einzelblöcke verändert, ist ebenfalls nahe liegend. Um die Effizienz von Intervallbäumen in Abhängigkeit der Granularität der Zerlegung eines Datensatzes untersuchen zu können, wurde der *Shock*-Datensatz herangezogen. Mit 61 MB pro Zeitschritt ist er ausreichend groß, um exemplarisch einen dieser Blöcke herauszugreifen und entlang der *i-j-k*-Achsen in lauter einzelne rectilineare Blöcke zu zerschneiden. Somit entstanden vier weitere Multi-Block-Datensätze, die mit XL (66 Blöcke), L (217 Blöcke), M (313 Blöcke) und S (798 Blöcke) bezeichnet wurden. Dabei ist zu berücksichtigen, dass an den Übergängen zwischen zwei Blöcken die gemeinsamen Punkte nun doppelt vorliegen, sodass mit jeder weiteren Unterteilung die Gesamtgröße immer etwas ansteigt. Zudem entsteht mit der zunehmend feineren Blockzerlegung ein erhöhter Bearbeitungsaufwand für diejenigen Ansätze, die Blöcke vor der Berechnung zuerst zusammenführen, bzw. nach der Berechnung die Einzelergebnisse zusammensetzen müssen.

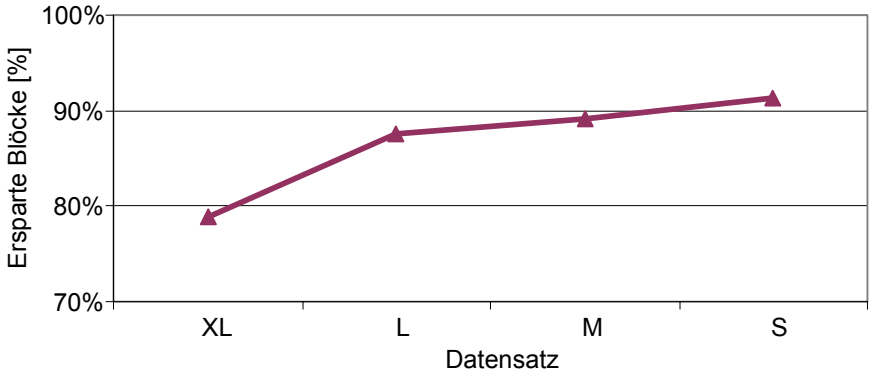

Abbildung 59: Intervallbäume und feinere Aufteilung erhöhen die Anzahl eingesparter Blöcke

Neben diesen Nachteilen liegen die Vorteile jedoch klar auf der Hand. Werden beispielsweise für die verschiedenen Auflösungen Intervallbäume erzeugt, um anschließend Isoflächen zu berechnen, dann gelten in Abhängigkeit des gewählten Isowertes deutlich weniger Teilblöcke als aktiv und müssen in den Speicher geladen werden (Abbildung 59).
Umgekehrt wird nun der Cache stärker belastet. Mit zunehmender Blockanzahl steigt die Fehlerrate. Auch das Prefetching wird jetzt stärker frequentiert und liefert häufiger nicht verwendbare Blöcke, sodass die Prefetch-Deckung abnimmt. Diese beiden Phänomene sind in Abbildung 60 erkennbar.

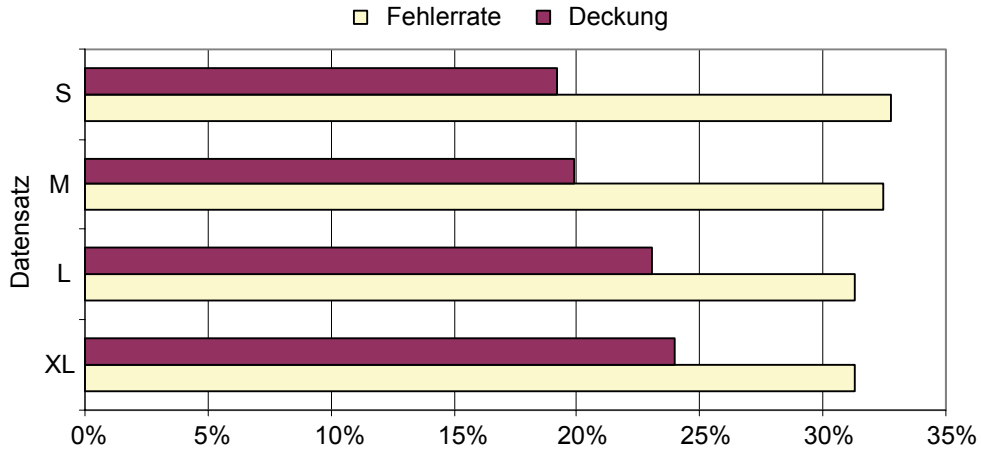

Abbildung 60: Steigende Fehlerrate mit zunehmender feinerer Granularität von Datensätzen

Allerdings steigt mit zunehmender Blockanzahl die Anzahl der nützlichen Prefetches nur relativ langsam an (Abbildung 61). Durch die kleinen Blöcke der Datensätze M und S verkürzen sich die Berechnungszeiten dermaßen, dass der im Hintergrund arbeitende Prefetcher zunehmend in Schwierigkeiten kommt, rechtzeitig die benötigten Daten in den Cache zu laden. Das Prefetching wird also ineffizienter, sodass es sinnig ist die Blockgröße genügend groß zu wählen, damit das Datenmanagement noch die Gelegenheit hat mit seinen Mechanismen hilfreich eingreifen zu können.

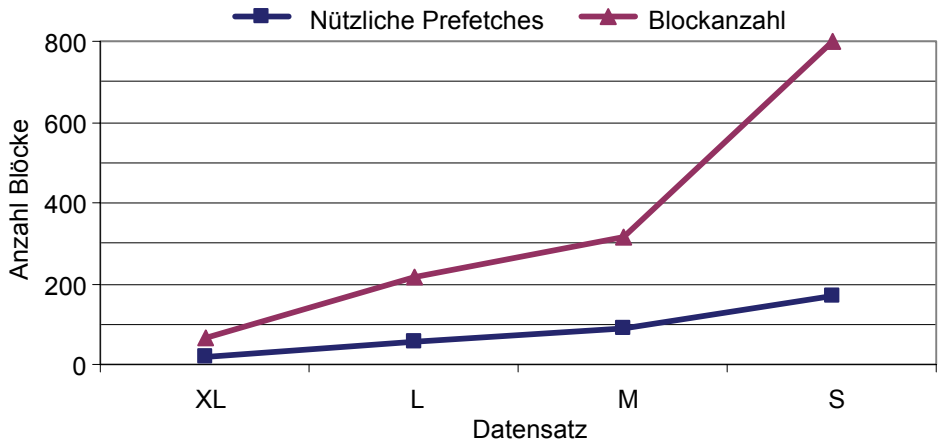

Abbildung 61: Anzahl nützlicher Prefetches wächst bedeutend langsamer als die Anzahl der Blöcke im Datensatz

Als Fazit lässt sich festhalten, dass sich der Aufwand durchaus löhnt eine optimale Datensatzzerlegung, die sich als Kompromiss zwischen Blockgröße und Blockanzahl definiert, zu ermitteln und anschließend entsprechende Intervallbäume für die Isoflächenextraktion zu berechnen.

4.7.1.3 Evaluierung von Isoflächenextraktion und VDMS

Nachdem die in das Datenmanagement von Viracocha integrierten Mechanismen bereits einzeln dargestellt und evaluiert wurden, steht nun das eigentliche Ziel, nämlich die effektivere Durchführung des CFD-Postprocessings, im Mittelpunkt der Betrachtungen. Hierfür wurden verschiedene Messreihen erstellt, die jeweils unterschiedliche Komponenten vom VDMS verwenden:

Ohne: Die hiermit durchgeführten Laufzeitmessungen laufen ohne aktiviertes Datenmanagement. Daher stellen die so erfassten Werte den Referenzrahmen aller anderen Messungen dar.

Cold: Zur Startzeit der Laufzeitmessungen sind Daten- und Namens-Caches leer. Sie können somit am Anfang noch nicht behilflich sein. Erst die zur Laufzeit gecachten Daten lassen sich dann für einen dadurch erhofften zunehmenden *Speed-up* verwenden.

***Prefetching* (Strategie)**: Hier handelt es sich ebenfalls um ein Kaltstartverhalten des Datenmanagements. Allerdings kommt nun noch zusätzlich System-Prefetching zum Einsatz, das im Hintergrund bereits Daten in den Cache lädt. Die verwendete Strategie wird in Klammern angegeben.

Data Service: Für Isoflächen- und Wirbelextraktionen kommt die in Abschnitt 4.6 beschriebene *Data-Service*-Komponente zum Einsatz. Diese beim Server angesiedelte Einheit bestimmt den als nächstes zu verwendenden Block, wobei der jeweilige *Proxy*-Cache berücksichtigt wird. Zum Startzeitpunkt der Messungen sind die Caches jedoch wiederum leer.

Cached: Diese Laufzeitmessung erfasst das Berechnungsverhalten, nachdem dieselbe Extraktion mit den gleichen Daten bereits vorher schon einmal durchgeführt wurde. Der Cache-Speicher ist ausreichend groß gewählt worden, sodass nun alle benötigten Daten im primären Cache liegen und der Namens-Cache die notwendigen Namens referenzen aufweist. Diese Messreihe stellt die optimale Grenze für den Laufzeitgewinn durch das Datenmanagement dar.

Zwei Isoflächen wurden über mehrere Zeitschritte unter Verwendung des Motordatensatzes im ACSII-Format extrahiert. Auf einem SunFire-6800-Knoten gemessen (Abbildung 62, Laufzeiten relativ zur Ohne-Messreihe) ist zu erkennen, dass der Kaltstart nicht von den schnelleren Ladestrategien profitieren kann. Zum Teil braucht diese Messreihe länger als die ohne VDMS-Unterstützung. Wird zusätzlich das Prefetching im Hintergrund aktiviert, ergibt sich hingegen eine deutliche Laufzeitersparnis von 11% bis 23%. Wie der gecachte Fall zeigt, ist der Algorithmus ohne Datenladen enorm schnell. Der *Data Service* kann erwartungsgemäß bei leeren Caches keinen zusätzlichen *Speed-up* bewirken.

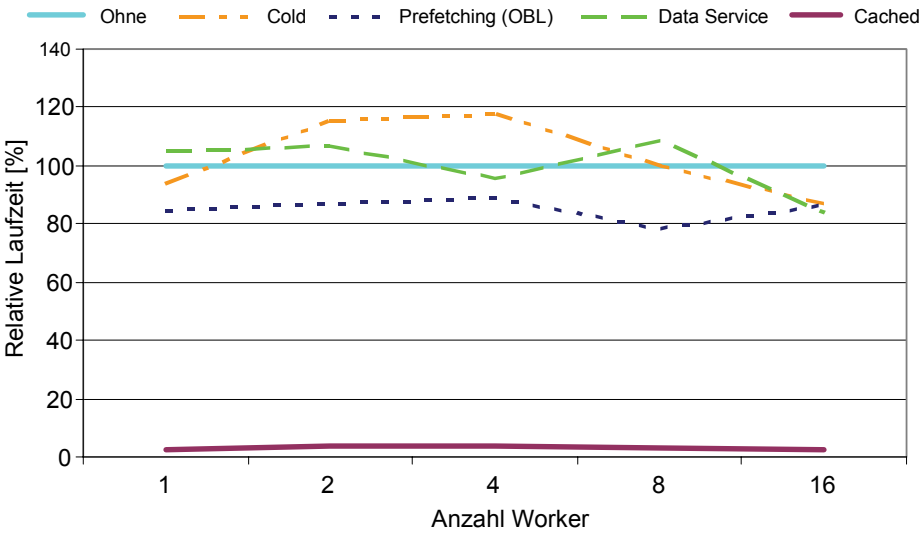

Abbildung 62: Isoflächenextraktion auf einer SunFire 6800

Im Gegensatz zu den Messungen auf der SunFire, kann auf dem Linux-Cluster kein merklicher Vorteil durch das VDMS erreicht werden. Dies liegt zum einen an der niedrigen Berechnungslast und zum anderen an der gegenüber der SunFire höheren Kommunikationlast. Alle Messreihen, mit Ausnahme der Werte für den gecachten Fall, liegen beieinander, wie Abbildung 63 zeigt.

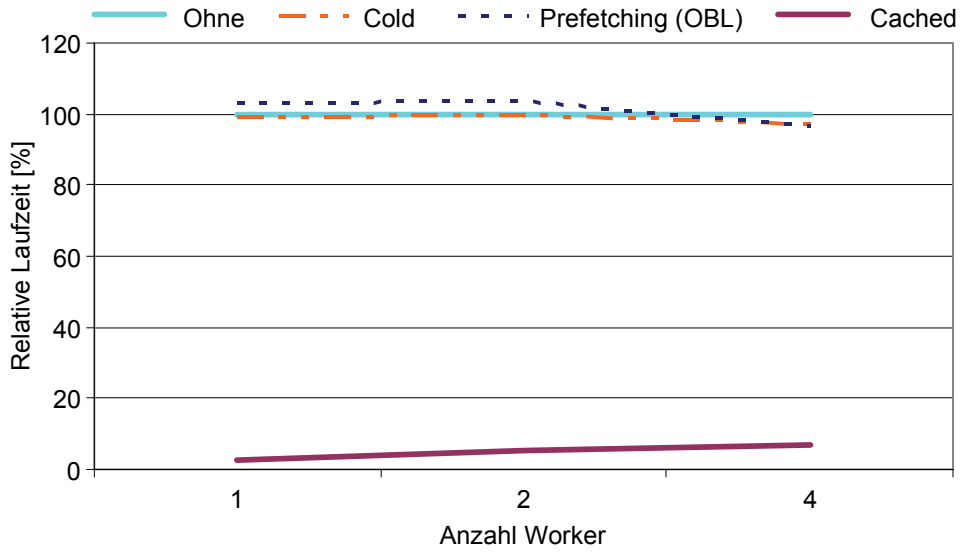

Abbildung 63: Isoflächenextraktion auf einem Linux-Cluster

4.7.2 Wirbelextraktion mit dem VDMS

Im Gegensatz zur Extraktion von Isoflächen steht die Verwendung von Intervallbäumen für die Wirbelmantelextraktion nicht zur Verfügung. Dies wäre nur möglich, wenn diese Metadaten in einem Präprozess für jede Wirbelmethode berechnet und dann auf Festplatte abgespeichert worden wären. Da manche Wirbelextraktionsverfahren durchaus variable Kontrollparameter aufweisen und somit Bäume für jede Einstellung erstellt werden müssten, erweist sich dieser Ansatz hier nicht immer als praktikabel.

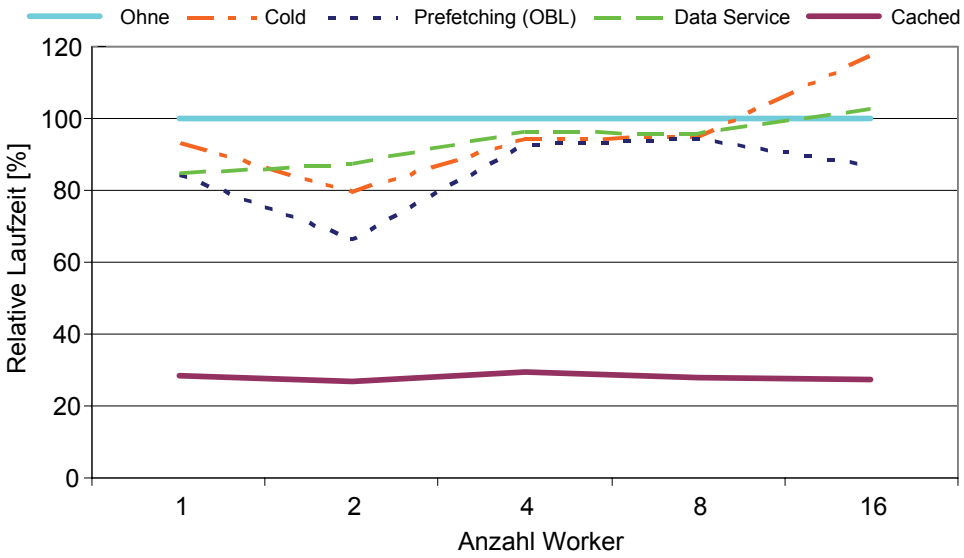

Abbildung 64: Normalisierte Helizität auf der SunFire 6800

Für die Evaluierung des VDMS wurde die Wirbelextraktion nach der Methode der normalisierten Helizität [DEGA90] mit einem Schwellwert von 0.95 ausgewählt. Grundlage war der im ASCII-Format abgespeicherte Motordatensatz. Durchgeführt wurden die gleichen Laufzeitmessungen wie sie bereits für die Evaluierung des VDMS bei Isoflächenextraktion

zum Einsatz kamen. Da die Extraktion von Wirbelregionen wegen der vorgeschalteten zellbasierten Berechnungen mit abschließender Isoflächenextraktion deutlich aufwändiger ist als die bloße Isoflächenberechnung, fällt nun der Anteil der Ladelast an der Gesamtlaufzeit deutlich niedriger aus.
Daher ist die nun auftretende zusätzliche Laufzeit für das Prefetching von besonderem Interesse. In Abbildung 64 sind die Messreihen aufgetragen, die auf der SunFire-6800 durchgeführt wurden. Obwohl die Laufzeiteinsparungen nun stärker schwanken, sind die Prefetching-Ergebnisse mit bis zu 34% deutlich schneller. Messungen auf dem Linux-Cluster (Abbildung 65) zeigen nun ebenfalls einen merklichen Gewinn beim Einsatz von Prefetching (5% bis 10%).

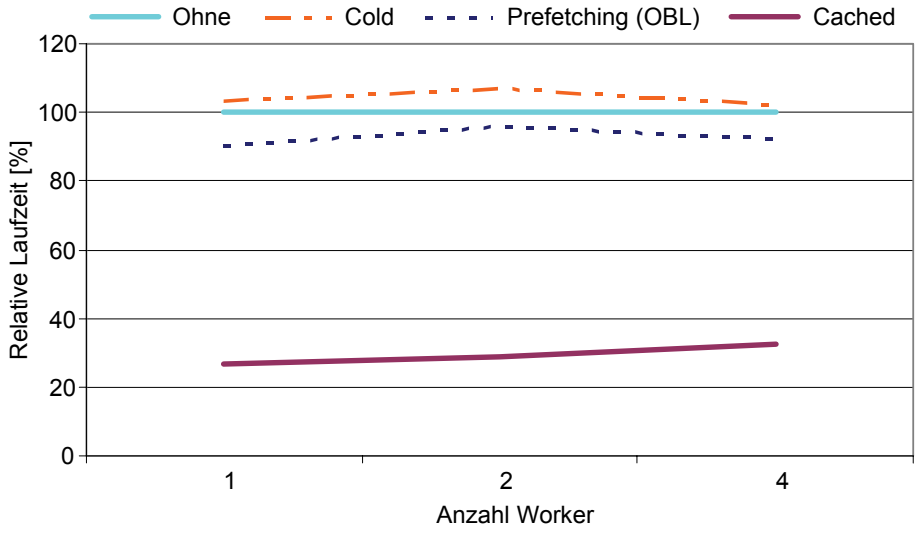

Abbildung 65: Wirbelextraktion auf dem Linux-Cluster

Da nun ein höherer Berechnungsaufwand besteht, kann auch die Messreihe mit gecachten Daten noch vom Einsatz mehrerer Prozessoren profitieren und skaliert deutlich besser gegenüber der Isoflächenextraktion. Auch der *Data Service* und die Kaltstart-Messreihe skalieren besser und können beim Einsatz auf der SunFire sogar anfänglich eine Laufzeitverbesserung bewirken, bis sich dann eine typische Sättigung einstellt.

4.7.3 Strom- und Bahnlinienberechnung mit dem VDMS

Der Haupteinsatzbereich des Markov-Prefetchers ist die Partikelverfolgung. Zum Startzeitpunkt der Partikelintegration ist nicht bekannt, wie viele und in welcher Reihenfolge Datenblöcke geladen werden müssen. Daher erfolgt die Datenzuweisung dynamisch während der Berechnungslaufzeit, wobei der tatsächlich benötigte Block vom Algorithmus angefordert wird.
In der Lernphase, in der noch kein Wahrscheinlichkeitsgraph zur Verfügung steht, kommt neben der reinen Markov-Variante als alternative Strategie der OBL-Prefetcher zum Einsatz. Da ein Partikelverlauf nicht zwangsweise einer vorgegebenen Blockreihenfolge folgen muss, kann es dazu kommen, dass OBL vorwiegend ungeeignete Vorschläge für die Partikel-integration liefert. Dann bleibt das Prefetching während der Lernphase weiterhin wenig effektiv. Daher wurde in Viracocha die Möglichkeit eingebaut, die Lernphase durch einen initialen Wahrscheinlichkeitsgraphen zu ersetzen. Diese Initialisierung geht jedoch von einem Detailwissen über Einsatzdomäne und Datenstruktur aus, sodass die Algorithmusschicht von Viracocha den Markov-Prefetcher entsprechend mit Informationen versorgen muss. Folgende Ansätze sind hierfür implementiert worden:

Größe der *Connection Windows*: Multi-Block-Datensätze weisen bereits Topologien auf, die auch als Graphen dargestellt werden können (vgl. Abschnitt 3.4.2). Ein nahe liegender Ansatz verwendet die Größen der Verbindungsfenster zwischen zwei Blöcken als

Anhaltspunkt für Wahrscheinlichkeiten. Diese stellen die Kantengewichte im Graphen dar.

Stromlinien-Stichproben: Dieser Ansatz beruht ebenfalls auf der MB-Topologie. Dabei werden die jeweiligen Wahrscheinlichkeiten, dass ein in einem Ausgangsdatenblock befindliches Partikel in dessen Nachblöcke gelangt, ermittelt. Hierfür werden in einem Präprozess eine Anzahl von Saatpunkten nach einer zuvor bestimmten Heuristik im zu untersuchenden Block verteilt und anschließend Stromlinien berechnet, bis diese den Block verlassen. Die IDs der betretenen Nachbarblöcke werden gezählt und für die Konstruktion des Graphen verwendet.

Die vorgestellte Stichproben-Strategie lässt sich auch abändern. Statt zufällig Saatpunkte im Block zu bestimmen, könnte eine einzige zufällige Probe pro Zelle erfolgen oder jeweils ein Saatpunkt in die Mitte (bezüglich des *C-Space*, vgl. hierzu Abschnitt 3.4.2) einer Zelle gesetzt werden. Weitere Varianten sind vorstellbar.

Zeitlevel-Gewichtung: Im instationären Fall, wenn beispielsweise Bahnlinien berechnet werden sollen, sind auch Blöcke des nächsten Zeitschrittes zu berücksichtigen. Der implementierte Ansatz verwendet trotzdem den Stromlinien-Initialisierungsgraphen, setzt dann jedoch eine zusätzliche Gewichtung für den nächsten Zeitschritt. Die Zwischenblockwahrscheinlichkeiten müssen entsprechend reduziert werden. Ein Beispiel ist in Abbildung 66 (vgl. auch Abbildung 20) gegeben.

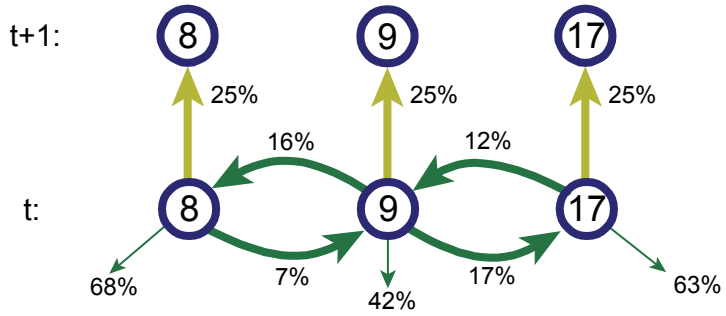

Abbildung 66: Ausschnitt aus dem Initialisierungsgraphen des Motordatensatzes mit Angabe der Zeitlevel-Gewichtung von 25% und der diesbezüglich reduzierten Restwahrscheinlichkeiten

Bei diesem Ansatz wird davon ausgegangen, dass sich die Block-ID beim Zeitlevel-Wechsel nicht ändert. Beim Einsatz von MB-Datensätzen, bei denen sich die Blöcke zwischen den Zeitleveln nicht verändern (Bsp.: Nasendatensatz), ist dies gegeben. In so genannten *Moving Grids* (Beispiele: Motor, Turbine) ist diese Annahme für eine Wahrscheinlichkeitsabschätzung in der Regel jedoch ebenfalls ausreichend. Üblicherweise ändert sich nämlich die Vernetzung des Strömungsfelds einer instationären CFD-Simulation zwischen zwei Zeitschritten nicht erheblich.

Zeitlevel-Sprung: Der Stromlinien-Initialisierungsgraph wird auch hier unverändert übernommen, jedoch wird die selektierte Block-ID dazu verwendet bereits den Block vom nächsten Zeitlevel zu laden. Abbildung 67 zeigt wiederum einen Ausschnitt aus dem entsprechenden Initialisierungsgraphen des Motordatensatzes.

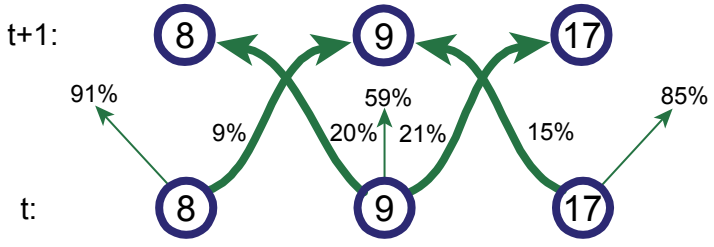

Abbildung 67: Der Markov-Prefetcher liefert beim Zeitlevel-Sprungverfahren immer einen Block vom nächsten Zeitlevel

Das Zeitlevel-Sprungverfahren lässt sich auch auf den Initialisierungsgraphen anwenden, der sich durch die Größenbestimmung der *Connection Windows* ergibt.

Stromlinien-Initialisierungsgraphen auch für die Bahnlinienberechnung zu verwenden, ist deswegen gerechtfertigt, da grundsätzlich auch Stromlinien einen gewissen Trend für instationäre Partikelverfolgung liefern sollten, da sich das Geschwindigkeitsfeld in zwei aufeinander folgenden Zeitleveln üblicherweise nicht völlig ändert. Dennoch weisen die beiden Strategien, die den Stromlinien-Initialisierungsgraphen für Bahnlinienvorhersagen verwenden, recht starre Zeitwechselannahmen auf. Bessere Ergebnisse könnten Bahnlinien-Initialisierungsgraphen liefern.
Eine Alternative zum System-Prefetching kann Code-Prefetching, das ebenfalls die MB-Topologie ausnutzt, darstellen. Nähert sich ein Partikel den Grenzen eines Blocks, kann das nächstliegende *Connection Window* und somit ein möglicher nächster Block bestimmt werden. In die Bestimmung kann auch die Richtung der Partikelbahn mit einfließen. Der so bestimmte Block lässt sich sodann anwendungsinitiiert prefetchen. Der Bereich, ab dem der Prefetching-Vorgang angestoßen wird, sollte nicht allzu groß gewählt sein, um nicht bei jedem Zellwechsel innerhalb des Blocks einen erneuten *Fetch*-Vorgang auszulösen. Mit Code-Prefetching kann auch wesentlich exakter auf anstehende Zeitlevelwechsel reagiert werden, da während der Partikelverfolgung die exakte Integrationszeit vorliegt. Diese algorithmusspezifischen Informationen sind dem VDMS unbekannt und stehen somit System-Prefetchern nicht zur Verfügung.

4.7.3.1 Evaluierung von Bahnlinienberechnung mit Markov-Prefetching

Um eine Auskunft über das Verhalten des Markov-Prefetchers beim Extrahieren von Partikeln zu erhalten, wurden im Motordatensatz 64 Bahnlinien in neun aufeinander folgenden Zeitschritten berechnet. Hierfür kam ein adaptives Runge-Kutta-Verfahren 4. Ordnung zum Einsatz [GERN05]. In der Initialisierungsphase wurde zusätzlich auf OBL zurückgegriffen. Für die Bewertung wurden zudem die Bahnlinien zum einen ohne aktivem VDMS (Messreihe *Ohne*) und zum anderen mit gecachten Datenblöcken (Messreihe *Cached*) berechnet. In Abbildung 68 sind die gemessenen Ergebnisse dargestellt.

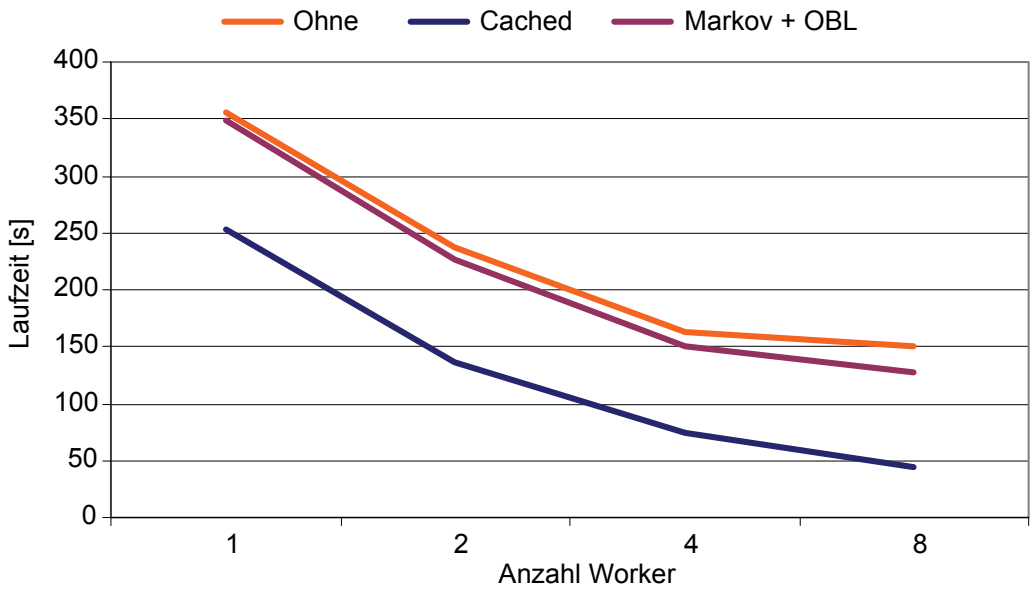

Abbildung 68: Berechnung von 64 Bahnlinien auf der SunFire 6800

Auffällig ist, dass trotz Verteilung der Berechnungslast die Ladelast fast konstant bleibt. Dadurch steigt deren Anteil an der Gesamtlaufzeit von unter 30% auf über 70% an, sodass auch das Prefetchings erheblich an Bedeutung gewinnt. Obwohl der Markov-Ansatz im

seriellen Fall (1 *Worker*) kaum Vorteile aufweist (was ein Hinweis darauf ist, dass hier Ladeempfehlungen zu spät kommen), steigt der Gewinn mit zunehmender Parallelisierung an. Die Gesamtlaufzeit wird im seriellen Fall um circa 2% reduziert und erreicht mit 8 *Workern* eine Einsparung von fast 15%. Die Ladelast reduziert sich von anfänglichen 6,8% auf 21,2%.
Damit lassen sich nicht die Einsparungen durch Caching erreichen (29% bis 70,1%), auch wenn dies das theoretisch erreichbare Maximalziel wäre. In dem hier dargestellten Fall werden zwar alle vom Prefetching empfohlenen Blöcke benötigt, doch erfolgen die Blockanforderungen vom Algorithmus in relativ kurzen Zeiten, sodass sich keine vollständige Nebenläufigkeit einstellt. Unter Berücksichtigung dieser Aspekte sind die erzielten Ergebnisse bereits durchaus zufrieden stellend.
Alternative Markov-Ansätze, die Vorhersagen mithilfe von Initialisierungsgraphen treffen, haben sich in verschiedenen Testläufen noch nicht als ausgereift erwiesen. Dafür waren die erfassten Stichproben noch zu grob. Allerdings sind die hierdurch möglichen Erfolge auch wesentlich vom Datensatz abhängig. Während der Motordatensatz deutliche Turbulenzen aufweist und somit eine blockbasierte Strömungsrichtungstendenz häufig nicht eindeutig getroffen werden kann, sind Versuche mit dem Nasendatensatz, der eine klare Strömungsrichtung aufweist, viel versprechend.

5 Multiresolution und Streaming

Für unterschiedliche Anwendungszwecke kann es wünschenswert sein, dass die zu bearbeitenden Daten in mehreren Auflösungsstufen (Multiresolution, MR) vorliegen. Da Ausgangsdaten nicht per se als Multiresolution-Datensatz erzeugt werden, sind Techniken gefragt, die aus den gegebenen Daten mehrere Auflösungsstufen errechnen. Diese werden dann in einer MR-Hierarchie angeordnet. Somit besteht die Möglichkeit komplexe algorithmische Probleme zunächst auf einer sehr geringen Auflösungsstufe des Datensatzes zu lösen. Anschließend lassen sich immer exaktere Ergebnisse durch Wechsel des MR-Levels erreichen. Der erzielte Vorteil liegt in der schnellen Präsenz eines ersten approximativen Ergebnisses.
Die Anwendungsgebiete für MR-Techniken sind vielfältig. Besonders in der Bildverarbeitung werden sie seit längerem eingesetzt. Grundsätzliche Ansätze entstammen dabei der Signalverarbeitung, hier besonders der Frequenzanalyse. So kommen Fourier-Transformation bis Wavelet-Methoden [STOL95a, STOL95b, WEST97] zur Anwendung. Durch das Weglassen niederfrequenter Anteile im Bild können enorme Mengen an Bilddaten eingespart werden ohne wesentlich an Bildinformationen zu verlieren. Daher werden diese Ansätze vor allem zur Datenkompression verwendet. Da der Kompressionsfaktor variiert werden kann, lassen sich somit auch MR-Bilddaten erzeugen.
Eine erhebliche Kompressionsdichte wird in der Videoverarbeitung erreicht. Neben der Einzelbildkompression werden Ähnlichkeiten benachbarter Einzelbilder zur Kodierung verwendet. Häufig reicht es vollkommen aus, wenn das aktuelle Bild als Grundlage genommen wird und lediglich Bewegungsvektoren sowie einige wenige neue Bildelemente hinzukodiert werden um das nächste Bild zu erhalten. Dies ist das zentrale Kodierungsmerkmal z. B. von MPEG-kodierten Videodateien[12]. Dadurch lässt sich die Bandbreite, die zum Übertragen eines Videostroms benötigt wird, erheblich verringern. Das kontinuierliche Übertragen großer Datenmengen wie z. B. solcher Videodaten oder von Musikstücken im Internet wird allgemein als *Streaming* bezeichnet.
Für langsame Internet-Verbindungen wurden progressive Übertragungstechniken von Bilddaten entwickelt. Das bekannteste Beispiel ist *Progressive JPEG*. Beim Herunterladen aus dem Internet wird das Bild nicht wie üblich zeilenweise sondern mosaikartig aufgebaut. Bereits nach wenigen übertragenen Prozent der Bilddaten wird ein erster Überblick über das gesamte Bild vermittelt. Mit weiter eintreffenden Bilddaten werden Bildinformationen immer besser abgebildet bis letztendlich das vollständig aufgelöste Bild erscheint.
Die skizzierten Ideen lassen sich auch für das Übertragen von Extraktionsobjekten verwenden. So beschreiben Olbrich et al. in [OLBR99, OLBR01] einen Web-Server für den Abruf von Visualisierungsdaten. Der besondere Ansatz ist die Vorberechnung des Postprocessings auf einem Hochleistungsrechner. Danach werden die Ergebnisdaten in Form von 3D-Sequenzen auf einem separaten so genannten 3D-Streaming-Server abgelegt. Die eigentliche Exploration findet erst danach statt. Die extrahierten Objekte werden in eine Web-Seite eingebunden und über ein spezielles 3D-Viewer-PlugIn per Web-Browser ausgewählt. Für den Abruf der Daten kommt das *Real Time Streaming Protocol* (RTSP) zum Einsatz, das sehr häufig beim Streaming von Videodaten im Internet verwendet wird. Statt eines Videostroms werden jedoch 3D-Objekte in einem kontinuierlichen Datenstrom versendet. Als Animation lässt sich dieser in einer vollwertigen virtuellen Umgebung präsentieren.
Viracocha hingegen verwendet die hier angerissenen Kodierungs- und Übertragungstechniken, um die interaktive Exploration von CFD-Datensätzen in virtuellen Umgebungen zu verbessern. In den folgenden Abschnitten werden daher zuerst Bewertungskriterien für auf

[12] http://www.mpeg.org

Multiresolution und Streaming basierende Extraktionsalgorithmen entwickelt, bevor die einzelnen Techniken genauer untersucht und existierende Paradigmen und Algorithmen beschrieben werden. Anschließend wird die Erweiterung des Parallelisierungs-Frameworks um Streaming- und Multiresolution-Strukturen erläutert. Der darauf folgende Abschnitt geht dann auf die für dieses Framework implementierten Algorithmen ein und bewertet sie entsprechend der hier definierten Kriterien. Laufzeitmessungen schließen das Kapitel ab.

5.1 Streaming von Postprocessing-Daten

Die Auslagerung des Postprocessings auf einen Hochleistungsrechner, Parallelisierungsstrategien und das Datenmanagement zielen hauptsächlich auf die Reduzierung der Gesamtlaufzeit. Mag durch die Entlastung der Visualisierungs-Workstation eine flüssige Darstellung in virtuellen Umgebungen erreichbar sein, so sind diese Ansätze nicht in der Lage die Anforderungen des zweiten Interaktionskriteriums der maximalen Systemreaktionszeit, wie es in der Einleitung (siehe Absatz 1.1) definiert wurde, zu erfüllen. Selbst bei kleineren Problemgrößen stellen sich schnell Berechnungslaufzeiten ein, die sich über mehrere Sekunden bis hin zu einigen Minuten erstrecken können. So besteht die Gefahr sogar die maximale Sekundärreaktionszeit (3. Interaktionskriterium) zu überschreiten. Die explorative Untersuchung eines Strömungsdatensatzes wird daher immer wieder durch störende Wartezeiten unterbrochen.
Es lässt sich jedoch beobachten, dass für eine interaktive Exploration durchaus schon vereinfachte approximative Ergebnisse ausreichen, um zu entscheiden entweder die Berechnung abzubrechen und Parameter für eine nächste Berechnungsiteration zu modifizieren oder aber das Endergebnis abzuwarten.
Hinzu kommt, dass eine schnelle Präsentation von ersten Ergebnissen die subjektiv empfundene Wartezeit verkürzt. Auch wenn die visualisierten Teildaten noch keine aussagekräftigen Informationen enthalten sollten, wird die Verkürzung der Reaktionszeit als Steigerung von Interaktivität gewertet. Durch die Integration von Abbruch und Neustart des Kommandos mit modifizierten Parametern kommt man so einer explorativen Analyse schon sehr nahe.

5.1.1 Bewertungsschema

Der Mechanismus, der noch während der laufenden Berechnung Teildaten von Viracocha zur Visualisierungsapplikation überträgt, wird als *Data Streaming* bezeichnet. Dabei muss jeder Extraktionsalgorithmus unter dem Aspekt der Streaming-Tauglichkeit beurteilt werden. Für die Bewertung von Streaming-Ansätzen wird das Bewertungsschema aus Abschnitt 1.3 zugrunde gelegt. Die drei hauptsächlich untersuchten Zeitgrößen sind:

Latenzzeit: Sie gibt an, wie lange ein Streaming-Algorithmus braucht, bis er erste Teilergebnisse zum Visualisierungsrechner senden kann.

Gesamtlaufzeit: Die Gesamtlaufzeit ist auch von Interesse, da dann im Vergleich mit der nicht-streamenden Berechnungsvariante der Mehraufwand für die jeweilige Streaming-Technik recht gut ermittelt werden kann.

Streaming-Intervall: Liegen zu große Zeiträume zwischen zwei gestreamten Teildaten, wird dies nicht mehr als kontinuierlich empfunden. Im schlimmsten Fall verliert der Anwender doch noch den Bezug zwischen eintreffenden Daten und ausgelöstem Extraktionsbefehl.

Neben den rein messbaren Werten, wie den Laufzeiten, gibt es weitere Kriterien, die für die Bewertung sinnvoll sein können. Beispielsweise kann die subjektive Beurteilung des Anwenders Anhaltspunkte liefern, ob der verwendete Streaming-Ansatz als interaktiv, schnell und intuitiv wahrgenommen wird. Ein weiterer Maßstab in dieser Kategorie ist die flüssige Visualisierung zwischen zwei präsentierten Auflösungsstufen. Gerade wenn sich größere

Objektmodifikationen ergeben, kommt es zu ruckartigen Darstellungsänderungen (das so genannte *Popping*), die vom Anwender häufig als unangenehm empfunden werden. Weiche Übergänge, beispielsweise durch zusätzliche Interpolationsberechnungen, können hier die Akzeptanz deutlich erhöhen.
Das dritte Kriterium aus dem Bewertungsschema, die Durchführbarkeit, ist für viele Ansätze im Bereich von Streaming und Multiresolution ein Ausschlusskriterium. So gehen z. B. die meisten MR-Techniken davon aus, dass sie bereits das fertig berechnete und vollständig aufgelöste Ergebnis vorliegen haben. Aus diesem werden dann die gröber aufgelösten Stufen ermittelt. Dieser *Top-Down*-Ansatz ist jedoch nur in Ausnahmefällen für das Postprocessing von Interesse, da hier zuerst grobe Daten präsentiert werden, die durch die andauernde Berechnung weiter verfeinert werden sollen. Es sind somit also *Bottom-Up*-Ansätze gefragt.

5.1.2 Klassifizierung von Streaming-Ansätzen

Im Rahmen der Entwicklung von Streaming-Algorithmen besteht primär die Frage, wie sich Standardalgorithmen in streaming-fähige Varianten überführen lassen. Eine erste Klasse von Ansätzen überträgt bereits berechnete Teildaten zum *VisHost*. Hierzu gehören auch die betrachteroptimierten Extraktionsverfahren. Dafür reorganisiert man den Datensatz in einzelne Teildatenblöcke und ordnet diese räumlich an. Üblicherweise kommen dafür baumartige Metadatenstrukturen zum Einsatz. Sodann werden zuerst diejenigen Blöcke bearbeitet, die räumlich am nächsten zur Betrachterposition liegen. Zusätzlich wird die Blickrichtung berücksichtigt. Erst danach kommen die weiter entfernt liegenden Blöcke an die Reihe, die durch Verdeckung und durch ihre Distanz vom Anwender weniger wahrnehmbare Details beitragen können.
Die zweite Klasse von Strategien versucht dagegen gröber aufgelöste Zwischendaten (*Level of Detail,* LOD) zu berechnen, die im Gegensatz zu den Teildaten der anderen Streaming-Varianten jedoch immer einen wenn auch groben Gesamtüberblick über das zu extrahierende Strömungsmerkmal liefern. Dieser Ansatz kann sich als sehr rechen- und daher zeitintensiv herausstellen, besonders wenn ein bereits berechneter grober Auflösungsschritt nicht für die Berechnung des nächsten Levels verwendet werden kann. Ist dies allerdings möglich, so wird die Beziehung zwischen den einzelnen Detaillierungsschritten als progressiv bezeichnet. Die Datenbeschreibung des Gesamtergebnisses, die sich aus den Informationen des niedrigsten Auflösungslevels bis zum vollständig berechneten Ergebnis ergibt, nennt man des Weiteren *Progressive Multiresolution*.
Zusammenfassend lassen sich also atomare Extraktionsalgorithmen mit den folgenden zwei Strategien in streaming-fähige Varianten überführen, wobei sie sich auch kombinieren lassen:

- Übertragung bereits berechneter **Teildaten des Endergebnisses**
- ***Level-of-Detail*-Ansätze**
    - Entsprechend der Darstellung lässt sich diese Variante in **progressive** und **nicht-progressive Verfahren** unterteilen.

5.2 Erweiterung von ViSTA FlowLib und Viracocha

Ohne Streaming-Funktionalität kann davon ausgegangen werden, dass sich der Berechnungsvorgang als kompakte Einheit darstellt. Nach dem Absenden der Extraktionsanforderung, muss die Berechnungsdauer abgewartet werden. Trifft dann ein Ergebnis ein, kann man aber auch davon ausgehen, dass dieses Ergebnis bereits vollständig und die Extraktion abgeschlossen ist. Für die Implementierung des *Master-Worker*-Konzepts wurde genau dieses Verhalten vorausgesetzt. Zusätzliche Kommunikation und zusätzlicher Datenaustausch finden lediglich innerhalb des *WorkHosts* statt. Für den *VisHost* sind diese Vorgänge transparent, da sich nach außen hin der *WorkHost* als *Black Box* darstellt.

5.2.1 Der Visualisierungsrechner

Auf der *VisHost*-Seite, also durch ViSTA FlowLib, muss dennoch ein Management der beauftragten Berechnungen stattfinden. Dies übernimmt der *Extraction Manager* (vgl. Abbildung 69). Dieser sorgt für die Verbindung zum *WorkHost* und sendet über den Befehlskanal von Viracocha Berechnungsaufträge. Damit diese eindeutig adressiert werden können, sendet der *Scheduler* nach Erhalt eines Extraktionsbefehls eine eindeutige *Task ID* als Bestätigung zurück (vgl. hierzu auch Abschnitt 3.3). Der *Extraction Manager* sichert diese Auftragsnummer, um zum einen Kontrollbefehle, die sich nur auf diesen Befehl beziehen, an den *WorkHost* senden zu können. Zum anderen wird er damit in die Lage versetzt, Ergebnisdaten, die bei ihm in beliebiger Reihenfolge eintreffen können, zu identifizieren und der passenden Weiterverarbeitungsroutine zuzuordnen.

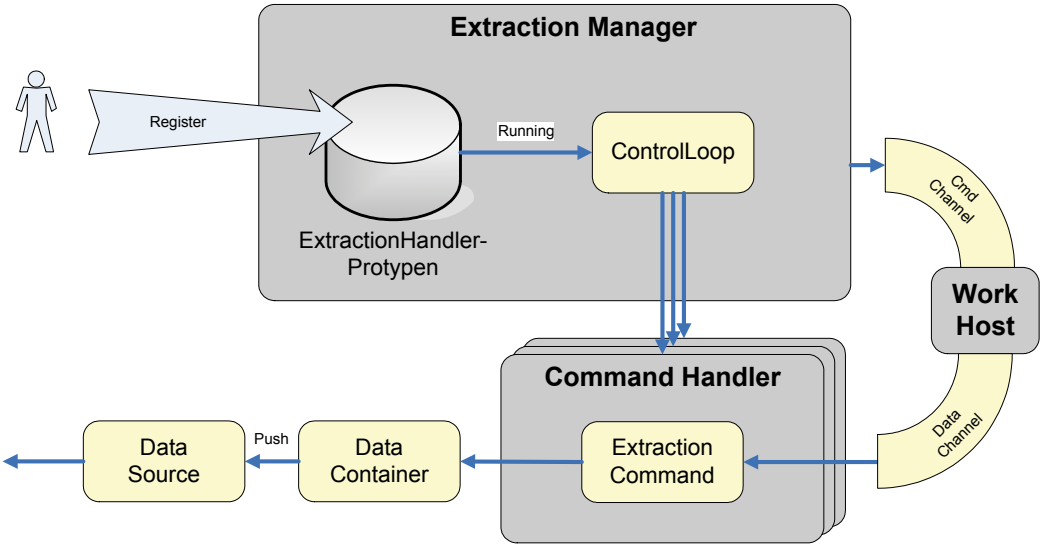

Abbildung 69: Der *Extraction Manager* koordiniert die Kommunikation mit dem *WorkHost* und ordnet empfangene Daten den richtigen Visualisierungspipelines zu

Für die Bearbeitung einer durch den Benutzer ausgelösten Extraktionsanfrage verfügt der *Extraction Manager* über eine Liste von so genannten *Command Handlern*. Als Prototypen registriert (Prototyp-Muster: vgl. [GAMM96]), lassen sich bei Bedarf beliebig viele Instanzen eines *Handlers* erzeugen. Während der *Extraction Manager* kein spezielles Wissen über Verwendung und Inhalt des Berechnungskommandos hat und lediglich Auftrag und Auftragsnummer verwaltet, steckt im *Command Handler* das Wissen, wie mit den empfangenen Ergebnisdaten zu verfahren ist.

Da diese in Form eines Byte-Datenarrays vorliegen, müssen sie interpretiert und für die Visualisierung aufbereitet werden. Diese Aufbereitung, die wiederum durch eine Visualisierungspipeline vorgenommen wird, kann Zeit kosten. Daher werden die verschiedenen Pipeline-Stufen durch mehrere *Threads* entkoppelt (siehe Abbildung 70). Die höchste Priorität hat dabei die Rendering-Schleife. Der Datenempfang und die Datenfilterung sind dabei nochmals voneinander getrennt. Nachdem neue Datenpakete eingetroffen sind und deserialisiert wurden, besteht die Möglichkeit, diese abschließend für die Visualisierung aufzubereiten. Typische Filterschritte sind Glyphenerzeugung, Normalenberechnung oder *Clipping*-Operationen. Bearbeitete Datenfragmente werden dann in ein als *Double-Buffered*-Speicher ausgelegtes Visualisierungsdatenobjekt, dem *Render Actor*, geschrieben. Während einer der beiden Speicherpuffer für jeden Render-Durchlauf Visualisierungsdaten liefert, wird der zweite von der Filter-Stufe befüllt. Ist die Aufbereitung beendet, werden beide Puffer umgeschaltet. Ab diesem Zeitpunkt werden die neuen Daten gezeichnet.

Am Ende der Visualisierungspipeline kommen Strategien zum Einsatz, die echtzeitfähiges Rendering durch Ausnutzung von programmierbarer Computergraphik-Hardware erreichen [SCHI05]. Somit stehen auf sämtlichen Stufen der Visualisierungspipeline Optimierungs-

strategien, angefangen mit Parallelisierungsansätzen auf dem *WorkHost* über adaptive Multiresolution-Verfahren (vgl. Abschnitt 5.2.3) bis zum hardware-optimierten Rendering, zur Verfügung.

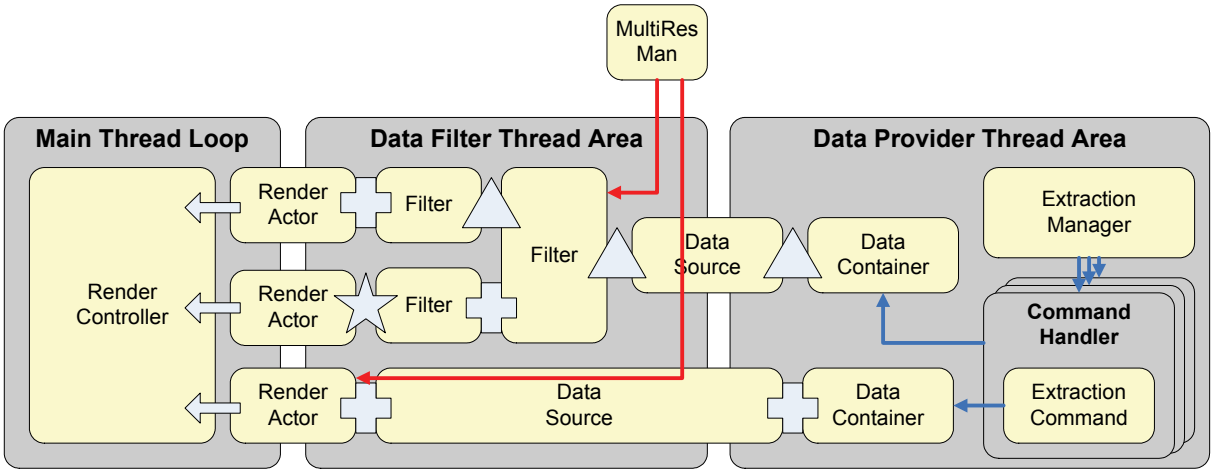

Abbildung 70: Stufen der Visualisierungspipeline (unterschiedliche Datentypen zwischen Knoten symbolisch angedeutet) auf dem *VisHost* und deren Aufteilung in *Threads*

5.2.2 Das Streaming-Kommando

Das *Master-Worker*-Konzept ist nicht in der Lage die Anforderungen des Daten-Streamings zu erfüllen. Daher war es notwendig einzelne Komponenten zu überarbeiten und zu ergänzen. Auf der *WorkHost*-Seite wurde ein neuer *Scheduler Command* für das Streaming implementiert. Zudem musste das Kommunikationsprotokoll zwischen *VisHost* und *WorkHost* modifiziert werden. Nach jedem Senden von Daten muss nun angegeben werden, ob das Kommando abgeschlossen ist oder noch weitere Daten folgen. Des Weiteren wurde eine neue Basis-Klasse angelegt, von der alle Klassen, die ihre Daten stückchenweise an den *VisHost* senden wollen, abgeleitet werden. Da hier bereits die wesentlichen Grundfunktionalitäten implementiert wurden, ist es nun deutlich leichter weitere Streaming-Klassen hinzuzufügen.

Reichte für den *Master-Worker*-Ansatz beim Einsatz von Standardberechnungen bereits ein einziger im *VisHost* arbeitender *Command Handler* aus, mussten nun in Abhängigkeit der Art, wie streamende Daten aufeinander aufbauen, unterschiedliche Varianten entwickelt werden. Dabei fällt der Arbeitsaufwand durchaus unterschiedlich aus.

LOD-Streaming: Hierfür muss lediglich die aktuelle Datendarstellung durch den neu eintreffenden Level ersetzt werden.

Teildaten-Streaming: Neue Daten werden einfach an die Liste der bereits zu visualisierenden Objekte angehängt.

***Progressive Streaming*:** Für solche Ansätze ist deutlich mehr Aufwand vonnöten. Hier überträgt der *WorkHost* am Anfang zuerst ein Basis-Mesh. Ein entsprechend angepasster *Handler* bereitet die Daten für eine erste grobe Visualisierung auf. Der *WorkHost* berechnet nun eine Verfeinerungsvorschrift nach der anderen und sendet diese an den *VisHost*, wo der spezialisierte *Command Handler* diese Vorschriften interpretiert und dementsprechend neue Daten in das bereits vorliegende Mesh integriert.

5.2.3 Der Multiresolution-Manager

Nach der Beendigung des Streaming-Vorgangs könnten alle Zwischendaten verworfen und lediglich das finale Ergebnis für das Rendering verwendet werden. Andererseits soll vom *VisHost* das erste Interaktionkriterium erfüllt werden, um ein interaktives Arbeiten in

virtuellen Umgebungen zu gewährleisten. Daher ist es möglich Streaming-Daten für eine adaptive Darstellungsauflösung in Abhängigkeit der Bildwiederholungsrate weiter zu verwenden. Um dies zu ermöglichen, kommt ein spezielles Multiresolution-Datenobjekt zum Einsatz. Die eintreffenden Streaming-Daten werden aufbereitet und hier abgelegt.
Ein Multiresolution-Manager kann später in Abhängigkeit der Render-Leistung Auflösungsstufen selektieren. Der MR-Manager ist aber auch in der Lage den Betrachterstandpunkt abzufragen, um eine betrachteroptimierte Darstellung der MR-Datenobjekte, sofern unterstützt, zu ermöglichen. Im Einzelnen übernimmt er folgende Aufgaben:

- Umschaltung von normalen **Datenstrukturen** zu Strukturen, die Multiresolution unterstützen.
- Änderung der Auflösung von Objekten in Abhängigkeit der **Bildwiederholungsrate**. Hierbei existieren Strategien, die Objekte zeitversetzt und iterativ anzupassen, bis vorgegebene Schwellwerte eingehalten werden können. Dadurch soll verhindert werden, dass durch die für die Auflösungsanpassung benötigte Rechenzeit die Bildwiederholungsrate weiter sinkt. Vergleiche hierzu auch [PASC00].
- Setzen von **Objektprioritäten** zur Steuerung der Detaillierungsgrade nach Wichtigkeit. So stellt beispielsweise die Kontextgeometrie gegenüber Extraktionsobjekten häufig eine untergeordnete Rolle dar, sodass hier die Darstellungsgenauigkeit bei Bedarf stärker gesenkt werden kann.
- Versorgung von Objekten, die das ***Progressive Refinement*** (vgl. hierzu [HOPP97, LUEB97, LUEB01]) unterstützen, mit den benötigten Kontrollparametern (wie beispielsweise dem *View Point*):

Leider eignen sich nicht alle Streaming-Ansätze für die nachträgliche Weiterverwendung. Beispielsweise ist die Familie der Streaming-Klassen, die nicht-progressiv bereits Teile des endgültigen Ergebnisses übertragen, ungeeignet. Sind die sichtbaren Lücken, die nach und nach aufgefüllt werden, während des Berechnungsvorgangs durchaus akzeptabel, wäre das Zurückschalten auf eine Teildarstellung mit deutlich wahrnehmbaren Löchern während der interaktiven Exploration kaum annehmbar.
Daher werden z. Zt. folgende Multiresolution-Datenstrukturen bzw. Behandlungen unterstützt:

- Container für gestreamte LOD-Extraktionsobjekte
- progressive Oberflächennetze (*FastMesh* [PAJA01, PAJA04], Wurzel-3-Netz-strukturen [KOBB00])
- Multiresolution-Darstellung von Polylinien

5.3 Evaluierung implementierter Streaming-Ansätze

Im ersten Teil dieses Kapitels wurden Klassifizierungs- und Bewertungsmuster für das Daten-Streaming vorgestellt. Daran schloss sich die Darstellung der Integration von Mechanismen für das Streaming und die Verwendung von Multiresolution-Datenstrukturen in ViSTA FlowLib und Viracocha an. Nun sollen die bisher beschriebenen Aspekte aufgegriffen und anhand von konkret implementierten Algorithmen untersucht werden. Im Zentrum steht dabei die so genannte Streaming-Matrix, die eine Strukturierung möglicher Verfahren vornimmt. Es werden anschließend exemplarische Lösungsansätze für die einzeln dort vorgestellten Kategorien präsentiert. Die mit den jeweiligen Ansätzen verbundenen Vor- und Nachteile werden ausführlich diskutiert.

5.3.1 Die Streaming-Matrix

Extraktionsalgorithmen lassen sich wie in der folgenden Auflistung angegeben gruppieren, sodass sich die Betrachtung der Streaming-Eigenschaften auf die jeweilige Gruppe einschränken lassen:

- **Schwellwertberechnungen**: Isoflächen, Wirbelregionen, kritische Punkte (Suche nach Kandidaten-Zellen), Wirbelkernlinien
- **Schnittobjekte**: implizite Funktionen, *Cell-Search*-Ansätze
- **Integrationsbasierte Ansätze**: Stromlinien, -röhren, -flächen, Bahnlinien, Streichlinien, Separationslinien

Soll das Streaming auf LOD-Ansätzen aufgesetzt werden, lassen sich Strategien nochmals entsprechend der verwendeten Datenstrukturen wie folgt unterscheiden:

- MR-Datenstrukturen der **Ausgangsdaten** (Volumengitter)
- MR-Datenstrukturen der **Extraktionsobjekte** (Oberflächennetze, Polylinien)

Mit den bereits in Abschnitt 5.1.2 definierten Streaming-Klassen und der gerade vorgenommenen Unterteilungen lassen sich nun mögliche Streaming-Algorithmen definieren. Es ergeben sich dabei Ansätze, die man in der so genannten Streaming-Matrix zusammenfassen kann. Dabei handelt es sich bei der in Tabelle 4 zusammengestellten Matrix um eine allgemeine Strukturierung. So kann es vorkommen, dass sich einige spezielle Ansätze zwar definieren lassen, aber ggf. nicht wirklich sinnvoll oder effizient auf eine der Extraktionsgruppen anwendbar sind. Andere Varianten lassen sich dagegen zusätzlich kombinieren.
Bei der Implementierung von Streaming-Ansätzen, die fortwährend bereits berechnete Teile des Endergebnisses übertragen (obere Hälfte der Matrix), spielen Parallelisierungsaspekte auf den Ausgangsdaten eine wesentliche Rolle. Häufig können die Standardextraktionsalgorithmen unverändert auf Partitionen der Simulationsdaten ausgeführt werden. Dann unterscheiden sich Streaming-Verfahren lediglich in der Strategie der Partitionierung. Adaptive Extraktionsverfahren greifen dagegen zusätzlich auf Metadaten, wie Intervallbäumen [BERG00, CIGN96] oder *Bounding-Box*-Informationen, zurück. Blickwinkelabhängige Ansätze (engl.: *View Dependent*, VD) brauchen weitergehende Modifikationen an den Ausgangsdaten.
LOD-Ansätze (untere Hälfte der Matrix), die auf den hier vorwiegend betrachteten strukturierten Ausgangsdatensätzen basieren, greifen in aller Regel auf Sampling-Ansätze zurück. Werden die Simulationsdaten bereits im Vorfeld entsprechend modifiziert, lassen sie sich direkt für die Extraktionen von MR-Ergebnisdaten verwenden. Andere Verfahren belassen die Eingangsdaten unverändert, versuchen jedoch die Berechnungsalgorithmen so abzuändern, dass das Ergebnis bereits aus MR-Datenstrukturen besteht. Die meisten Ansätze allerdings warten das Ergebnis einer Extraktion ab und starten erst dann eine Restrukturierungsphase, um MR-Versionen zu produzieren.
Progressive Ansätze für Isoflächenextraktion, obwohl besonders in dem hier behandelten Kontext äußerst wünschenswert, sind bisher nicht bekannt. Nur unter besonderen Annahmen wurden Verfahren vorgestellt, wie das von Labsik et al. in [LABS02] definierte *Isosurface Fitting*. Ebenfalls interessant sind Ansätze, die auf Topologiebeschreibungen von Skalarfeldern basieren, wie das Togologie-*Tracking*-Verfahren [BAJA02], die *Contour Propagation* [BAJA96, BAJA99] oder das *Topology-guided Downsampling* [KRAU01]. Trotz dieser topologiebasierten Ansätze (vgl. auch [WEST01]) wurden im Bereich des CFD-Postprocessings bisher keine wirklich zufrieden stellenden Isoflächenalgorithmen für ein *Progressive Streaming* basierend auf *Bottom-Up*-Extraktionen entwickelt. Daher beschränken sich die nachfolgenden Betrachtungen über *Progressive Streaming* auf Ansätze, die sich für Schnittobjekte entwickeln lassen.

Daten		Extraktionsgruppe		
Streaming	**Behandlung**	**Schwellwert**	**Schnitt**	**Integration**
Teile des Endergebnisses				
Prozess-weise	Satz von Blöcken	Standard-Algorithmus	Standard-Algorithmus	--
	Adaptives Blockladen	Adaptive Ansätze	Adaptive Ansätze	Standard-Algorithmus
Block-weise	Einzelblock	Standard-Algorithmus	Standard-Algorithmus	--
Zellweise (Bins)	Entsprechend der Zellnummer	Zellbasierte Ansätze	Zellbasierte Ansätze	--
	View Dependent	Space Partitioning Tree	1.) Standard-Algorithmus 2.) Sortiertes Sampling	--
	Zufälliges Sampling	Zellbasierte Ansätze	Zellbasierte Ansätze	--
Punktweise (Bins)		--	Punktbasierte Ansätze	übertrage Punkte (Menge) sobald berechnet
Level of Detail				
Datensatz-LOD	Datensatz als MR vorhanden	Standard-Algorithmus	Standard-Algorithmus	Standard-Algorithmus
	Subsampling	Standard-Algorithmus	Standard-Algorithmus	Standard-Algorithmus
Ergebnis-LOD	Extraktions-Ergebnis als MR	Standard-Algorithmus + Top-Down-Ansätze	MR-Schnittobjekt	Variierende Integrations-parameter
Adaptives MR	Einzelblöcke	Standard-Algorithmus + Wavelets	--	--
	Baum-Hierarchie	Baumbasierte Ansätze	Baumbasierte Ansätze	Standard-Algorithmus
Progressive Streaming		--	Subsampling, ggf. mit Selective Refinement	--

Tabelle 4: Streaming-Ansätze, Varianten und verwendbare Extraktionsverfahren

Wegen der Bedeutung von Datenstrukturen, werden in den folgenden Abschnitten vor allem die geometrischen Eigenschaften von 3D-Gitterstrukturen sowie von polygonalen Oberflächen-Meshes, wie man sie durch Isoflächenextraktion oder Schnittoperation erhält, genauer untersucht. Des Weiteren werden exemplarische Lösungsansätze für die einzelnen Kategorien der Streaming-Matrix präsentiert und bewertet.

5.3.2 Subsampling von strukturierten Gittern

Um MR-Hierarchien von strukturierten Datengittern zu erhalten, wird häufig das so genannte Subsampling eingesetzt. Hierfür wird ein Sampling-Operator definiert, der entsprechend einer vorgewählten Sampling-Rate den Originaldatensatz abtastet und neue Knoten ermittelt. Als Ergebnis erhält man einen reduzierten Datensatz. Durch die wiederholte Anwendung des Sampling-Operators bekommt man die gewünschte Hierarchie.
Besonders schnell kann das Subsampling durchgeführt werden, wenn sich die neuen Knoten ohne Interpolation bestimmen lassen, also nur Knoten des Eingangsgitters zurückgeliefert werden. Weist ein strukturiertes Gitter zumindest entlang einer Gitterachse eine Auflösung

von $n = 2^m + 1$ ($m > 0$) auf, dann ist diese Anforderung leicht zu erfüllen und MR-Hierarchien schnell generierbar. Der Subsampling-Operator kann dann entlang dieser Achse jeden zweiten Datenpunkt überspringen. Er lässt sich nun solange rekursiv anwenden, bis nur noch die Randpunkte des Ursprungsgitters übrig sind.
Statt jeden zweiten Punkt abzutasten, können auch andere Abtastraten s verwendet werden. In diesem Fall sollte die Auflösung $n = s^m + 1$ betragen. Kommen hintereinander mehrere Sampling-Raten s_i zum Einsatz, erhält man folgende Formel, die man letztendlich bis zur Primfaktorzerlegung der um eins reduzierten Auflösung fortführen kann:

$$n = 1 + \prod_i s_i^{m_i}; \qquad s_i, m_i \in \mathbb{N} \tag{5}$$

Dieser Ansatz lässt sich für die Erzeugung einer akkuraten Multiresolution-Struktur verwenden. Abbildung 71 stellt ein Beispiel dar, das nur mit den im Datengitter definierten Punkten auskommt, und basierend auf den verschiedenen Auflösungsstufen Isoflächen extrahiert.

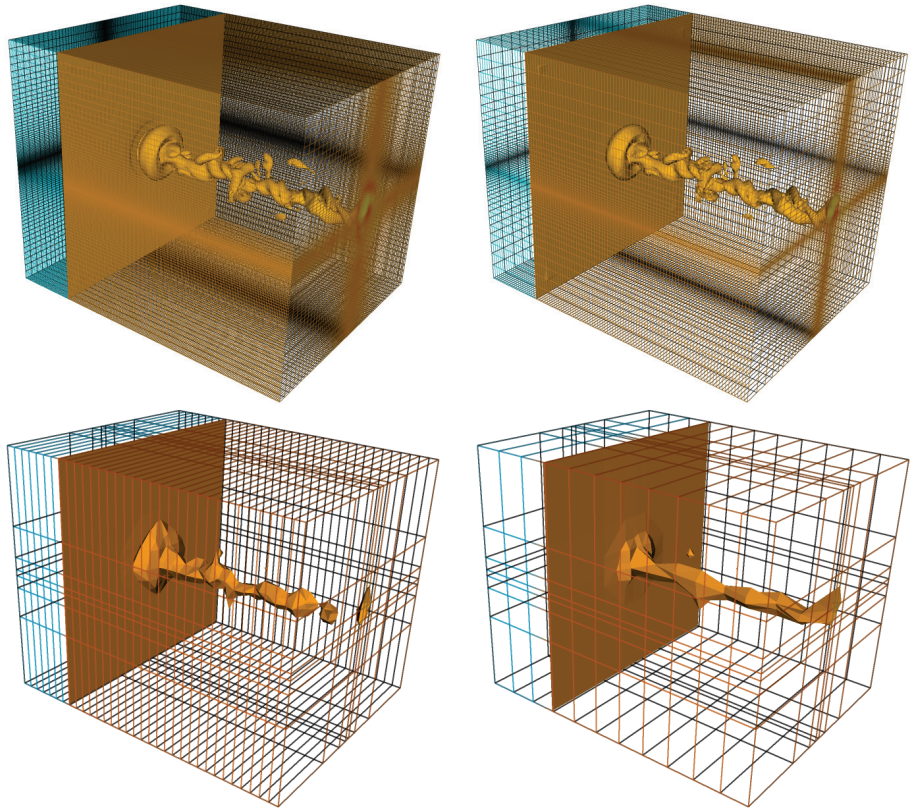

Abbildung 71: Multiresolution-Subsampling eines *Shock*-Datensatzes mithilfe von per Primfaktorzerlegung ermittelten Sampling-Raten

Verzichtet man auf die gleichmäßige Verteilung der Samples, kann der strukturierte Datensatz entlang der *i*-*j*-*k*-Achsen auch mit beliebigen Schrittweiten abgetastet werden. Verschiedene Varianten, die wiederum ausschließlich auf die definierten Gitterpunkte zugreifen, sind hier denkbar. Eine dieser Möglichkeiten unterteilt die Datenpunkte entlang einer Achse in eine vordere und eine hintere Teilmenge. Diese werden sodann vom Startpunkt, bzw. vom Endpunkt aus beginnend mit der vorgegebenen Schrittweite abgetastet, bis die jeweilige Teilmenge überschritten wurde. Entsprechend wird mit den Punkten auf den weiteren Achsen verfahren. Eine andere Variante tastet regelmäßig ab bis der Datensatz verlassen wurde. Dann wird abschließend einfach der letzte Datenpunkt noch hinzugenommen. Da alle bisher vorgestellten Operatoren direkt auf den originalen Daten arbeiten, sind sie sehr schnell und können während des Postprocessing angewendet werden.

Ein anderer Abtast-Operator arbeitet unabhängig von der Datensatzauflösung. Dabei wird vorgegeben, wie viele Samples pro Level und Richtung genommen werden sollen. Dadurch kommt es in der Regel dazu, dass die Sample-Punkte nicht mehr mit den Gitterknoten übereinstimmen. Der Operator muss daher nun zwischen den benachbarten Werten interpolieren, sodass dieser Ansatz auch als *Resampling* bezeichnet wird. Um optimale Ergebnisse zu erhalten, wird nicht iterativ vorgegangen, sondern vielmehr immer der Originaldatensatz zur Berechnung eines jeden Levels verwendet. Da durch diesen Operator besonders bei großen Datensätzen das Erstellen von MR-Hierarchien sehr zeitaufwändig sein kann, werden die einzelnen Daten-Level in einem Vorbearbeitungsschritt erstellt und auf Festplatte gespeichert. Hieraus ergibt sich aber ein zusätzlicher, nicht unerheblicher Festplattenspeicherbedarf[13].
Bei allen vorgestellten Sampling-Ansätzen wird die Attributverteilung des zugrunde liegenden Skalarfeld nicht berücksichtigt. Markante Merkmale können ggf. eliminiert werden. Die Topologie von Datenfeldern wird üblicherweise nicht erhalten. Ein Subsampling-Ansatz für Isoflächen-Preview unter Verwendung von strukturierten Datensätzen ist in [KRAU01] dargestellt.

5.3.2.1 Subsampling in Multi-Block-Strukturen

Werden strukturierte Datensätze isoliert betrachtet, lassen sich die beschriebenen Sampling-Operatoren ohne Einschränkungen einsetzen. Werden jedoch curvilineare Datenblöcke als Bestandteile eines gemeinsamen Multi-Block-Datensatzes verwendet (vgl. Abbildung 72), treten Probleme an den Grenzflächen auf.

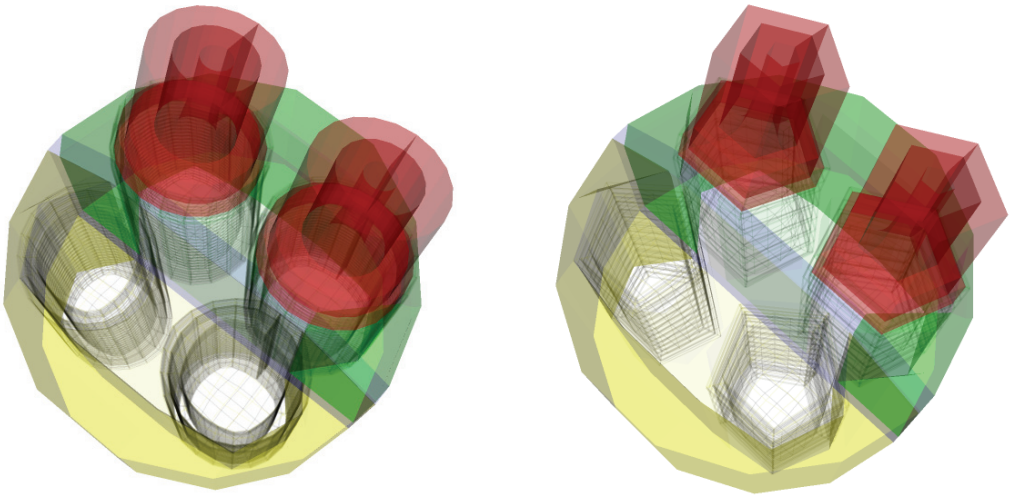

Abbildung 72: Subsampling in einem MB-Datensatz, 1. Level (links) und 3. Level (rechts)

Beim Subsampling bleiben die Start- und Endpunkte pro Richtung in allen Leveln erhalten. Für ein gleichmäßiges Subsampling wird vom Startwert mit äquidistanten Schritten innerhalb des Gitters fortgeschritten. Liegen die Startwerte benachbarter regulärer Gitter nicht übereinander, ist die Anwendung gleichmäßiger Schrittweiten häufig verhindert. Möchte man dennoch punktgenaues Sampling durchführen, bleibt in diesem Fall nur ein individuelles unregelmäßiges Abtasten übrig. Ein Beispiel zeigt Abbildung 73.
Dieser Ansatz ist nur schwer automatisch zu realisieren. Daher bietet es sich an Blöcke unabhängig voneinander abzutasten. Die *Full-Matching*[14]-Eigenschaft würde sich dann nicht mehr erhalten lassen. Da es sich bei den Blöcken allgemein um reguläre oder rectilineare Blöcke handelt, treten allerdings nun noch zusätzlich zwischen vorher deckungsgleichen Verbindungsflächen Lücken oder Überschneidungen auf. Extraktionsberechnungen, wie z. B.

[13] Bei einer Sampling-Rate von 2 ist der gesamte Speicherbedarf durch den Faktor 2 begrenzt.
[14] Gitterrandpunkte benachbarter Blöcke liegen exakt übereinander.

Isoflächenextraktionen, die nur lokal arbeiten und nun auch in einem gröber aufgelösten Datensatz eingesetzt werde sollen, kommen damit zurecht, weisen dann aber an den Übergängen so genannte *Cracks* auf. Andere Verfahren, wie Stromlinienberechnungen, würden hier ohne Modifikationen des Algorithmus abbrechen.

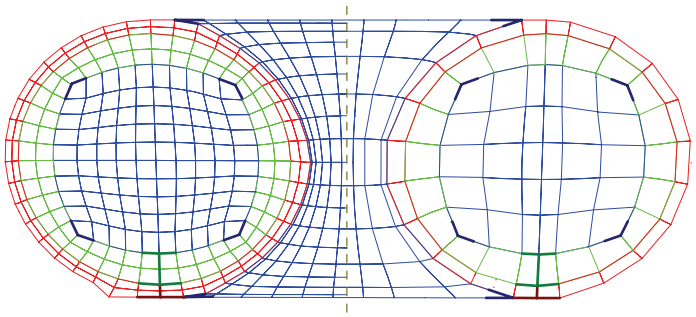

Abbildung 73: Adaptives Subsampling in einem Multi-Block-Datensatz (links: originale Auflösung, rechts: ausgedünnt), im *C-Space* definierte Ecken sind hervorgehoben

5.3.2.2 Multiresolution-Datensätze

Um möglichst schnell Daten einlesen und eine Vorschau der angeforderten Strömungsmerkmale extrahieren zu können, wurden die verwendeten Datensätze mithilfe des Subsamplings als LOD-Datensätze offline erstellt und die jeweiligen Stufen in einzelne Dateien abgespeichert. Nach dem Erhalt einer Extraktionsanforderung kann der Streaming-Algorithmus nun zuerst die gröbste Auflösungsstufe einlesen. Dabei sind nicht nur die Zugriffszeiten sondern auch die Berechnungszeiten wegen der geringen Auflösung erheblich verkürzt. Während der Anwender bereits ein quasi vollständiges, wenn auch sehr grobes Bild seiner Postprocessing-Anforderung analysieren kann, wird der nächste Auflösungslevel nachgeladen, extrahiert und das Ergebnis gestreamt. Zum Schluss kommt der Original-datensatz an die Reihe. Da hier unabhängig voneinander berechnet wird, geht allerdings die mit diesem Ansatz erreichte schnelle Bereitstellung erster Komplettübersichten auf Kosten der Gesamtlaufzeit.
Ein zweites Problem ist die erwähnte Problematik LOD-Datensätze durch Subsampling zu erzeugen. Durch eine naive Auslassungsstrategie können in Multi-Block-Datensätzen Lücken und Überlappungen entstehen, die bis auf fehlerhafte Übergänge an den Blockgrenzen allerdings keinen Einfluss auf die Isoflächenberechnung haben. Die auftretenden Artefakte sind wegen des temporären Charakters der Zwischenergebnisse während des Streamings jedoch tolerierbar.
Das in Abbildung 74 dargestellte Beispiel zeigt den AIA-Motordatensatz. Seine einzelnen Blöcke wurden durch das in Abschnitt 5.3.2 beschriebene *Resampling* gleichmäßig um die Faktoren zwei, vier und acht reduziert und für die spätere Verwendung auf Festplatte abgespeichert. Während des Streamings von Isoflächen ergeben sich sodann die von links nach rechts dargestellten Ergebnisse.

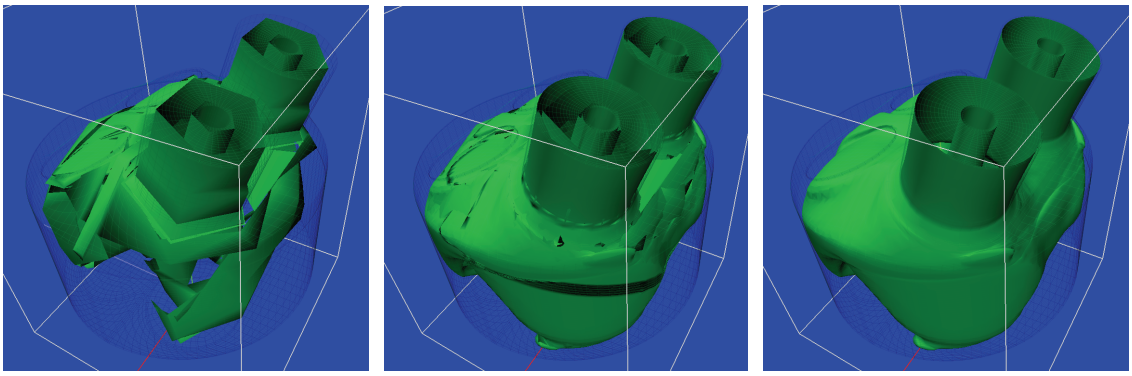

Abbildung 74: Isoflächenextraktion auf einem MR-Datensatz

5.3.3 Streaming von Isoflächen

Die Extraktion von Isoflächen spielt eine herausragende Rolle in der Analyse von Skalarfeldern, die üblicherweise bei jeder numerischen Simulation anfallen. Daher liegt ein Fokus bei der Implementierung von Streamingverfahren auf der effizienten Umsetzung für Isoflächen. Der bereits erwähnte Ansatz mithilfe des MR-Datensatzes ist z. B. exemplarisch als Isoflächen-Streaming-Verfahren realisiert. Aber auch einfache Partitionierungsverfahren sowie die aufwändigen VD-Verfahren arbeiten vorzugsweise mit Isoflächen. Im Folgenden werden diese Verfahren genauer untersucht.

5.3.3.1 Blockweises Streaming von Isoflächen

Isoflächenberechnungen sind zellbasierte Extraktionsverfahren, d. h. die Anzahl der ermittelten Polygone des Ergebnisses hängt ganz entscheidend von der zugrunde liegenden Datensatzauflösung ab. Streamingfähige Einfachversionen extrahieren auf Blockbasis und übertragen Blockergebnisse. Aber gerade bei einem geringen Berechnungsaufwand kann es schnell zum Streaming-Stau kommen, d. h. Datenpakete treffen schneller beim *VisHost* ein als er verarbeiten kann. Das führt dazu, dass das komplette System durch Kommunikation blockiert und kaum noch Ressourcen für das Rendering bzw. für Extraktionsberechnungen zur Verfügung stellen kann. Abbildung 75 veranschaulicht dieses Problem anhand von Messergebnissen, die mit dem Turbinendatensatz erstellt wurden.

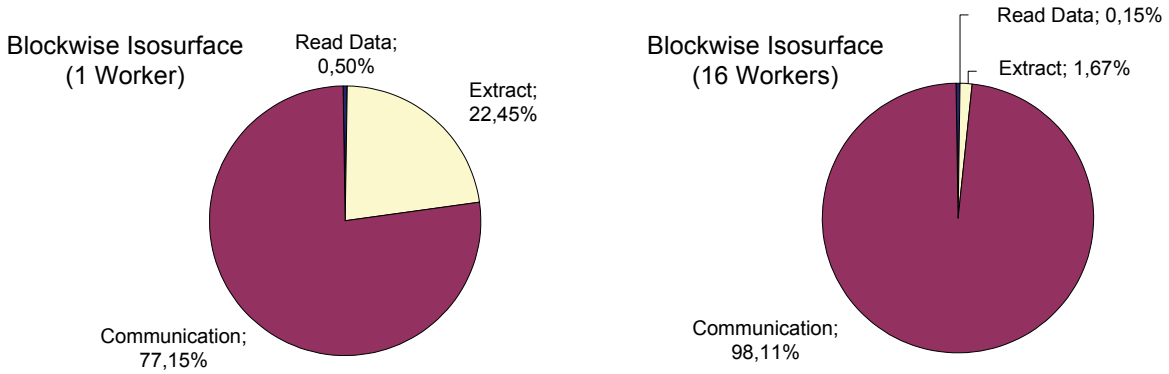

Abbildung 75: Kommunikations-Overhead mit zunehmender *Worker*-Anzahl

Dieses Kommunikationsproblem konnte nicht mehr beobachtet werden, nachdem der einzelne *Worker* die Ergebnisse statt nach der Bearbeitung eines Blocks (blockweises Streaming) erst nach der Berechnung aller ihm zugewiesenen Blöcke (prozessweises Streaming) verschickte. Dies belegt die Brisanz nicht-optimierter Netzwerkkommunikation verteilter Komponenten, die im Falle von ViSTA FlowLib und Viracocha auf einem TCP/IP-basierenden, proprietären Kommunikationsprotokoll (für das Versenden und Verwalten von Kommandos und Extraktionsergebnissen) beruht.

5.3.3.2 Betrachterabhängige Extraktion

Um den Benutzer schnell mit relevanten Daten, d. h. in diesem Fall mit nahe liegenden und mutmaßlich unverdeckten Objektdetails zu versorgen, wird die Betrachterposition zum Zeitpunkt der Anforderung mit in der Berechnung berücksichtigt. Die Isofläche wird also quasi von vorne nach hinten extrahiert. Um dies ansatzweise zu gewährleisten, werden die *Bounding-Boxen* der Multi-Blöcke für die Distanzbestimmung verwendet und entsprechend sortiert. Sodann werden die Blöcke in dieser Reihenfolge für die Berechnung verwendet, sowie die jeweils blockweise ermittelten Daten an den *VisHost* übertragen. In Abhängigkeit der Block-Anzahl eines Datensatzes kann somit schon eine recht ordentliche betrachterabhängige Berechnung präsentiert werden. Bei grober Aufteilung und durch die üblicherweise verwendeten curvilinearen Blockstrukturen kann dieser Effekt aber auch zunichte gemacht werden.

Um dieser Gefahr entgegenzutreten, kommt ein *Binary-Space-Partitioning*-Baum (BSP-Baum) zum Einsatz. Für den Wurzelknoten wird eine *Bounding-Box* berechnet, die alle Zellen umfasst. Ihm wird neben dieser *Bounding-Box* noch der Skalarbereich, der von allen Zellen abgedeckt wird, zugewiesen. Danach wird die *Bounding-Box* in der Mitte ihrer längsten Achse durchgeschnitten. Die beiden entstehenden Teil-Boxen werden jeweils bei den Kinderknoten abgespeichert. Diese berechnen noch für sich die nun vorliegenden Intervalle der Skalarwerte. Dann werden ihre *Bounding-Boxen* ebenfalls zerteilt und wiederum an deren Kinderknoten weitergereicht. Dieses Procedere wird nun solange fortgesetzt, bis eine Mindestanzahl von Zellen, die sich in einer *Bounding-Box* befinden müssen, unterschritten wird. So erhält man einen vollständigen BSP-Baum, der zusätzlich als approximativer Intervallbaum eingesetzt werden kann. Da die Konstruktion unabhängig vom aktuellen Isowert ist, kann der Baum vorberechnet und bei Bedarf von der Festplatte nachgeladen werden.[15]
Die abgespeicherten Skalarintervalle können nun dazu verwendet werden die Bestimmung der aktiven Zellen zu beschleunigen. Nach dem Eintreffen einer Isoflächenanforderung wird in einem ersten Schritt durch eine Breitensuche der vollständige Baum traversiert. Verzweigt wird, sofern sich dort ein gültiges Skalarintervall befindet, immer zuerst zu demjenigen Kindknoten, dessen *Bounding-Box* die kürzere Distanz zum Betrachter aufweist. An einem Blattknoten angekommen, werden alle Einzelzellen der Reihe nach untersucht, ob sie entsprechend des Isowertes eine aktive Zelle sind. Ist dies der Fall, werden sie in einer Liste gespeichert. Wurden alle Blattknoten mit gültigem Intervall-Bereich eingesammelt, liegt eine Liste vor, die zumindest durch die Granularität des BSP-Baums vorsortierte, betrachter-orientierte aktive Zellen beinhaltet. Diese Liste kann nun von vorne nach hinten durchlaufen werden und innerhalb von aktiven Zellen lassen sich die Isoflächenfragmente berechnen. Wurde eine gewisse Anzahl von Zellen abgearbeitet, werden die Ergebnisse schon einmal zum *VisHost* gestreamt. Dort kann sodann die betrachterabhängige Isofläche nach und nach aufgebaut werden. Ein Beispiel ist in Abbildung 76 gegeben.

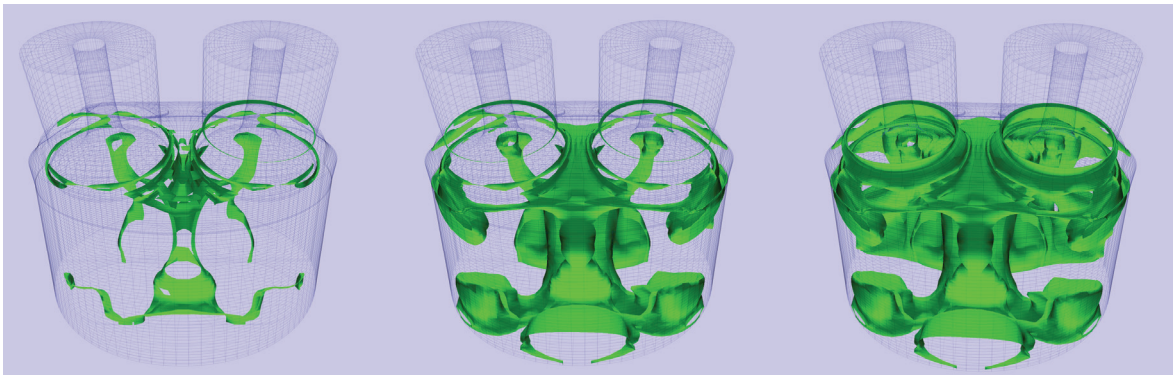

Abbildung 76: Betrachterabhängige Isoflächenextraktion, von links nach rechts zunehmende Detaillierungsstufen

5.3.3.3 Vergleich von Streaming-Ansätzen für Isoflächen und Wirbeln

Die bisherige Analyse der Multiresolution- und Streaming-Algorithmen wurden losgelöst vom Datenmanager, der von Viracocha zur Verfügung gestellt wird, betrachtet. Daher sollen in diesem Abschnitt nun zusätzlich die Auswirkungen des VDMS auf das Streaming-Verhalten dargestellt werden (vergleiche hierzu auch [GERN04c]).
Als Testszenario wurden der Motordatensatz des AIA und der Turbinendatensatz des DLR auf einer SunFire-6800 verwendet. Die Messreihen *Simple Iso* und *Simple Vortex* zeigen das Parallelisierungsverhalten ohne Streaming auf. Als Streaming-Varianten kamen mit

[15] Der hier beschriebene Ansatz entspricht weitestgehend dem von Wilhelms und van Geldern in [WILH92] vorgestellten Ansatz. Allerdings kommt dort anstelle des BSP-Baums ein Octree zum Einsatz.

Viewer Iso eine betrachteroptimierte Isoflächenextraktion (vgl. Abbildung 76) und mit *Streamed Vortex* ein blockweises Streaming von Lambda-2-Wirbelregionen [JEON95] (vgl. Abbildung 77) zum Einsatz.

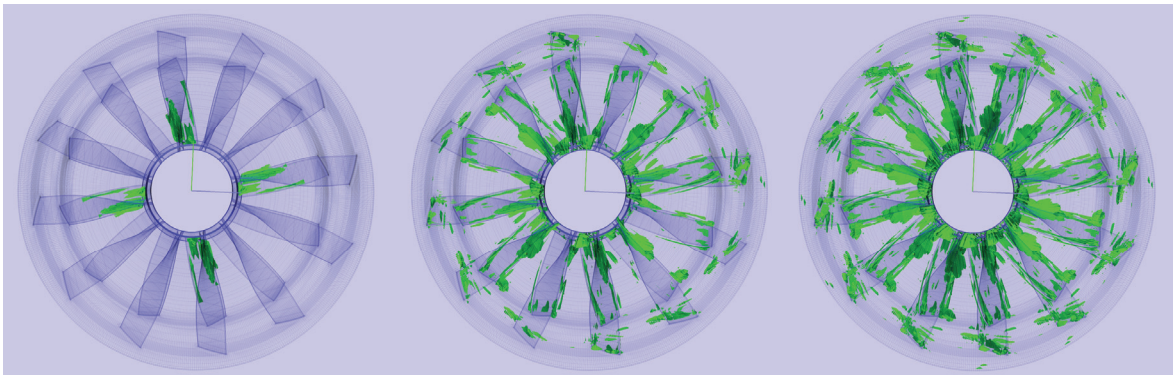

Abbildung 77: Mehrere Streaming-Schritte von Lambda-2-Wirbelfragmenten in einer Turbine

Die Streaming-Varianten arbeiteten im Gegensatz zu den Standardparallelisierungsansätzen mit gefüllten Caches, sodass die Auswirkungen der Lade-, Caching- und Prefetching-Strategien nicht Gegenstand der Untersuchung sind. Ergänzend wurden zwei weitere Messungen *Iso DataMan* und *Vortex DataMan* durchgeführt, die mit aktivem VDMS (vollständig gecachten Daten) aber ohne Streaming arbeiteten. Da somit fast ausschließlich die reine Berechnungszeit erfasst wurde, liegt hier eine untere Laufzeitschranke vor, die gut zur Bewertung der Streaming-Ansätze dienen kann.
Die Gesamtlaufzeiten zur Berechnung von Isoflächen sind in Abbildung 78 gegeben. Der Einfluss des Cachings von Daten ist beim Vergleich der Standardverfahren ohne und mit VDMS deutlich zu erkennen. Vergleicht man daneben den betrachterabhängigen Streaming-Ansatz *Viewer Iso* mit der Variante *Iso DataMan*, so stellt man eine um ein Vielfaches erhöhte Laufzeit fest. Dies ist der Mehraufwand, den man für das Streaming aufbringen muss. Der Anstieg lässt sich hauptsächlich durch die Online-Konstruktion des BSP-Baums für den VD-Ansatz begründen.

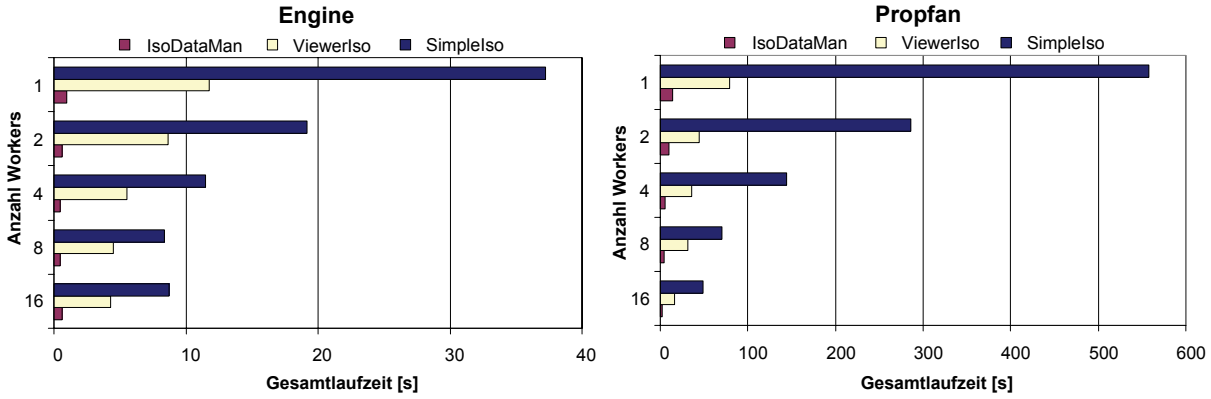

Abbildung 78: Gesamtberechnungszeiten für Isoflächen, Engine (links) und Propfan (rechts)

Was man ebenfalls in Abbildung 78 erkennen kann, ist die verbesserte Skalierung mit zunehmender Datensatzgröße. Beim relativ kleinen Motordatensatz (linkes Diagramm, *Engine*) ist bereits mit 8 *Workern* eine Sättigung festzustellen. Die Turbine dagegen (rechtes Diagramm: *Propfan*) weist durch ihre grundsätzlich höhere Datenlast auch noch mit 16 *Workern* einen ordentlichen *Speed-up* auf.
Jedoch wirkt sich nicht nur die Datenlast sondern auch die Berechnungslast positiv auf die Skalierbarkeit aus. In Abbildung 79 sind die Ergebnisse für die deutlich aufwändigere Berechnung von Wirbelregionen zusammengefasst. Die durch den Berechnungsalgorithmus entstehende höhere Last ist direkt ablesbar, wenn man das *Simple-Vortex*-Ergebnis mit der

Simple-Iso-Laufzeitmessung in Abbildung 78 für den Fall, dass nur ein *Worker* arbeitet, vergleicht.

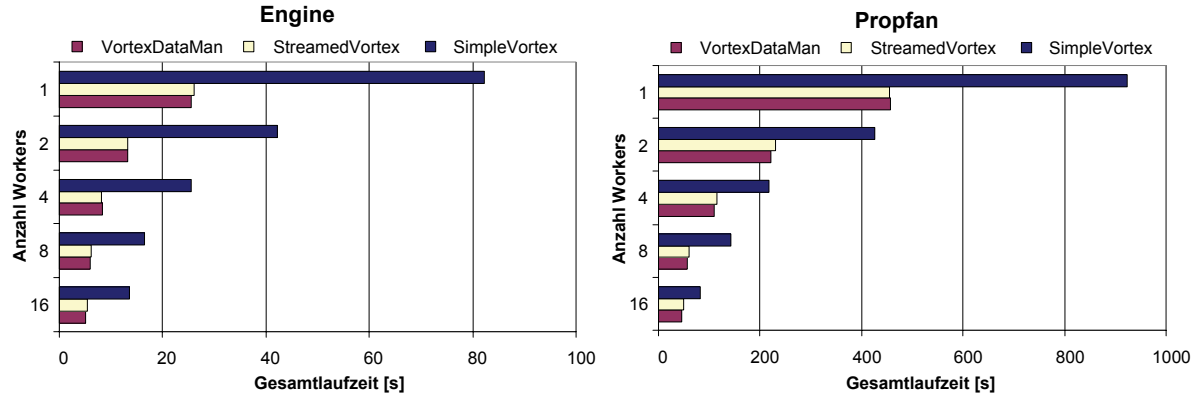

Abbildung 79: Gesamtberechnungszeiten für Wirbel, Engine (links) und Propfan (rechts)

Aber auch die beiden anderen Verfahren brauchen nun deutlich länger. Betrachtet man *Vortex DataMan* wiederum als untere Schranke für die Gesamtlaufzeit, dann fällt auf, dass der *Streamed Vortex* fast gleich auf ist. Im Gegensatz zur vorher betrachteten Streaming-Version *Viewer Iso* werden keine zusätzlichen Baumstrukturen erzeugt. Der Vergleich gibt somit die reinen Kommunikationskosten wieder. Bedingt durch den hohen Berechnungsaufwand ist die zusätzliche Streaming-Kommunikation aber kaum noch wahrnehmbar.
Die Latenzzeiten, d. h. die Zeit bis zum Eintreffen der ersten Teildaten beim *VisHost*, wurden ausschließlich mit der Turbine durchgemessen. Die Latenzzeiten für den VDMS-verwendenden Standardextraktionsalgorithmus sind mit seiner Gesamtlaufzeit (vgl. Abbildung 78 rechts) identisch. Wie die Ergebnisse in Abbildung 80 zeigen, erfüllt der auf den VDMS setzende betrachteroptimierte Algorithmus die in ihn gesetzten Erwartungen. Bereits ab zwei *Workern* können innerhalb einer Sekunde die ersten *View-Dependent*-Isoflächenteile präsentiert werden. Werden die Berechnungen komplexer, dann steigt erwartungsgemäß die Latenzzeit. Dennoch liegen auch die Wartezeiten für den *Streamed Vortex* immer noch deutlich unter fünf Sekunden.

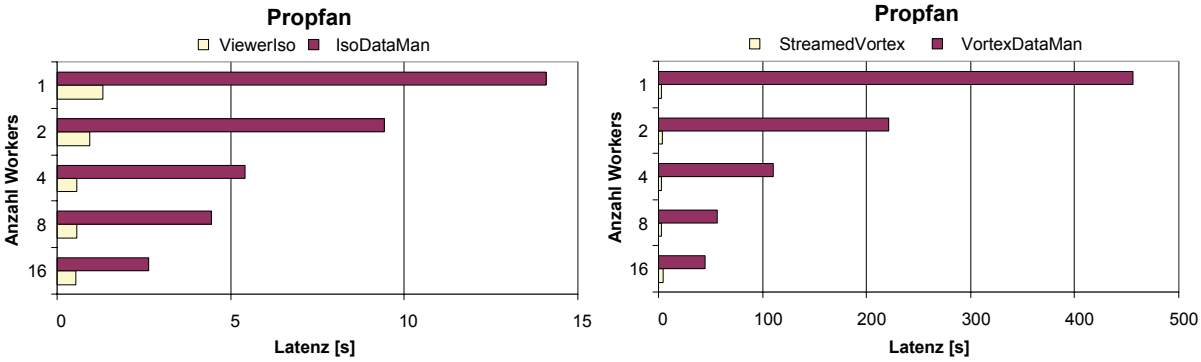

Abbildung 80: Latenzzeiten für die Isoflächen- (links) und Wirbelextraktion (rechts) unter Verwendung des Turbinendatensatzes

Die gemessenen Zeiten schwanken mit zunehmenden *Workern* etwas, liegen aber alle relativ beisammen. Das sollte auch so sein, denn sobald der erste *Worker* Ergebnisse vorweisen kann, werden diese zur Visualisierung verschickt. Schwankungen lassen sich durch unterschiedliche Blockgrößen, die zuerst verarbeitet werden müssen, begründen. Dass die Unterschiede der Latenzzeiten zwischen *Viewer Iso* und *Iso DataMan* nicht extrem hoch sind, erklärt sich erneut durch die geringe Berechnungslast, die während der Isoflächenberechnung auftritt. Wächst diese Last, dann kann Streaming seine Stärke zeigen, wie der Vergleich von *Streamed Vortex* und *Vortex DataMan* eindrucksvoll unterstreicht.

Zusammenfassend können beide hier untersuchten Streaming-Verfahren überzeugen. Der einfachere der beiden kommt gut mit der zusätzlichen Kommunikation zurecht. Kommunikationsstaus, wie in Abschnitt 5.3.3.1 berichtet, konnten bei der hier zu bearbeitenden Problemgröße nicht beobachtet werden. Aber auch der VD-Streaming-Algorithmus konnte überzeugen. Der Verzicht, den BSP-Baum als Offline-Datei einzulesen, ermöglichte die Untersuchung der vollständig anfallenden Streaming-Kosten. Angesichts dessen sind die Laufzeiten durchaus zufrieden stellend und die Latenzzeiten im gewünschten Bereich.

5.3.4 Streaming von Schnittflächen

Klassischerweise versteht man unter einer Schnittfläche eine planare Ebene, die durch den Datensatz gelegt wird. Es werden die geschnittenen Gitterzellen gesucht, Schnittpunkte auf Zellkanten bestimmt, skalare Werte interpoliert und dann auf die Schnittebene gemappt. Zwar lässt sich eine Schnittfläche mit einfachen interaktiven Hilfsmitteln platzieren, doch in Abhängigkeit des verwendeten Datensatzes sind die präsentierten Ergebnisse häufig wenig informativ und daher recht unbefriedigend.
Das in Abbildung 81 dargestellte Beispiel kann dies verdeutlichen. Wegen des rotationssymmetrischen Aufbaus der dort dargestellten Turbine, ist die eingefügte Schnittfläche (linkes Bild) von zweifelhaftem Wert. Zum einen ist sie entlang der Strömungsrichtung platziert und zum anderen ist sie in zwei kleine Bereiche zerteilt worden. Beide Aspekte führen dazu, dass man lediglich ein eingeschränktes, mäßig aussagekräftiges Strömungsbild erhält.

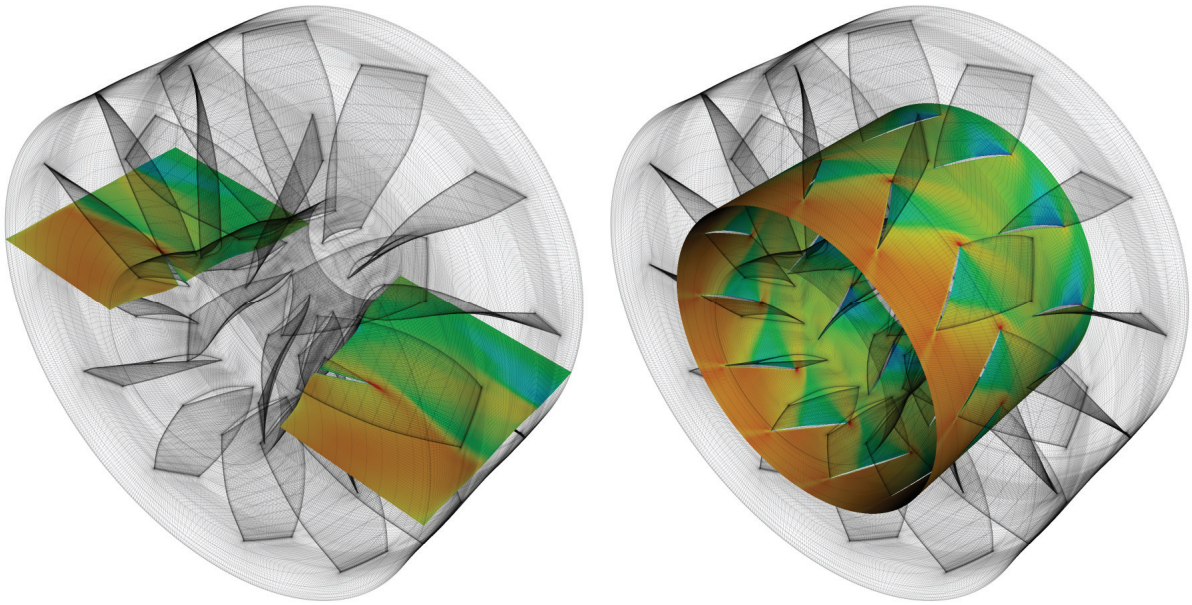

Abbildung 81: Schnittfläche und Schnittzylinder durch einen Turbinendatensatz

Dabei spricht im beschriebenen Schnittalgorithmus nichts dagegen, beliebige Schnittfunktionen zu verwenden. Gegebenenfalls muss der Algorithmus zur Bestimmung der Schnittpunkte innerhalb von Gitterzellen erweitert werden. Wie auf der rechten Seite der Abbildung 81 zu sehen, können sodann wesentlich informativere Schnittbilder erzeugt werden. Für die hier dargestellte Turbine bieten sich Zylinder- bzw. Kegeloberflächen geradezu als Schnittobjekte an. So platziert, dass deren Mittelachsen mit der Rotationsachse der Turbine übereinander liegen, kann durch Variation des Radius das Strömungsfeld sehr schnell exploriert werden.
Das hier dargestellte Beispiel dient als Motivation die folgenden Betrachtungen vorerst auf zylindrische Schnittobjekte zu konzentrieren. Grundsätzlich lassen sich die Ansätze jedoch auf beliebige analytische Schnittobjekte übertragen. Mithilfe von *Mesh-Refinement*-Verfahren (vgl. [ZORI00, AKEN02]) lassen sie sich sogar auf beliebige Oberflächennetze anwenden.

Ein Zylinder kann leicht durch wenige Eigenschaften analytisch definiert werden. Neben einem Punkt, der den Mittelpunkt der Grundfläche beschreibt, benötigt man noch einen Vektor, der die Mittelachse bestimmt. Die Länge des Vektors ergibt die Höhe des Zylinders. Der letzte Parameter, den man für die vollständige Beschreibung braucht, ist der Radius. Neben der höhenbeschränkten Definition kann man diese Eigenschaften, in Analogie zur unbeschränkten Schnittfläche, auch für die Beschreibung eines unendlich langen Schnittrohrs verwenden.
Der in Abbildung 81 (rechts) auf einem Zylinder dargestellte Skalarverlauf wurde mit dem auf VTK basierenden parallelen Schnittalgorithmus von Viracocha realisiert. VTK ermöglicht grundsätzlich die Verwendung von beliebigen, implizit definierten Schnittfunktionen. Das Ergebnis liegt dann als Dreiecksoberfläche vor, deren Auflösung in direkter Beziehung zur Anzahl der geschnittenen Datenzellen steht. Die Schnittergebnisse lassen sich nun einfach blockweise zum *VisHost* senden und darstellen. Hierbei treten allerdings einige Schwächen zutage. So muss jeweils das komplette Dreiecksnetz inklusive aller Skalar- und Normalen-informationen übertragen werden. Zudem stellt der Schnittalgorithmus eine *Black Box* dar, sodass sich Zwischenergebnisse für ein filigraneres Streaming nicht abgreifen lassen.

5.3.4.1 Einfacher Streaming-Ansatz

Da auf dem *VisHost* die Form und Position des Schnittobjekts bestimmt wird, um diese Informationen dann als Parameter der Berechnungsanforderung an den *WorkHost* zu verschicken, kann es als Geometrieobjekt direkt nach Absetzen des Extraktionsbefehls gezeichnet werden. Allerdings ist die exakte Oberflächentesselierung, die sich durch den Schnitt durch den zugrunde liegenden Datensatz ergibt, im Vorfeld nicht bekannt. Trotzdem wäre es wünschenswert bereits das Oberflächen-Mesh zu verwenden, das exakt dem Schnittergebnis entspricht. Dann könnte man sich das Übertragen des Ergebnisnetzes ersparen und müsste nur die berechneten Skalare für die einzelnen Netzpunkte versenden. Die Datenlast der Kommunikation würde sich deutlich verringern und das Zusammenfügen von Teilergebnisflächen würde entfallen.

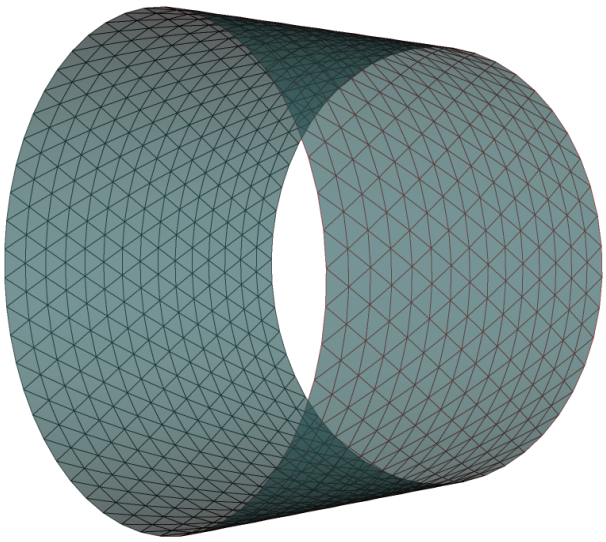

Abbildung 82: Gleichmäßig tesselierter Zylinder (Detaillierungsgrad 50)

Daher kommt nun ein alternativer Algorithmus zum Einsatz, in dem der *VisHost* neben der Schnittgeometrie auch gleich noch dessen Tesselierung vorgibt. Der Detaillierungsgrad lässt sich über einen weiteren Parameter einstellen. Wird ein Zylinder als Schnittobjekt verwendet, versucht der Algorithmus immer annähernd gleichseitige Dreiecke zu generieren, die dann direkt als *Triangle Strips* zusammengefasst werden (vergleiche Beispiel in Abbildung 82). Die Erstellung der Flächennormalen an den Mesh-Punkten geschieht zudem quasi

automatisch. So erreicht man bereits im initialen Schritt eine gleichförmige und performante Darstellung der Zylindergeometrie, allerdings noch ohne aufgetragene Skalarverläufe.
Zur Berechnung von Skalaren auf der Zylindergeometrie müssen lediglich Zylinderbasispunkt, -achse, -radius und der Detaillierungsgrad für die Tesselierung übermittelt werden. Anhand dieser Informationen lassen sich nun auch auf dem *WorkHost* alle Punkte auf der Mantelfläche bestimmen. Jeder einzelne Punkt wird sodann verwendet, um einen *Cell Search* durchzuführen, d. h. es wird diejenige Zelle gesucht, die den Punkt umschließt. Entsprechend der Position des Mesh-Punktes in der Zelle werden die Skalare an den Zellknoten gewichtet aufaddiert, sodass man den gesuchten Interpolationswert erhält. Als Ergebnis werden lediglich diese Ergebniswerte zurückgeschickt. Der *VisHost* braucht sie nur noch in das bisher leere Skalarfeld der Manteloberfläche eintragen, sodass ein Farbverlauf auf dem Zylinder erscheint. Ein Beispiel unter Verwendung des AIA-Motordatensatzes ist in Abbildung 83 dargestellt.

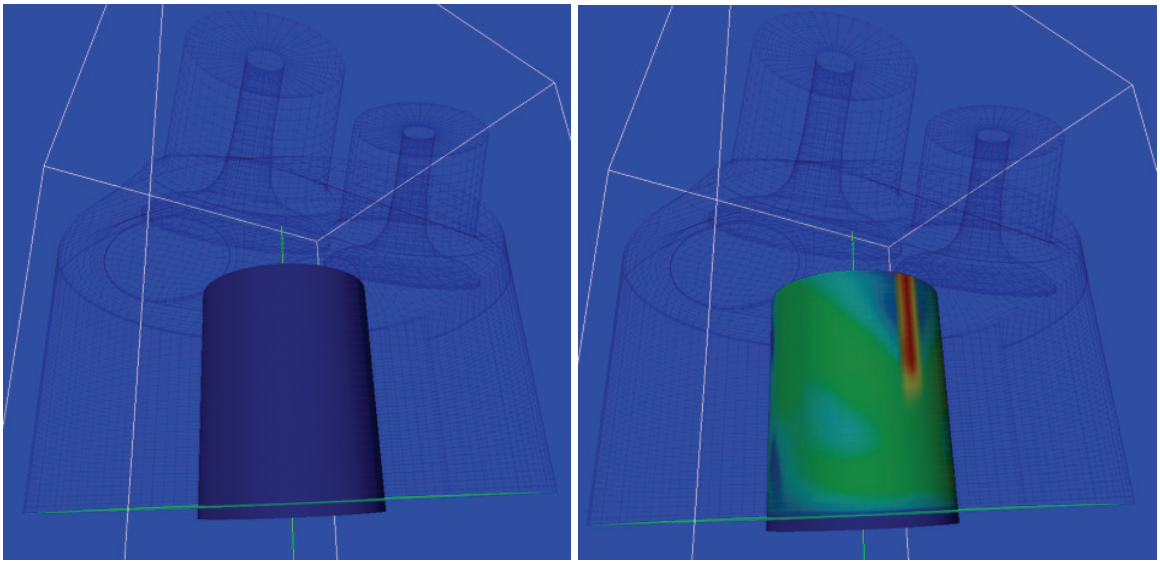

Abbildung 83: Eingebrachter Schnittzylinder in den Motordatensatz, direkt nach der Anforderung (links) und nach Beendigung der Berechnung (rechts)

Der gerade vorgestellte Algorithmus lässt sich auch recht einfach in einen Streaming-Ansatz überführen. Im Prinzip könnte jeder neu berechnete Positionswert direkt zum Visualisieren verschickt werden. Aber schon alleine aus Effizienzgründen ist es ratsamer immer eine gewisse Anzahl von Werten zuvor zu einem Paket zusammenzufassen. Da die Datenpunkte eine gewisse Ordnung aufweisen, muss nur dafür gesorgt werden, dass der *VisHost* den als ersten in das Datenpaket eingefügten Wert richtig in sein Skalarfeld einordnet. Die nachfolgenden Paketwerte werden dann alle sequenziell in das Datenarray für das Skalarfeld übernommen. Neben diesem Offset-Parameter kann noch eine Zeitinformation dazukommen, sobald das Streaming für mehrere Zeitschritte gleichzeitig erfolgt.
Um die Berechnung zu beschleunigen, stehen zwei Varianten der Parallelisierung zur Auswahl:

Blockbasiert: In der ersten Variante arbeiten alle *Worker* gleichzeitig innerhalb eines Zeitschritts am Zylinder. Dabei werden die einzelnen Dreiecksstreifen vom Basiskreis aus gleichmäßig auf die *Worker* verteilt. Dann lädt jeder *Worker* die von ihm benötigten Datenblöcke und rechnet unabhängig seine Streifen aus. Nach Beendigung der Berechnung werden die Streifen zum *VisHost* geschickt. Das Streaming ergibt sich zum einen dadurch, dass bereits berechnete Streifen schon einmal zurückgeschickt werden, und zum anderen durch die unterschiedliche Berechnungsdauer der einzelnen *Worker*.

Zeitbasiert: Die zweite Variante verteilt nicht Teile des Zylinders sondern die Verantwortung für einen Zeitschritt auf die einzelnen Worker. Es werden wiederum Dreiecksstreifen

berechnet und zurückgestreamt. Ein Vorteil dieser Verteilungsstrategie liegt darin, dass keine Blöcke mehr mehrfach geladen werden.

Viel bedeutender ist allerdings die Tatsache, dass der Anwender bei einer instationären Visualisierung vor allem an der Entwicklung von Datenfeldern über die Zeit interessiert ist. In der blockbasierten Variante springt die Visualisierung immer schnell über den gerade bearbeiteten Zeitschritt hinweg. Es kann keine vernünftige Analyse stattfinden. Vielmehr wird das kurze Aufflackern von bereits eintreffenden Daten eher noch als störend empfunden. Können aber auch schon in den nachfolgenden Zeitschritten Trends dargestellt werden, kann der Anwender deutlich früher mit der zeitvarianten Analyse beginnen. Dies gewährleistet die zweite Variante. Wie sich die beiden Streaming-Versionen zur Laufzeit unterscheiden, deuten die *Screenshots* in Abbildung 84 an.

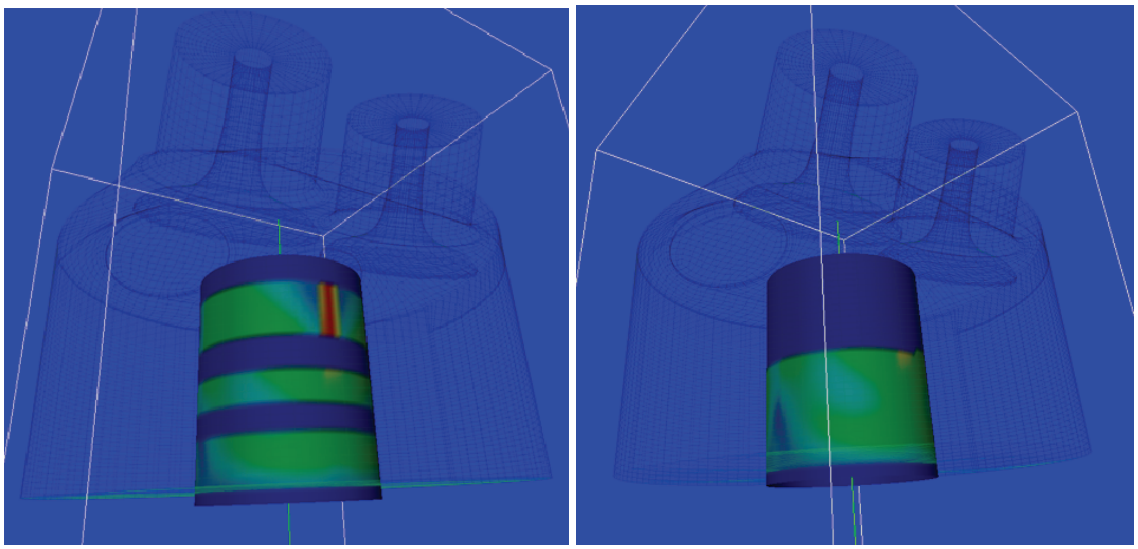

Abbildung 84: Darstellung gestreamter Skalarstreifen, Parallelisierung innerhalb eines Zeitschritts (links) und zeitschrittorientiert (rechts)

Durch die Vorgabe des Oberflächennetzes durch den *VisHost*, das für das *Point Sampling* auf dem *WorkHost* verwendet werden soll, entstehen einige bedeutende Probleme:

Auflösung der Tesselierung: Die Auflösung des Meshes wird von außen vorgegeben und kann sich nicht adaptiv an die zugrunde liegende Datensatzstruktur anpassen. Die durchschnittliche Zellgröße sowie die räumliche Verteilung der Daten sind üblicherweise im Vorfeld nicht bekannt. Eine Heuristik zur Abschätzung einer geeigneten Auflösung kann jedoch aus der erweiterten Topologiedefinition aus Abschnitt 3.4.2 hergeleitet werden.

Geometrische Begrenzungen: Mit der Definition z. B. des Zylinders werden feste geometrische Grenzen vorgegeben. Liegt das Schnittobjekt nicht im Volumen des Datensatzes, werden auch immer an außerhalb liegenden Punkten Zellsuchoperationen durchgeführt. Das gleiche Problem tritt mit Löchern im Datensatz auf (Beispiel: Schaufeln der Turbine, vgl. Abbildung 81). Bricht der Schnittalgorithmus an den bezeichneten Punkten ab, muss trotz allem ein Wert zurückgeliefert werden, damit der *VisHost* sein Skalarfeld ordnungsgemäß auffüllen kann. Es kommt dann zu Auslassungen, wie sie beispielsweise im unteren Abschnitt der in Abbildung 84 dargestellten Zylinder angedeutet sind.

5.3.4.2 Progressives Streaming

Ein einfaches LOD-Streaming lässt sich dadurch erreichen, dass man den *WorkHost* anweist mehrere Detaillierungsgrade, die zunehmend feiner werden, hintereinander durchrechnen zu

lassen. Die Ergebnisnetze werden der Reihe nach beim *VisHost* dargestellt. Um den mit diesem Ansatz einhergehenden erheblichen Overhead zu vermeiden, sind progressive Ansätze gefordert.
Der implementierte progressive Streaming-Algorithmus geht wiederum davon aus, dass auf dem *WorkHost* das vollständig triangulierte Oberflächennetz des Schnittzylinders bekannt ist. Dieser wird nun zunächst mit einer vorgegebenen Schrittweite abgetastet, sodass ein erstes grobes Ergebnisnetz an den *VisHost* geschickt werden kann. Dort wird nur der Zylinder mit entsprechend empfangenem Auflösungsgrad visualisiert. Der *WorkHost* verfeinert als nächstes nun die Sampling-Rate, berechnet jedoch nur die noch nicht bereits vorher ermittelten Skalarwerte. Sobald die nächste Auflösungsstufe vorliegt, wird diese zum *VisHost* gestreamt, der den vorherigen durch den neu erhaltenen Zylinder ersetzt. Dies wird bis zur höchsten Auflösungsstufe fortgeführt. Bei diesem Ansatz besteht das Problem, geeignete Detaillierungsstufen für das Subsampling zu finden, wobei die Zylindermantelfläche immer durch möglichst gleichseitige Dreiecke approximiert wird. Um dies zu erreichen wurde ein in Abbildung 85 verdeutlichtes Berechnungsschema entwickelt.

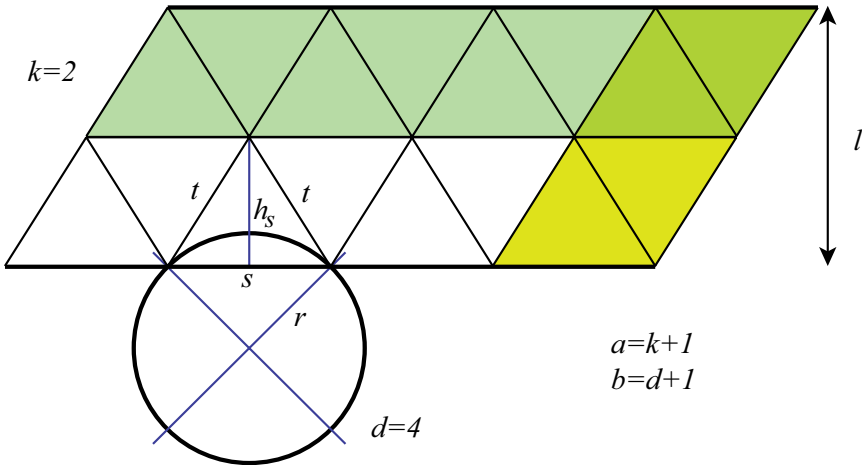

Abbildung 85: Zu einem Parallelogramm aufgerollte Zylindermantelfläche

In einem ersten Schritt wird die Grundfläche des Zylinders entsprechend des vorgegebenen Detaillierungsgrads *d* in gleichgroße Kreissegmente zerteilt. In Abhängigkeit des Radius *r* kann nun für solch ein Segment die Sehnenlänge *s* mit

$$s = 2r \cdot \sin(\pi / d) \tag{6}$$

ausgerechnet werden, die gleichzeitig die erste Seitenlänge der gesuchten Dreiecke bestimmt. Würden sich tatsächlich gleichseitige Dreiecke in Streifen über die Mantelfläche verteilen lassen, bräuchte jetzt nur noch die Höhe *h* eines gleichseitigen Dreiecks durch

$$h = s \cdot \frac{1}{2}\sqrt{3} \tag{7}$$

berechnet werden, und man würde die Höhe eines Dreiecksstreifens erhalten. Die Länge *l* der Zylindermittelachse dividiert durch die Höhe *h* würde sodann die Anzahl der Dreiecksstreifen *k* ergeben. Da jedoch in den seltensten Fällen so berechnete Streifen die Mantelfläche exakt abdecken, muss die Anzahl der Streifen auf- bzw. abgerundet werden:

$$k = \lfloor l / h + 0{,}5 \rfloor \tag{8}$$

Dadurch erhält man auch eine modifizierte Streifen- und somit eine neue Dreieckshöhe h_s:

$$h_s = l / k \tag{9}$$

Diese Dreieckshöhe bestimmt jetzt die beiden gleichlangen, bisher noch unbestimmten Seiten *t*. Man bekommt also kein gleichseitiges sondern ein gleichschenkliges Dreieck:

$$t = \sqrt{h_s^2 + \left(\frac{s}{2}\right)^2} \tag{10}$$

Wichtig für das Subsampling ist nun die Auflösung der Mantelfläche, die durch die Umkreisauflösung $b = d + 1$ und die Höhenauflösung $a = k + 1$ gegeben ist. Schneidet man den Mantel von einem Punkt am Umkreis der Grundfläche entlang der Dreiecke bis zum oberen Umkreis durch, erhält man ein Parallelogramm, das gleichmäßig durch horizontale und vertikale Dreiecksstreifen aufgeteilt wird. Fasst man jeweils zwei Dreiecke, die durch zwei zueinander quer verlaufende Streifen (in Abbildung 85 grün hinterlegt) überdeckt werden, zu einem Viereck zusammen, erhält man ein reguläres zweidimensionales Gitter. Daher können für den Zylindermantel die Subsampling-Regeln aus Abschnitt 5.3.2 angewendet werden. Das visualisierte Streaming-Ergebnis sieht dann beispielsweise wie in Abbildung 86 aus.

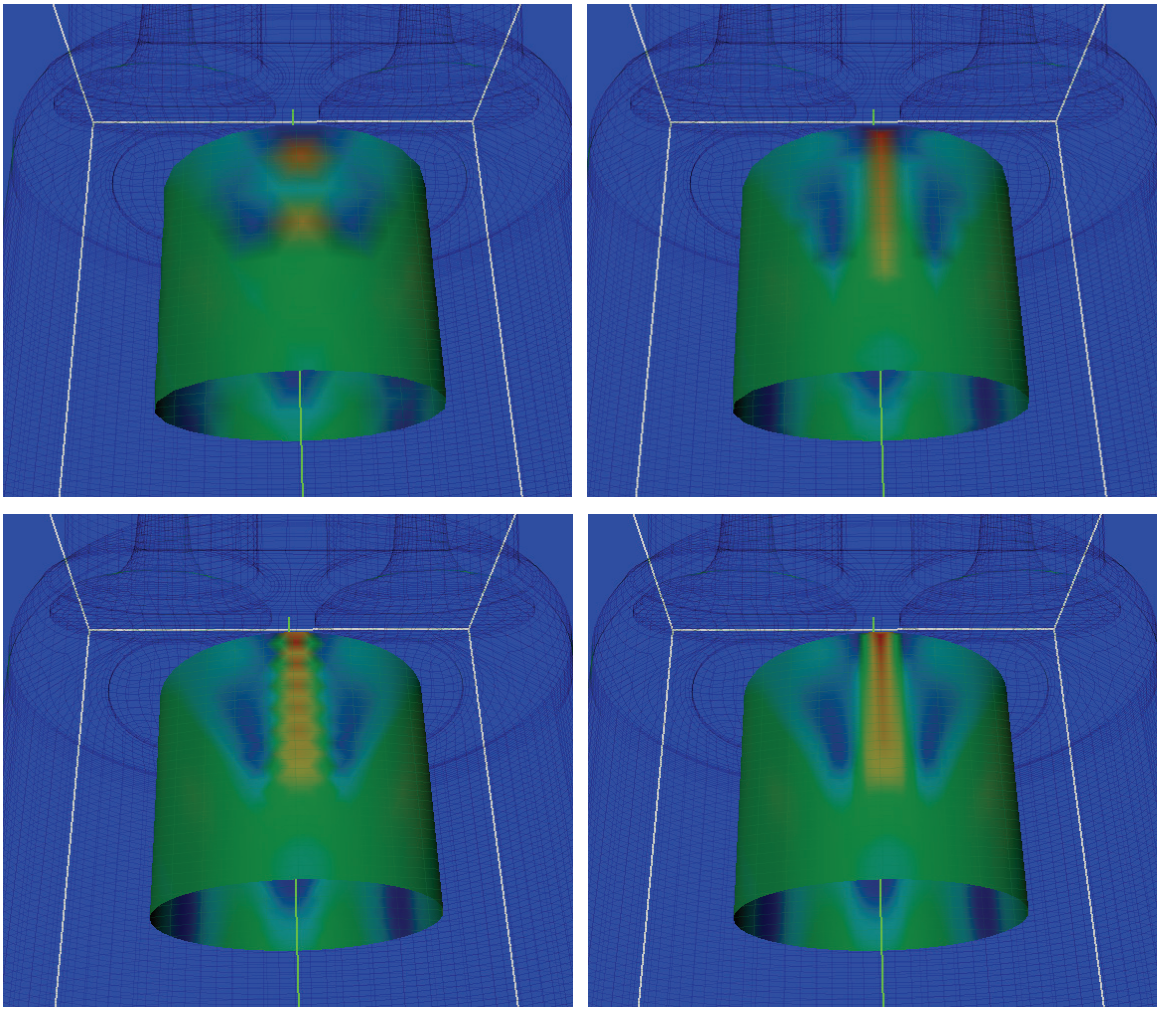

Abbildung 86: Progressiver Aufbau des Schnittzylinders

Damit der Subsampling-Ansatz funktioniert, müssen die gröber aufgelösten Detaillierungsstufen immer eine Untermenge der feiner aufgelösten Stufen darstellen, d. h. alle Punkte niedriger Stufen müssen deckungsgleich mit Punkten höherer Stufen sein. Es existiert aber nur ein Steuerungsparameter, nämlich der Detaillierungsgrad *d*. Um die exakte Punktüberdeckung zu erreichen, muss über diesen Parameter gewährleistet werden, dass von der untersten Auflösungsstufe ausgehend in beide Kantenrichtungen des Parallelogramms die Abtastrate gleichmäßig zunimmt. Dadurch erhält man eine affine Abbildung der Ausgangsdreiecke, d. h. die Winkel und die Seitenverhältnisse der neu entstehenden Dreiecke sind identisch, womit die Bedingung erfüllt ist.

Ein Beispiel soll die Zusammenhänge der einzelnen Parameter verdeutlichen. Die Höhe l des Zylinders sei 3,5 und der Radius r seiner Grundfläche betrage 2,25. Der gröbste Detaillierungsgrad d sei mit 10 angegeben. Dann ergibt sich eine Mantelflächenauflösung von $a = 4$ und $b = 11$. Des Weiteren seien 3 Verfeinerungen gewünscht. Die ersten beiden Verfeinerungsstufen sollen um jeweils das dreifache, die letzte dagegen um das doppelte abgetastet werden. Nach Gleichung 5 würde man folgende Detaillierungsgrade für die einzelnen Auflösungsstufen erhalten:

$$\begin{aligned} & d_3 = 10 \Rightarrow a_3 = 4 = 1 + 3^1,\ b_3 = 11 = 1 + 2^1 \cdot 5^1 \\ & a_2 = 1 + 3^2 = 10,\ b_2 = 1 + 2^1 \cdot 5^1 \cdot 3^1 = 31 \Rightarrow d_2 = b_2 - 1 = 30 \\ & a_1 = 1 + 3^3 = 28,\ b_1 = 1 + 2^1 \cdot 5^1 \cdot 3^2 = 91 \Rightarrow d_1 = b_1 - 1 = 90 \\ & a_0 = 1 + 3^3 \cdot 2^1 = 55,\ b_0 = 1 + 2^2 \cdot 5^1 \cdot 3^2 = 181 \Rightarrow d_0 = b_1 - 1 = 180 \end{aligned} \tag{11}$$

Bei niedrigen Auflösungen erscheint der visualisierte Zylinder trotz berechneter Flächennormalen recht grob approximiert. Daher sieht eine Variante des vorgestellten Subsampling-Ansatzes nicht nur die Verwendung des vollständig aufgelösten Zylinders für die Berechnung sondern auch für die Visualisierung vor. Kommen dann die ersten grob abgetasteten Ergebnisse beim *VisHost* an, werden die Punktewerte an die richtigen Stellen gemappt. Die zwischen den bereits berechneten Punkten liegenden Werte werden erst einmal durch Interpolation bestimmt. Man erhält somit einen hoch aufgelösten Zylinder mit einer „verwischten“ Oberfläche.

5.3.4.3 Laufzeitmessungen

Als Testszenario kam erneut der Motor-Datensatz zum Einsatz, wovon 16 aufeinander folgende Zeitschritte für die Extraktion ausgewählt wurden. Bis zu 16 *Worker* berechneten die Teilergebnisse. Während die SunFire-6800 für die Berechnung zuständig war, wurde als Visualisierungsrechner eine SunFire-V880z mit vier 900 MHz-CPUs und zwei XVR-4000-Graphikkarten eingesetzt.
Für Messungen stand zum einen der auf die implizite Schnittfunktion aufsetzende Ansatz (*Simple Cut*) zur Verfügung. Hier berechnete immer ein *Worker* einen kompletten Zeitschritt und übertrug sodann seine berechneten Schnittdaten, bevor er, sofern noch vorhanden, mit dem nächsten Zeitschritt fortfuhr. Zum anderen kam die Streaming-Strategie zum Einsatz, welche die Zeitschritte unter den *Workern* verteilt (*Streamed Cut*). In dieser Methode wurde ein wie in Abschnitt 5.3.4.2 definierter Schnittzylinder eingesetzt, der eine Auflösung aufwies, die optisch ähnlich ansprechende Ergebnisse lieferte wie der erste Ansatz. Zudem wurden insgesamt 8 Datenpakete pro Zylinder gestreamt.

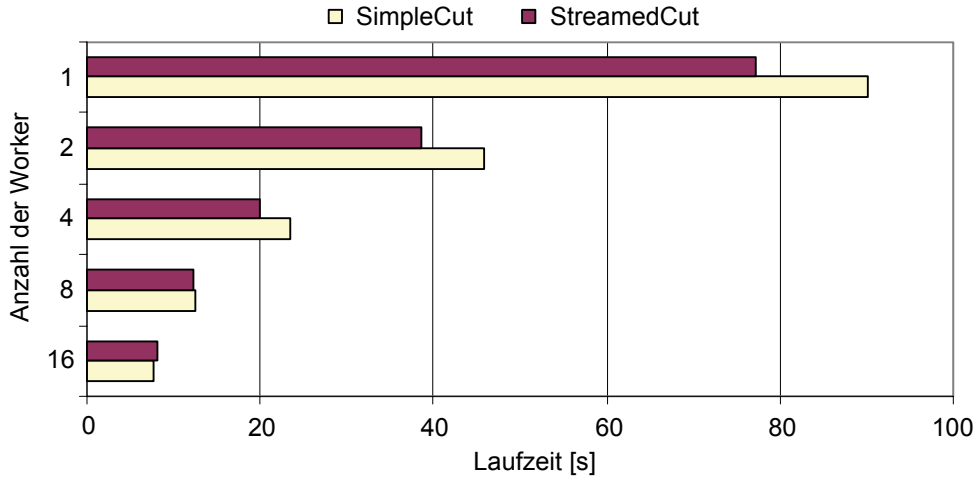

Abbildung 87: Gesamtlaufzeiten beider Schnittansätze

Die Gesamtlaufzeiten, in Abbildung 87 angegeben, weisen für beide verglichenen Schnittansätze gute Skalierungen auf. Der Streaming-Ansatz ist anfänglich auch schneller, was sich hauptsächlich durch die verringerte Punktanzahl im regulären Zylinder gegenüber der adaptiven Schnittfläche begründet. Erst mit 16 *Workern* ist der adaptive Ansatz geringfügig schneller. Hier schlägt sich der zunehmende Anteil für die Kommunikation auf die Gesamtlaufzeit des Streaming-Ansatzes nieder.
In Abbildung 88 sind die Latenzzeiten angegeben, d. h. die Zeit, die vom Absetzen des Berechnungsbefehls bis zum Eintreffen der ersten Ergebnisse vergeht. Entsprechend der gemessenen Gesamtzeit liefert der *Simple Cut* seinen ersten Zeitschritt nach der erwarteten Laufzeit ab. Wegen der unbalancierten Datenlasten unter den *Workern* schwankt diese Latenzzeit etwas bei der unterschiedlichen Anzahl verwendeter *Worker*.

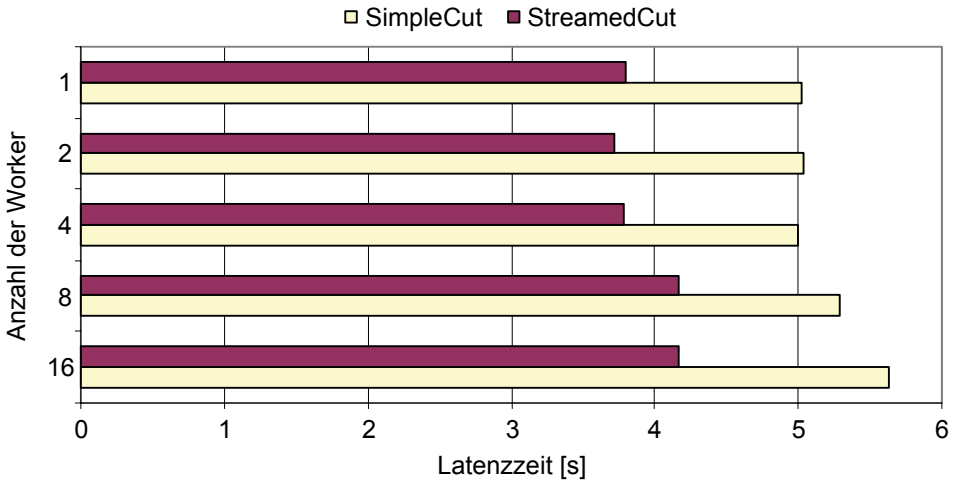

Abbildung 88: Vergleich der Latenzzeiten

Der *Streamed Cut*, der acht Zwischenergebnisse pro Zeitschritt streamt, sollte eigentlich, gemessen an seiner Gesamtlaufzeit, eine deutlich kürzere Latenzzeit aufweisen. Hier macht sich allerdings die Initialisierung der Berechnung bemerkbar, sodass die ersten Teilergebnisse erst recht spät präsentiert werden können. Danach wird jedoch einigermaßen zügig gestreamt. Die erreichte Latenzzeit liegt daher nur in etwa 25 Prozent unter der von der Standardschnittmethode.
Auf der Visualisierungsseite sind die Vorteile des Streaming-Ansatzes jedoch sehr viel deutlicher erkennbar (vgl. Abbildung 89). Da beim Streaming-Ansatz die Zuordnung der eintreffenden Teildaten direkt durchführbar ist, zudem das Datenvolumen insgesamt deutlich gesenkt wurde, schneidet hier der *Streamed Cut* nun erheblich besser ab.

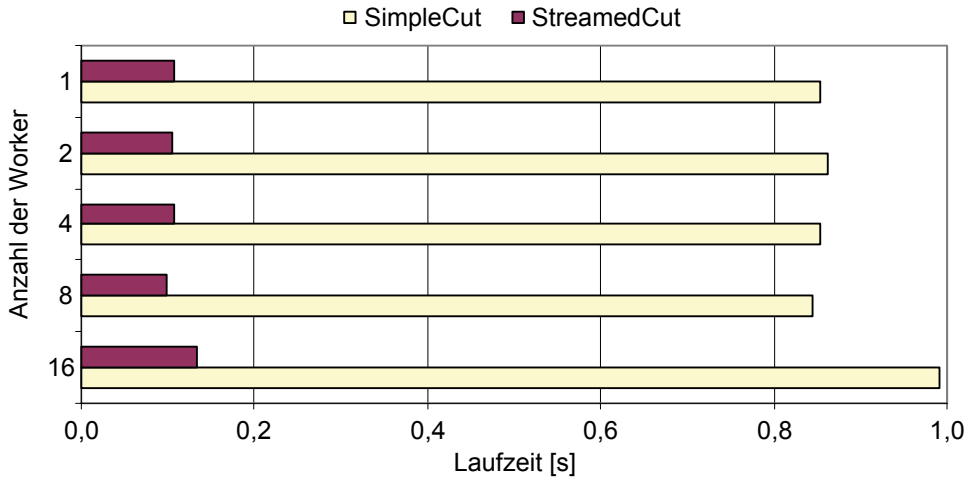

Abbildung 89: Benötigte Zeit auf dem *VisHost* zur Datenaufbereitung

5.3.5 Streaming für das Partikelverfolgung

Die Berechnung von Partikelbahnen ist durch seinen inkrementellen Charakter, d. h. durch die nacheinander einzeln durchzuführenden Positionsbestimmungen, aufwändiger zu einer streamingfähigen Version umzugestalten als beispielsweise Schnittflächen. So ist ein einfaches blockweises Streaming nicht ohne weiteres anwendbar, da nach dem Verlassen eines Blockes die Partikel durchaus erneut in den Block eintreten können, sich also weitere dem Block zugehörige Daten ergeben. Daher bieten sich alternativ dazu folgende Verfahren an:

Einzelne Partikelpositionen: Besonders für stationäre Untersuchungen recht intuitiv ist der Ansatz, der eine gewisse Anzahl von Partikelpositionen berechnet und diese zum *VisHost* sendet. Dort können somit bereits die ersten Bewegungen der Partikel verfolgt und ggf. durch Neusetzen von Saatpunkten korrigiert werden.

Prozessweises Streaming: In diesem Verfahren berechnet jeder Worker immer eine komplette Partikelbahn, bevor diese an den VisHost gestreamt wird. Sodann wird mit der Berechnung der nächsten Partikelbahn fortgesetzt. Dieser Ansatz eignet sich besonders dann, wenn eine große Anzahl von Partikeln in einer instationären Strömung verfolgt werden sollen.

***Level-Of-Detail*-Ansatz**: LOD-Extraktionen lassen sich durch einfache Manipulation der Integration erreichen. Variieren lassen sich sowohl die Integrationsverfahren als auch die einzelnen Integrationsparameter (vgl. hierzu [FRUE97], S. 84ff). Beispielsweise kann mit einem einfachen Euleransatz und grober Schrittweite startet werden, sich dann erneut die Partikelbahn mithilfe von Runge-Kutta 2. Ordnung verfolgen lassen, bis man abschließend durch den Einsatz der Fehlberg-Integration [PRES02, CASH90] mit adaptiver Schrittweitenkontrolle eine Partikelbahn mit geringster Fehlerordnung erhält.

Auf Kosten der Geschwindigkeit erhält man somit immer genauere Ergebnisse. Da in Abhängigkeit des Strömungsfeldes bereits mit einfacheren Verfahren und gröberer Schrittweite gute Ergebnisse erreicht werden können, lässt sich anhand der Abweichungen bereits berechneter Bahnen abschätzen, ob eine weitere Verfeinerung und somit eine Verlängerung der Gesamtlaufzeit überhaupt nötig ist. Der Algorithmus könnte sodann frühzeitig beendet werden.

Subsampling On Demand: Dieses LOD-Verfahren lässt sich recht einfach auf strukturierte Datensätze anwenden. Es wird dabei auf der Originalauflösung gearbeitet, aber entsprechend einer vorher bestimmten Sampling-Schrittweite werden mehrere Hexaederzellen zu einer einzigen großen Hexaderzelle zusammengefasst. Auf diesen während der Berechnung erzeugbaren Metazellen wird nun integriert. Nach der groben Partikelverfolgung kann ein genauerer Pfad durch die Verringerung der Sample-Rate erfolgen, bis die Originalauflösung für die Integration verwendet wird (vgl. hierzu auch Abschnitt 5.3.2 und Abbildung 71).

In stationären regulären Gittern kann sogar auf die numerische Integration verzichtet werden. Dieser alternative Ansatz geht zurück auf die Tetraedisierung von Hexaederzellen für die Stromlinienberechnung, wie sie von Kenwright und Lane in [KENW96] vorgestellt wurde. Durch die Interpolation aufgrund von baryzentrischen Koordinaten kann mit der Angabe der Eintrittsstelle des Partikels direkt sein Austrittpunkt bestimmt werden.

Ein interessanter Visualisierungsaspekt betrifft die LOD-Darstellung während des Streamings. Da es zwischen den einzelnen LOD-Stufen zu erheblichen Pfadabweichungen der Partikel kommen kann, führt ein einfaches Durchschalten durch die Auflösungsstufen verstärkt zu *Popping*-Artefakten. Daher ist es visuell ansprechbarer die Partikelpositionen von einem Level zum nächsten durch Interpolation überblenden zu lassen. Laufen die Bahnen stark auseinander, kommt es zwar verstärkt zu Peitscheneffekten (siehe Abbildung 90), was

allerdings recht gut den temporären Charakter gestreamter Daten verdeutlicht. Voraussetzung für die flüssige Visualisierung ist jedoch die Abbildbarkeit der berechneten Partikelpositionen. Durch die zeitliche Normierung der Abstände im verwendeten Algorithmus ist diese Bedingung erfüllt.

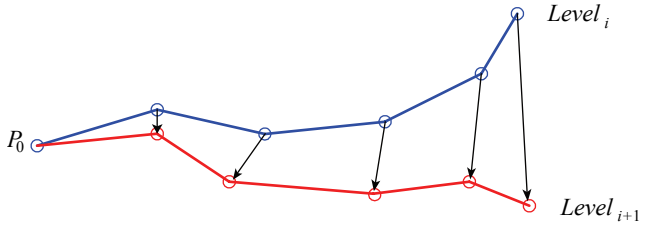

Abbildung 90: Interpolation zwischen zwei LOD-Partikelbahnen mit Peitscheneffekt

Die weiteren in der Streaming-Matrix aufgelisteten Verfahren lassen sich nicht oder kaum sinnvoll für Partikelverfolgung einsetzen. So ist es z. B. zwar grundsätzlich möglich die Integrationsparameter betrachterabhängig zu variieren, das Ergebnis aber ist für eine explorative Analyse in virtuellen Umgebungen kaum verwendbar. *Selective Refinement* macht dagegen auf der Visualisierungsseite durchaus Sinn, insbesondere wenn massive Partikelströme als Polylinien dargestellt werden sollen (vgl. hierzu auch Abschnitt 5.2.3).
Aufgrund der iterativen Eigenschaft von Integrationsverfahren scheiden auch progressive Ansätze für die Extraktion aus. Wird mit einem einfachen Integrationsverfahren, das eine hohe Fehlerordnung aufweist, eine erste grobe Approximation der Partikelbahn berechnet, lassen sich zwar einzelne Liniensegmente herausgreifen, startet man jedoch ein Verfahren niedrigerer Fehlerordnung am Startpunkt eines dieser Liniensegmente, wird das neu berechnete Segment von der vorher berechneten Bahn abweichen. Die Endpunkte der beiden Auflösungsstufen werden nicht deckungsgleich sein. Somit stimmen Endpunkt des aktuell verfeinerten und der Startpunkt des nächsten Segments auch nicht mehr überein. Es kommt damit unvermeidlich dazu, dass die Partikelbahn aufbricht.
Bei der Verwendung von aus Multi-Blöcken bestehenden MR-Datensätzen gibt es ebenfalls Probleme, die beim Schnitt und bei Isoflächen nicht auftreten. Durch naives Subsampling besteht nämlich die Gefahr, dass bei den Übergängen zwischen Datenblöcken Überlappungen oder Lücken entstehen (vgl. Abschnitt 5.3.2.1). Läuft das Integrationsverfahren in diese Bereiche hinein, bricht es im schlimmsten Fall ab. Überlappungen lassen sich noch abfangen, indem diejenige Zelle für die Integration verwendet wird, die als erstes gefunden wurde. Bei Lücken müsste der Integrationsalgorithmus jedoch erkennen, dass es sich hier um eine fehlerhafte Stelle handelt und somit das Partikel nicht das Strömungsfeld über Randzellen verlassen hat. Sodann müssten zur Überbrückung der Lücke Datenpunkte des nächsten Blocks identifiziert werden, um anschließend mit der Integration fortfahren zu können. Wegen dieses Aufwands wurde Partikelverfolgung für Multi-Block-Datensätze bei den in diesem Kapitel beschriebenen LOD-Verfahren nicht berücksichtigt.

6 Zusammenfassung und Ausblick

An der RWTH Aachen werden VR-Anwendungen vor allem mit der VR-Software ViSTA erstellt. Um auch wissenschaftliche Visualisierung in virtuellen Umgebungen integrieren zu können, wurde ein Zusatzmodul mit dem Namen ViSTA FlowLib entwickelt. Damit können bereits instationäre Vorgänge visualisiert werden; bei etwas größeren Datensätzen lassen sich jedoch schnell die in der Einleitung definierten Interaktionskriterien nicht mehr erfüllen. Besonders das Unterschreiten der Mindestbildwiederholungsrate macht das Arbeiten in interaktiven Umgebungen unmöglich.
Da allerdings besonders im Bereich der Strömungssimulation extrem große Datenmengen anfallen, sich die Analyse in virtuellen Umgebungen aber als besonders effizient herausgestellt hat, mussten Mechanismen untersucht werden, die dennoch eine interaktive Exploration ermöglichen sollten. Daher wurde ein verteiltes Postprocessing-System integriert, das die Visualisierung entlastet und den berechnungsintensiven Anteil der Merkmalsextraktion auf Viracocha, dem parallelen Berechnungs-*Backend*, auslagert. Bereits durch diese Entkopplung konnte erreicht werden, dass die ersten beiden Interaktionskriterien, die ein flüssiges Arbeiten an VR-Arbeitsplätzen gewährleisten, eingehalten werden konnten. Zudem war es nun möglich, durch die isolierte Betrachtung wesentlich einfacher optimierte Algorithmen zum einen für die Visualisierung und zum anderen für die Extraktion zu entwickeln. Der reibungslose Datenfluss auf beiden Seiten wurde dabei immer durch das integrative Konzept der Visualisierungspipeline aufrechterhalten.
Eine wichtige Rahmenbedingung war der flexible Einsatz des Gesamtsystems. Dies konnte vor allem durch das plattformunabhängige Design und durch die Verwendung von standardisierten Parallelisierungsschnittstellen erreicht werden. Somit liegt eine Postprocessing-Umgebung vor, die sowohl den Ansprüchen einer schnellen Exploration kleiner bis mittlerer Simulationsgrößen vor Ort an den Instituten erlaubt als auch die Analyse sehr großer Datensätze unter Zuhilfenahme moderner Hochleistungsrechner und großer immersiver Arbeitsumgebungen ermöglicht. Diese für den Anwender transparente Skalierung ist eines der wichtigsten Merkmale des präsentierten Systems und konnte die Akzeptanz von VR-Systemen für die Strömungsanalyse deutlich erhöhen.
Neben dem Design der Systemarchitektur stand die Analyse und prototypische Implementierung von parallelen CFD-Postprocessing-Algorithmen im Vordergrund. Im Zentrum der Bewertung stand dabei das dritte Interaktionskriterium, das die Reaktionszeit des Postprocessing-Systems beurteilt. Um diese auch als Latenzzeit definierte Größe zu reduzieren, wurde primär auf einen *Speed-up* durch Parallelisierung gesetzt. Dabei wurde auf die unterschiedlichen Anforderungen der verwendeten Extraktionsalgorithmen eingegangen und Strategien evaluiert, die die Balancierung und Skalierung deutlich verbesserten. Hier konnte in erster Linie der Datenmanager von Viracocha durch Caching und Prefetching-Ansätzen eine weitere Verbesserung bewirken. Ergänzende Ansätze wie optimierte Ladestrategien und der *Data Service* konnten weitere Performancepotentiale aufzeigen.
Neben all diesen Ansätzen, die in Abhängigkeit des untersuchten Datensatzes und der eingesetzten Algorithmen bereits erhebliche Laufzeitgewinne erbrachten, hatte jedoch das integrierte Streaming, das im letzten Kapitel im Vordergrund der Evaluierung stand, den bedeutendsten Einfluss auf die sekundäre Reaktionszeit. In den folgenden Abschnitten werden die wesentlichen Resultate der drei Bereiche Parallelisierung, Datenmanagement und Streaming nochmals zusammengefasst und bewertet.

6.1 Viracocha

Durch das modulare, objekt-orientierte Gesamtkonzept von ViSTA konnte Viracocha schnell in das bestehende VR-System integriert werden. Zudem standen weitere Basisbibliotheken von Drittanbietern zur Verfügung, die eine flexible und effiziente Verwendung garantierten ohne der Entwicklung eigener Algorithmen entgegenzustehen. Der Einsatz auf HPC-Clustern mit *Shared-Memory*-Anbindung sowie auf PC-Clustern mit *Distributed-Memory*-Architektur wurde damit gewährleistet. Durch die konsequente Umsetzung von Designmustern ist es zudem gelungen, die Integration neuer Extraktionsmethoden deutlich zu vereinfachen. Vor allem das Schichtenkonzept mit eindeutig definierten Schnittstellen erlaubt es schnell neue, bzw. optimierte Strategien für Kommunikation, Parallelisierung und Merkmalsberechnung zu entwickeln und anschließend zu validieren. Damit steht ein flexibles Gesamtkonzept zur Verfügung, das besonders gut geeignet ist die Fragestellungen aus Kapitel 1 zu untersuchen und zu beantworten.
Die Organisation der Parallelisierung durch die Definition von *Scheduler* und mehreren *Workern*, die sich zu Berechnungsgruppen zusammenschließen können, hat sich ebenfalls als sinnvoll erwiesen. Durch den *Scheduler Task*, der die Koordinierung der Berechnung übernimmt, können recht individuelle Berechnungsabläufe implementiert werden, die weit über die Möglichkeiten der Pipeline-Parallelisierung anderer Visualisierungssysteme hinausgehen. Für die Standardextraktionsverfahren, wie sie im CFD-Postprocessing normalerweise auftreten, reicht das integrierte *Master-Worker*-Konzept bereits häufig aus. Anwendungsbeispiele unterstreichen die Effizienz dieser Verteilungsstrategie.
Die bevorzugt untersuchte Multi-Block-Datenstruktur lässt sich besonders gut für angepasste Parallelisierungsstrategien verwenden. Die entwickelten Ansätze konnten dazu beitragen unter Zuhilfenahme von als Metadaten abgespeicherten Topologieinformationen die Ladelast zu reduzieren. Dabei kamen hierarchisch angeordnete Entscheidungsstrategien zum Einsatz. Hiervon konnten vor allem Partikelverfolgungsalgorithmen profitieren.
Während der Analyse von Parallelisierungspotentialen für Extraktionsalgorithmen konnten die wesentlichen Laufzeitanteile identifiziert und die Interdependenz zwischen Balancierung und Datensatzpartitionierung nachgewiesen werden. Aber auch die Bedeutung der unterschiedlichen Berechnungslast pro Block ließ sich belegen. Die Integration des *Round-Robin*-Ansatzes für Strom- und Bahnlinien führte hier zu einer deutlich verbesserten Skalierung des Parallelisierungsalgorithmus. Eine zusätzliche dynamische Zuordnung von Blöcken konnte das Laufzeitverhalten noch weiter verbessern.
Besonders überzeugend war der Einsatz von *Nested OpenMP*. Hiermit war es möglich eine über mehrere Stufen verschachtelte Parallelisierung mit dynamischer Lastverteilung zu realisieren. Als besonders gut geeignet hat sich dabei die Berechnung von kritischen Punkten dargestellt, da hier eine relativ hohe Berechnungslast und ein ohne Nachbarschafts-beziehungen auskommender, zellbasierter Algorithmus zum Einsatz kam. Zwar ist die Verwendung von OpenMP auf Rechnerknoten mit *Shared Memory* beschränkt, integriert sich jedoch sehr gut als Bestandteil eines hybriden Ansatzes in Viracocha. Zudem trat hier eine Sättigung der Skalierung, wie bei den rein MPI-basierten Ansätzen zum Teil beobachtbar, nicht auf, sodass ein sehr hoher *Speed-up* unter Ausnutzung aller Prozessoren eines SunFire-6800-Knotens erreicht werden konnte.

6.2 Datenmanagement

Um der bestehenden Ladelast und den damit verbundenen Balancierungsproblemen, die bei einigen Extraktionsalgorithmen beobachtet werden konnten, entgegenzutreten, wurde das *Viracocha Data Management System* (VDMS) entwickelt. Die Integration konnte vom Software-Design profitieren und platziert sich für den Extraktionsalgorithmus weitestgehend

transparent in die Organisationsschicht. Als *Middleware* ausgelegt, stellt es daher die zusätzlichen Dienste im Wesentlichen automatisch zur Verfügung. Die entwickelten VDMS-Methoden lassen sich dabei in zwei Bereiche einteilen. Die erste Gruppe sorgt für eine Reduzierung von Latenzen beim Zugriff auf Daten, indem sie im Hauptspeicher vorgehalten werden. Die zweite Klasse von Verfahren beschäftigt sich mit Ladestrategien.
In der ersten Gruppe befinden sich Algorithmen, die zeitliche Lokalität von Benutzeranfragen ausnutzen. Ergänzt wird dies durch die räumliche Lokalität, die sich durch die Topologie der Datenblöcke untereinander ergibt. Anhand dieser Informationen und durch das Benutzerverhalten versucht das Datenmanagement Vorhersagen zu treffen, welche Daten vermutlich als nächstes verwendet werden. Diese hält das VDMS bevorzugt für die weitere Verwendung im Cache. Bei den eingesetzten Caching-Ersetzungsstrategien schnitten die frequenzbasierten Verfahren besser als die anderen ab. Überzeugen konnte vor allem das *Frequency Based Replacement* (FBR), das die Blockersetzung in den Testläufen fast halbierte.
Aber nicht nur Ersetzungsstrategien haben Einfluss auf die Effizienz von Caches. Das mit dem Ziel der Berechnungsbalancierung entwickelte *Data Service* bündelt das Wissen über den Inhalt aller Caches bei der Server-Komponente des VDMS. Durch einige Testmessungen mit Isoflächen- und Wirbelmantelberechnungen könnte der balancierende Einfluss dieser Strategie belegt werden.
Die zweite große Gruppe der VDMS-Methoden bezieht sich auf das Laden von Datenblöcken. Als besonders effizient hat sich das Prefetching erwiesen, das versucht Daten in den Cache zu laden noch bevor sie vom Algorithmus benötigt werden. In den Testfällen konnte es fast immer sinnvoll eingesetzt werden und eliminierte bis zu 40% der ohne Prefetching entstandenen *Cache Misses*. Selbst so einfache Ansätze wie die sequentiellen Strategien lieferten gute Ergebnisse, was vor allem der statischen Struktur der Datenblockanordnung gängiger CFD-Datensätze zu verdanken ist. Besonders innovativ sind die wahrscheinlichkeitsbasierten Markov-Prefetcher, die ihr Wissen aus zeitlichen Zugriffserfahrungen gewinnen. Daher sind sie insbesondere für die Partikelverfolgung geeignet. Die unterschiedlichen hierfür entwickelten Ansätze konnten die vielfältigen Möglichkeiten des Markov-Prefetchings aufzeigen. Eine deutliche Verbesserung der Vorhersagen während der Lernphase wurde durch die Vorgabe geeigneter Initialisierungs-graphen erreicht.
Bedingt durch *Cache Misses* oder der Datenanforderung durch das Prefetching werden Ladestrategien zum Datenladen eingesetzt. Drei der vier vorgestellten Methoden holen sich die angeforderten Blöcke vom Sekundärspeicher, während die vierte in Form von kooperativen Caches Rohdaten zwischen Knoten verschickt. Die erzielten Bandbreiten waren stark von der zugrunde liegenden Hardware abhängig, wobei auch die entstehende Ladelast Einfluss auf den Erfolg einer Ladestrategie hatte. Kam beim Einsatz des kooperativen Ansatzes *Shared Memory* für den Datenaustausch zum Einsatz, dann konnte diese Ladestrategie besonders dann überzeugen, wenn die anderen Strategien eine erhöhte Ladelast aufwiesen.
Eine Sonderrolle nehmen kollektive Ladestrategien ein. In Anlehnung an optimierte I/O-Techniken, wie sie in den beschriebenen parallelen Datenmanagementsystemen zum Einsatz kommen, wurden gemeinsame Datenzugriffsoperationen integriert. Diese sollen dafür sorgen, dass Festplattenzugriffszeiten minimiert und Datenzuweisungen optimiert werden. Obwohl MPI-2 solche kollektiven I/O-Funktionen mit ROMIO bereits anbietet, konnte die Effizienz auf dem verwendeten SunFire-Cluster nicht belegt werden. Das lag ausschließlich daran, dass kein paralleles Dateisystem vorhanden war. Da Viracocha das Ziel verfolgt plattformunab-hängig zu sein, stehen die Funktionalitäten trotz allem solchen Hochleistungsrechnern zur Verfügung, die ein entsprechendes Dateisystem besitzen.
Üblicherweise arbeitet das VDMS im Hintergrund, und der Algorithmus muss nicht entsprechend modifiziert werden, um seine Dienste in Anspruch nehmen zu können. Dennoch

kann es Sinn machen angepasste Extraktionsmethoden zu entwickeln, wodurch sich weitergehende Optimierungen erreichen lassen. Das erste präsentierte Beispiel beschrieb das Code-Prefetching, das seinen Vorteil durch algorithmusspezifisches Wissen über Datenzuordnung und Berechnungsreihenfolgen ziehen kann. Im zweiten Beispiel wurde anhand von Isoflächenextraktion demonstriert, wie die Datenlast durch Intervallbäume und Datensatzzerlegungsstrategien auf unter 10% gedrückt werden konnte.

6.3 Streaming

Lag der Schwerpunkt der bisher untersuchten Ansätze auf der Verkürzung der Gesamtlaufzeit einer Berechnung, wird durch die integrierten Streaming-Ansätze eine verbesserte interaktive Datenexploration innerhalb virtueller Umgebungen angestrebt. Die Integration in das Gesamtsystem wurde dargestellt und eine Klassifizierung von Streaming-Algorithmen vorgenommen. Dieses Klassifizierungsschema wurde erneut aufgegriffen, um in der Streaming-Matrix eine Zuordnung streaming-fähiger Postprocessing-Algorithmen vornehmen zu können. Diese Matrix stellte den Ausgangspunkt für die darauf folgenden Untersuchungen exemplarischer Ansätze dar. Als Testfälle kamen erneut Schnitt- und Isoflächen, Wirbelregionen und Partikelbahnen zum Einsatz. Alle Extraktionsverfahren ließen sich schnell als einfache Streaming-Varianten, die Teildaten versenden, implementieren.
Der erste der untersuchten Ansätze verwendete einen auf Basis von Subsampling-Verfahren erstellten Multiresolution-Datensatz. Dessen erste Auflösungsstufe konnte sehr schnell eingeladen und entsprechend grob aufgelöste Isoflächen berechnet werden. Da es sich hier um einen Multi-Block-Datensatz handelte, traten *Cracks* in den als Zwischenstufen gestreamten Isoflächen auf, bis abschließend der Originaldatensatz verwendet wurde und fehlerfreie Endergebnisse geliefert werden konnten. Durch den Sampling-Ansatz mithilfe der Primfaktorenzerlegung waren aber auch die Zwischenresultate aussagekräftig und angesichts des temporären Charakters störten die *Cracks* kaum.
Bei den Streaming-Ansätzen, die bereits berechnete Teildaten zum Visualisierungsrechner schicken, kamen ein blockweise streamender und ein blinkwinkelabhängiger Isoflächen-extraktionsalgorithmus zum Einsatz. Besonders der letztere ist sehr gut für immersive Umgebungen geeignet. Im günstigsten Fall werden nachträglich eintreffende Datenteile durch bereits visualisierte Fragmente verdeckt, sodass das Streaming gar nicht mehr wahrgenommen wird. Beide Ansätze sorgten für sehr kurze Reaktionszeiten. Während das Block-Streaming bereits innerhalb von fünf Sekunden Teildaten einer Wirbelstruktur liefern konnte, brauchte der *View-Dependent*-Ansatz nicht einmal eine Sekunde für die ersten Isoflächenteile.
Die beschriebenen auf Schnittflächen basierenden Streaming-Verfahren verwenden nicht das Standardverfahren mithilfe von analytischen Schnittfunktionen, sondern verfolgen Sampling-Ansätze. Die Vorteile liegen in der unverzüglichen Darstellung des finalen Schnittobjekts (ohne berechneter Skalarwerte) und der schnellen Verarbeitung von eintreffenden Streaming-Daten auf dem Visualisierungsrechner. Neben Teildaten-Streaming konnte aber auch ein progressiver Ansatz realisiert werden.
Dies ist deswegen besonders hervorzuheben, da sich die Umsetzung von progressiven Streaming-Ansätzen als außerordentlich schwierig herausgestellt hat. Das Hauptproblem ist die Bestimmung von progressiven Zwischenschritten während der Berechnung. Die meisten publizierten Ansätze gehen von der feinsten Auflösung aus und bestimmen sodann die gröberen Stufen. Ein *Bottom-Up*-Ansatz, der für das Streaming mit kurzen Latenzzeiten benötigt wird, ist somit für die meisten CFD-Extraktionsverfahren nicht ohne weiteres möglich.
Eine Ausnahme stellte die Berechnung von Schnitten dar, da hierfür häufig im Vorfeld bereits die Ergebnisgeometrien ermittelt werden können. Unterschiedlich stark tesselierte Schittobjekte ermöglichten so das Abtasten von Skalarwerten im Strömungsfeld für den

Einsatz in einem progressiven Streaming. Durch die Verwendung von iterativen Tesselierungsansätzen ließ sich eine progressive Übertragung von Schnittdaten ermöglichen und zudem die zu streamende Gesamtdatenlast deutlich reduzieren.

6.4 Abschließende Bemerkungen

Mit der vorgestellten Arbeit wurde ein kompaktes Framework für die parallele Verarbeitung von Postprocessing-Anfragen vorgestellt. Zudem wurden Mechanismen beschrieben, wie sich dieses System in interaktive Umgebungen integriert. Basierend auf diesem Parallelisierungs-Framework konnten sodann die wichtigsten für die Strömungssimulationsanalyse existierenden Extraktionsalgorithmen auf ihre Parallelisierbarkeit und ihr Laufzeitverhalten hin untersuchen werden. Obwohl einige Ansätze nur exemplarisch evaluiert wurden, liegt dennoch mit dieser Arbeit eine vollständige Bearbeitung der eingangs aufgeworfenen Fragestellungen vor.
Nicht abgedeckt wurden bewusst solche Visualisierungsansätze, die über die behandelten Extraktionsobjekte hinausgehen. Dazu gehört z. B. das *Volume-Rendering*. In zahlreichen Publikationen wurden Ansätze dargestellt, die für das Berechnen von einzelnen Volumenschichten (engl.: *Slices*) parallele Strategien vorsehen [LACR95]. Nach der Erstellung werden diese in die Graphikkarte geladen und gerendert. Es bietet sich nun an, auch hierfür unterstützend das Parallelisierungs-Framework z. B. für das regelmäßige Samplen von *Regions of Interest* (ROI) einzusetzen [LIN02]. Dasselbe gilt für die direkte Berechnung von Texturen für die Visualisierung. Beispielsweise lassen sich mit der Linienintegralfaltung (engl.: *Line Integral Convolution*, LIC) [CABR93, STAL95, INTE97, LEEU98] aufschlussreiche Strömungsbilder generieren, die effizient von moderner Graphik-Hardware gerendert werden können [HEID99]. Deren Erstellung ist jedoch zeit- und speicherintensiv und daher durchaus eine Aufgabe für Viracocha.
Auf der Grundlage des vorgestellten Parallelisierungs-Frameworks lassen sich nun weitergehende Konzepte realisieren. Da die Kopplung zwischen ViSTA FlowLib und Viracocha nur lose über TCP/IP erfolgt, könnte das Berechnungs-*Backend* zu einem dezentral installierten Postprocessing-Dienstleistungsserver weiterentwickelt werden, der Anfragen von beliebig vielen Visualisierungs-Clients bearbeiten kann. Da verwendete Datensätze dann ebenfalls dezentral verteilt wären, würden z. B. Sicherheitsaspekte verstärkt in den Vordergrund der Betrachtungen rücken. Bei Überlastungen oder bei großen Latenzen wäre es zudem wünschenswert, mehrere Server-Instanzen zur Verfügung zu haben, die sich nach entsprechenden Kriterien die Arbeit aufteilen. Dadurch wäre dann auch eine Ausfallsicherheit gegeben.
Solch ein erweitertes Konzept des verteilten Postprocessings würde wiederum den Ingenieurswissenschaften zugute kommen, die dann nicht nur für ihre aufwändigen CFD-Simulationen auf Hochleistungsrechner zurückgreifen könnten, sondern diese auch jederzeit verfügbar für die interaktive Exploration der CFD-Ergebnisse zur Seite gestellt bekämen.

Anhang

A. Verwendete CFD-Datensätze

Für die Entwicklung akkurater Postprocessing-Funktionalitäten wurden in dem vorgestellten Framework Viracocha vorwiegend Multi-Block-Datensätze verwendet, die an den Randpunkten vollständig aneinander passen (Konnektivitätstyp: *Matching*). Diese lassen sich auch innerhalb eines Zeitschrittes einfach auf mehrere Prozesse verteilen und dann parallel bearbeiten. Grundsätzlich sind auch andere Datenstrukturen in Teilblöcke zerlegbar. So wurden einige Experimente für rectilineare Gitter präsentiert. Unstrukturierte Datensätze lassen sich mithilfe von Graphpartitionierungsstrategien [SCHL02] unterteilen oder per *Out-of-Core*-Zugriffsmethoden verteilen. Die letztgenannten Aspekte werden jedoch nicht durch diese Arbeit abgedeckt.
Das vorrangige Ziel von Viracocha ist es sehr große instationäre Datensätze bearbeiten zu können. Dabei orientiert sich die Frage, was als groß anzusehen ist, vor allem an der Leistungsfähigkeit von Hochleistungsrechnern. Mit der stetigen Zunahme der Rechenleistung sind auch die Simulationergebnisse im Umfang gestiegen. Dieser Trend ist ungebrochen.
Am Rechen- und Kommunikationszentrum der RWTH Aachen stehen einige Datensätze zur Verfügung, auf denen für die Bewertung der in dieser Arbeit beschriebenen Verfahren zurückgegriffen werden konnte. Neben den hauptsächlich verwendeten instationären Multi-Block-Datensätzen, wurden auch einige rectilineare und unstrukturierte Gitter für die Evaluierung herangezogen. Tabelle 5 fasst die wesentlichen Informationen aller Datensätze zusammen.

	Kitchen	**Engine**	**Nase**	**Prop Fan**	**Shock**
Quelle	VTK	AIA	AIA	DLR	AIA
Gittertyp	Rectilinear	MB	MB	MB	Rectilinear
Zeitschritte	1	62	1	50	919
Blockanzahl	1	23	34	144	1
Gesamtgröße binär (ASCII)	(3,04 MB)	365 MB (1,12 GB)	(50 MB)	5.137 MB (19,5 GB)	73,4 GB
Knotenanzahl[16] pro Zeitschritt	11.424 28x24x17	40.822 - 201.434	443.329	2.533.356	1.950.399 99x99x199
Blockgröße	(3,04 MB)	15 – 1284 KB	(227 – 5296 KB)	256 – 800 KB	81 MB
Datengröße pro Zeitschritt	(3,04 MB)	1,92 – 9,37 MB	(50 MB)	102,74 MB	81 MB
# Vektorfelder	1	1	1	1	1
# Skalarfelder	19	4[17]	5	4	5

Tabelle 5: Verwendete CFD-Datensätze

Der am häufigsten verwendete Datensatz ist der *Engine*-Datensatz. Dieser beschreibt den Kaltgaseinströmvorgang in den Verbrennungsraum eines Vierventilmotors [ABDE98] und wurde vom Aerodynamischen Institut Aachen (AIA) zur Verfügung gestellt. Die Simulation umfasst Einströmung und Kompression, aber nicht den eigentlichen Verbrennungsprozess. Das physikalische Strömungsgebiet wurde in 23 Blöcke zerlegt (vgl. Abbildung 16) und weist 63 diskrete Zeitschritte auf. Um sich der Kolbenbewegung besser anpassen zu können,

[16] Für die Multi-Block-Datengitter wurden die an den Blockgrenzen gemeinsam genutzten Datenpunkte nur einmal gezählt.
[17] Eines der Skalarfelder enthält als Größe den Geschwindigkeitsbetrag. Dieses ist redundant, da die Geschwindigkeiten noch als Vektorfeld vorliegen.

bewegen sich die Blöcke während der Simulationszeit und ändern vereinzelt ihre Auflösung. Ab Zeitschritt 35 sind die Einlassventile geschlossen und die Kompressionsphase beginnt. Daher werden die Blöcke, die den Einlassbereich außerhalb der Verbrennungskammer vernetzt haben, nicht mehr verwendet. Die Blockanzahl verringert sich somit auf 17 Blöcke.
So besteht das gesamte MB-Gitter zum Zeitschritt 1 aus 114.998 Punkten und wächst bis zu 245.588 Punkten im Zeitschritt 34 an, bevor die Punktanzahl wieder abnimmt. Insgesamt erhält man somit eine binäre Datensatzgröße von 365 MByte. Verglichen mit anderen Datensätzen aktueller CFD-Simulationen, die bis zu einigen hundert Giga-Byte erreichen können [BRYS99, REIN01], fällt dieser recht klein aus. Trotzdem weist er alle Eigenschaften auf, um als Standard für die Framework- und Algorithmenentwicklung eingesetzt werden zu können.
Ebenfalls vom AIA ist der als Multi-Block erstellte Nasen-Datensatz, der für einen festen Zeitpunkt der Einatmung die Innenströmung einer menschlichen Nase beschreibt [GERN03]. Dieser Datensatz liegt zurzeit lediglich für einen Zeitschritt vor, jedoch wird aktuell an verbesserten Geometrien und verfeinerten Vernetzungen gearbeitet, die sodann auch instationär durchgerechnet werden sollen.
Der letzte der drei Multi-Block-Datensätze kommt vom Deutschen Zentrum für Luft- und Raumfahrt (DLR), Institut für Antriebstechnik, Köln. Hier handelt es sich um eine gegenläufig rotierende Turbine. Mit 50 Zeitschritten und 144 Einzelblöcken kommen über 5 GB Datenvolumen zusammen. Wegen der Rotationssymmetrie wurden jedoch lediglich 12 Blöcke pro Zeitschritt durchgerechnet. Um die Postprocessing-Methoden applikations- und datensatzunabhängig evaluieren zu können, wurden diese kopiert und rotiert, sodass die Turbine vollständig vernetzt vorliegt.
Neben diesen hauptsächlich verwendeten Datensätzen kam noch ein strukturierter Datensatz zum Einsatz, der von VTK mitgeliefert wird und Luftströmung in einer Küche (Datensatz von Dr. L. Besse, ETH Zürich, Schweiz) beschreibt (Abbildung 91). Dieser wurde vor allem zur Bestimmung von disjunkten Laufzeitanteilen von parallelen CFD-Postprocessing-Algorithmen verwendet.

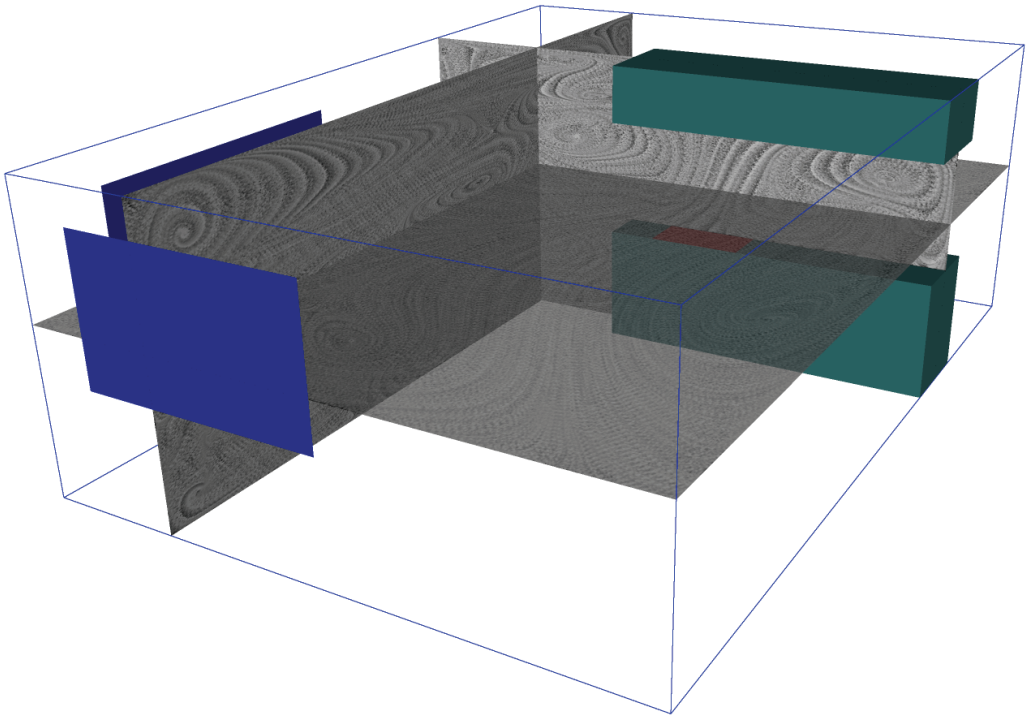

Abbildung 91: Texturbasierte Strömungsvisualisierung mithilfe der *Integrate-and-Draw*-Methode [RISQ98] im Küchendatensatz

Für die Evaluierung von kritischen Punkten und für einige speziellere Unterteilungsstrategien wurde letztendlich noch ein rectilinearer Datensatz eingesetzt, der im Rahmen einer Dissertation [THOM03] am AIA entstand und einen Verdichtungsstoß simuliert. Diese in Abbildung 92 visualisierte Strömung stellt im Rahmen dieser Arbeit den größten instationären Datensatz dar.

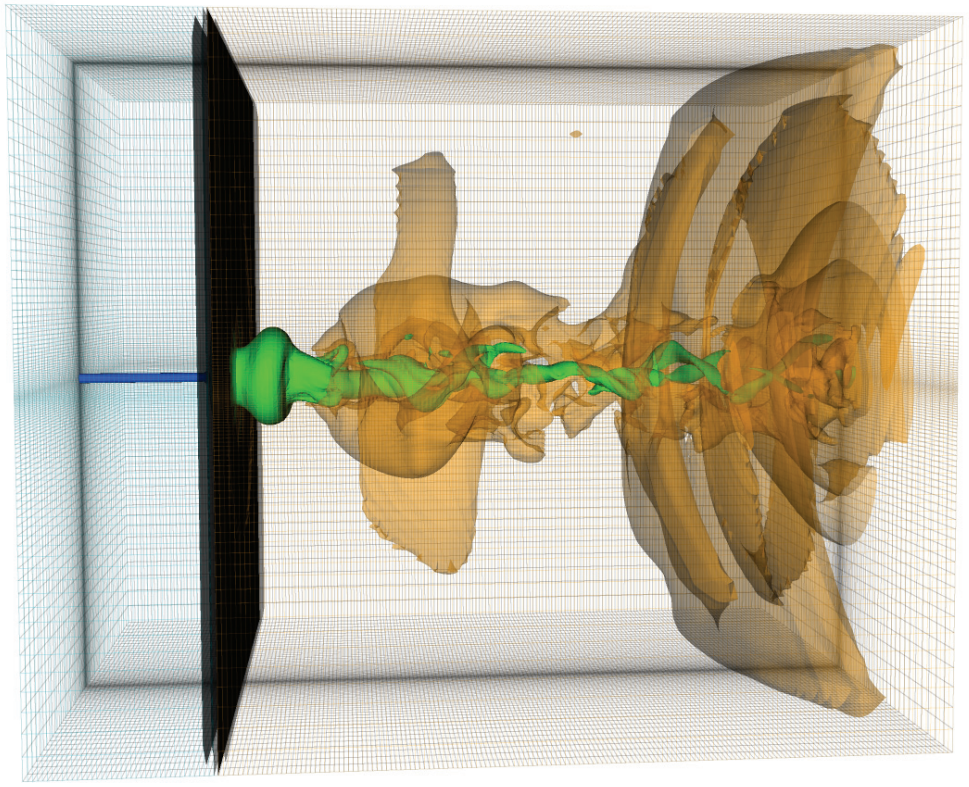

Abbildung 92: Verdichtungsstoß in einem rectilinearen Datensatz

B. Verwendete Rechnerarchitekturen

Im Rahmen dieser Arbeit kamen unterschiedliche Rechnersysteme zum Einsatz. Das größte System ist ein SunFire-Hochleistungsrechner von Sun Microsystems. Er besteht aus 16 SunFire-6800 und 4 SunFire-15K. Während jeder SunFire-6800-SMP-Knoten aus 24 UltraSPARC-III-Prozessoren besteht, werden davon in der SunFire-15K jeweils 72 Stück verwendet. Verbunden sind die einzelnen SMP-Knoten über einen Interconnect namens SunFire-Link, der jeweils 8 SunFire-6800 bzw. die 4 SunFire-15K zu Clustern verbindet. Auf einem Prozessorboard sind jeweils 4 Prozessoren zusammengefasst, und es kann bis zu 8 GByte Hauptspeicher aufnehmen. Insgesamt ist der Hochleistungsrechner somit mit 960 GByte Hauptspeicher und 672 CPUs ausgestattet. Als Prozessoren kommen die 64-Bit UltraSPARC-III Cu mit 900 MHz zum Einsatz, die 64 kByte primären und 8 MByte sekundären Cache-Speicher aufweisen. Insgesamt soll das System eine Peak-Performance von 1,2 Tera-Flops pro Sekunde liefern. Neben einem lokalen temporären Dateisystem haben alle Knoten über ein *Fast Storage Area Network* (SAN) direkten Zugriff auf ein gemeinsames NFS-Dateisystem. Als Betriebssystem wird Sun-Solaris 9 eingesetzt. Da sich der HPC-Cluster während der Erstellung dieser Arbeit noch im Aufbau befand, konnten sich die Leistungsdaten des Systems bei den verschiedenen Messungen durchaus deutlich unterscheiden.
Neben diesem Vertreter der *Shared-Memory*-Systeme kam ein Linux-Cluster der Firma Fujitsu-Siemens zum Einsatz. Der mit dem Namen hpcLine versehene Rechner besitzt 16 PC-Knoten, die zu einem *Distributed-Memory*-System verclustert sind. Jeder Einzel-PC besteht aus einem Dual-Prozessorboard, auf dem sich Intel-Pentium-II-Prozessoren mit 400 MHZ befanden, die später durch Pentium-III-Prozessoren mit je 800 MHz ersetzt wurden. Sowohl die PII- als auch die PIII-Prozessoren besitzen 512 KByte große Level-2-Caches und 32 KByte große primäre Caches. Die Größe des Hauptspeichers beträgt pro Knoten 512 MByte. Verbunden sind die Knoten über ein schnelles SCI-Netzwerk der Firma Scali Computer AS, das eine Transferrate von 80 MByte/s aufweist. Die Peak-Performance dieses Systems liegt somit bei 12,8 Giga-Flops.
Als Hochleistungsvisualisierungsrechner stand eine Onyx-2 Infinite Reality 2 der Firma SGI zur Verfügung. Vier MIPS-10000-Prozessoren, jeweils mit 195 MHz getaktet, haben in diesem *Shared-Memory*-System Zugriff auf insgesamt 2 GByte Hauptspeicher. Das Graphiksystem besteht aus einer Graphik-Pipe mit zwei Raster-Managern. Das Betriebssystem ist IRIX 6.5.
Neben diesem Graphikrechner kamen auch Windows-Rechner zum Einsatz, die primär für die Visualisierung eingesetzt wurden. Ein erstes System war eine Intergraph TDZ 2000 GT1. Als Dual-Prozessorsystem mit zwei Xeon-Prozessoren 500 MHz und 1 GByte Hauptspeicher ausgestattet, produziert die Intergraph mithilfe von zwei 3D Wildcat 4000 der Firma Intersense stereofähige Bilder. Unterdessen kommen jedoch wesentlich leistungsstärkere PCs zum Einsatz, die vor allem auf 3D-Graphikkarten der Firma NVidia aufsetzen. Ergänzt werden die Windows-Rechner noch durch zwei Workgroup-Server der Firma SUN. Diese SunFire-V880z sind mit jeweils zwei XVR-4000-Graphikkarten und vier UltraSPARC-III-Prozessoren mit je 900 MHz ausgestattet. Der Hauptspeicher beträgt jeweils 8 GByte.
Die Onyx und die beiden V880, obwohl primär als Visualisierungsrechner vorgesehen, sind mit ihren jeweils vier Prozessoren natürlich auch in der Lage das parallele CFD-Postprocessing vorzunehmen. Aber auch die Windows-PCs, verbunden über Gigabit-Ethernet, sind als loser Berechnungsverbund einsetzbar. Speziell hierfür ist daher auf allen Windows-Rechnern MPICH (siehe hierzu Anhang B.1) installiert.

B.1 MPI-Versionen

MPI läuft sowohl auf den im Rechen- und Kommunikationszentrum installierten *Shared-Memory*-Architekturen als auch auf *Distributed-Memory*-Systemen. Des Weiteren lassen sich Einzelrechner mit MPI zu einem Parallelisierungs-Cluster verbinden. Dann kommt allerdings meistens die allgemeine Variante MPICH, entwickelt vom Argonne National Laboratory, zum Einsatz, deren Transportprotokoll auf TCP/IP aufsetzt. Bleibt man auf einer homogenen Plattform, können dagegen optimierte Kommunikationswege gewählt werden. So hat Sun Microsystems für ihre SunFire-Hochleistungsrechner eine MPI-Version entwickelt, die Nachrichten und Daten über *Shared Memory* verschickt, was gegenüber einer Netzwerk-Variante um einige Faktoren schneller ist.
Aber auch die anderen in der Arbeit verwendeten Systeme sind mit MPI bzw. MPICH ausgestattet. Der Visualisierungsrechner Onyx-2 erreicht mit seinem nativen MPI rund 95 MByte/s, während die MPICH-Version 1.1.2 über TCP/IP wie üblich gerade einmal auf 10 MByte/s kommt. Wichtig für den Einsatz in heterogenen Systemen ist, dass MPICH mit XDR ausgestattet ist um eine automatische Datenkonvertierung zwischen Architekturen mit *Little-Endian*- und *Big-Endian*-Zahlenspeicherformaten, also z. B. zwischen Intel- und MIPS-Prozessor-Rechnern, durchführen zu können. Läuft die hpcLine, der verwendete Linux-Cluster, nicht im Verbund mit der Onyx, ist die über das schnelle SCI-Netzwerk kommunizierende MPI-Version ScaMPI der Version 1.10.2 einsetzbar und erreicht dann bis zu 80 MByte/s. Auf den Windows-Rechnern läuft ausschließlich MPICH.

Literaturverzeichnis

[ABDE98] Aschaf Abdelfattah, „Numerische Simulation von Strömungen in 2- und 4-Ventil-Motoren“, Dissertation, RWTH Aachen, 1998.

[AKEN02] Thomas Akenne-Möller, Eric Haines, „Real-Time Rendering”, 2. Auflage, AK Peters, 2002.

[AHRE00] James P. Ahrens, C. Charles Law, William J. Schroeder, Ken Martin, Michael Papka, „A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets”, Technischer Bericht #LAUR-00-1620, Los Alamos National Laboratory, 2000.

[ARLI96] Martin F. Arlitt, Carley L. Williamson, „Trace-Driven Simulation of Document Caching Strategies for Internet Web Servers“, Technischer Bericht, University of Saskatchewan, Kanada, September 1996.

[ASSE04] Ingo Assenmacher, Torsten Kuhlen, Tobias Lentz, Michael Vorländer, „Integrating Real-Time Acoustics into VR-Applications“, Proceedings, 10. Eurographics Symposium on Virtual Environments, VE04, Eurographics / ACM Siggraph, Grenoble, Frankreich, 2004.

[BAJA96] Chandrajit Bajaj, Valerio Pascucci, Daniel R. Schikore, „Fast Isocontouring for Improved Interactivity“, Proceeding, Volume Visualization Symposium, S. 39 – 46, Oktober 1996.

[BAJA99] Chandrajit Bajaj, Valerio Pascucci, Daniel R. Schikore, „Seed Sets and Search Structures for Optimal Isocontour Extraction“, Technischer Bericht, University of Texas, Austin, 1999.

[BAJA02] Chandrajit Bajaj, Ariel Shamir, Bong-Soo Sohn, „Progressive Tracking of Isosurfaces in Time-Varying Scalar Fields“, Technischer Bericht, University of Texas, Austin, 2002.

[BANC90] Gordon V. Bancroft, Fergus J. Merritt, Todd C. Plessel, Paul G. Kelaita, R. Kevin Kelaita, Al Globus, „FAST: A Multi-Processed Environment for Visualization of Computational Fluid Dynamics“, A. Kaufmann (Hrsg.), Proceedings, IEEE Visualization, San Francisco, CA, S. 14 – 27, 23. – 26. Oktober 1990.

[BARU98] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, Michael Wan, „The SDSC Storage Resource Broker”, Proceedings, CASCON ’98, Toronto, Kanada, 1998.

[BERG00] Marsha Berger, Michael Aftosmis, Gedas Adomavicius, „Parallel Multigrid on Cartesian Meshes with Complex Geometry“, Proceedings, 8. International Conference on Parallel CFD, Trondheim, Norwegen, 2000.

[BROD04] Ken W. Brodlie, David A. Duce, Julian R. Gallop, Jeremy P. R. B. Walton, Jason D. Wood, „Distributed and Collaborative Visualization”, Computer Graphics Forum, Jg. 23, Ausg. 2, S. 223 – 251, 2004.

[BRUE00] Christoph Brücker, „3-D Multiplanar PIV Measurements“, in M. Riethmuller (Hrsg.), „Particle Image Velocimetry and Associated Techniques“, VKI Lecture Series, von Karman Institute, Belgien, 2000.

[BRYS91] Steve Bryson, Creon Levit, „The Virtual Windtunnel – An Environment for the Exploration of Three-Dimensional Unsteady Flows“, Proceedings, IEEE Visualization, San Diego, CA, IEEE CS Press, S. 17 – 24, 1991.

[BRYS92] Steve Bryson, Michael J. Gerald-Yamasaki, „The Distributed Virtual Windtunnel“, Proceedings, ACM / IEEE Conference on Supercomputing, Minneapolis, MN, S. 275 – 284, 1992.

[BRYS96] Steve Bryson, „Virtual Reality in Scientific Visualization”, Communications of the ACM, Jg. 39, Ausg. 5, S. 62 – 71, Mai 1996.

[BRYS99] Steve Bryson, David Kenwright, Michael Cox, David Ellsworth, Robert Haimes, „Visual Exploring Gigabyte Data Sets in Real Time“, Communication of the ACM, Jg. 42, Ausg. 8, August 1999.

[BUNI85] Pieter G. Buning, Joseph L. Steger, „Graphics and Flow Visualization in Computational Fluid Dynamics“, Proceedings, 7. Computational Fluid Dynamics Conference, AIAA-1985-1507, Cincinnati, OH, 15. – 17. Juli 1985.

[CABR93] Brian Cabral, Leith Casey Leedom, „Imaging Vector Fields using Line Integral Convolution“, Proceedings, ACM Siggraph, ACM Press, New York, S. 263 – 270, 1993.

[CASH90] Jeff R. Cash, Alan H. Karp, „A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides”, ACM Transaction on Mathematical Software, Jg. 16, Ausg. 3, S. 201 – 222, 1990.

[CHAN99] Chialin Chang, Renato Ferreira, Alan Sussman, Joel Saltz, „Infrastructure for Building Parallel Database Systems for Multi-Dimensional Data“, Proceedings, 13. International Parallel Processing Symposium (IPPS) und 10. Symposium on Parallel and Distributed Processing (SPDP), San Juan, Puerto Rico, 12. – 16. April 1999.

[CHAN00] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, Jeff McDonald, „Parallel Programming in OpenMP“, Morgan Kaufmann, 2000.

[CHOU99] Alok Choudhary, Mahmut T. Kandemir, Harsha Nagesh, Jaechun No, Xiaohui Shen, Valerie Taylor, Sachin More, Rajeev Thakur, „Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems“, Proceedings, 8. IEEE Symposium on High-Performance Distributed Computing Conference, HPDC 99, Rendondo Beach, CA, IEEE Computer Society Press, 3. – 8. August 1999.

[CIGN96] Paolo Cignoni, Claudio Montani, Enrico Puppo, Roberto Scopigno, „Optimal Isosurface Extraction from Irregular Volume Data“, Proceedings, Volume Visualization Symposium, VVS ‘96, S. 31 – 38, 1996.

[CORT97] Tomi Cortes, „Cooperative Caching and Prefetching in Parallel / Distributed File Systems“, Dissertation, Unversitat Politècnica de Catalunya, Barcelona, Spanien, 1997.

[CRUZ92] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, John. C. Hart, „The CAVE – Audio-Visual Experience Automatic Virtual Environment“, Communications of the ACM, Jg. 35, Ausg. 6, S. 64 – 72, Juni 1992.

[CRUZ93] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, „Surround-Screen Projection-Based Virtual Reality: The Design and the Implementation of the CAVE“, Proceedings, ACM Siggraph, Anaheim, CA, ACM Press, S. 135 – 142, 1. – 6. August 1993.

[CURE93] Kenneth M. Curewitz, P. Krishnan, Jeffrey Scott Vitter, „Practical Prefetching via Data Compression“, Proceedings, Conference on Management of Data, ACM SIGMOD, S. 257 – 266, Mai 1993.

[DAGU98] Leonardo Dagum, Ramesh Menon, „OpenMP: An Industry-Standard API for Shared-Memory Programming“, IEEE Computational Science and Engineering, Jg. 5, Ausg. 1, S. 46 – 55, Januar – März 1998.

[DAHL94] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, David A. Patterson, „Cooperative Caching: Using Remote Client Memory to Improve File System Performance“, Proceedings, First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, CA, S. 267 – 280, 14. – 17. November 1994.

[DAM00] Andries van Dam, Andrew Forsberg, David H. Laidlaw, Joseph LaViola, Rosemary Michelle Simpson, „Immersive Virtual Reality for Scientific Visualization: A Progress Report“, IEEE Computer Graphics and Application, Jg. 20, Ausg. 6, S. 26 – 52, November / Dezember 2000.

[DEGA90] David Degani, Arnan Seginer, Yuval Levy, „Graphical Visualization of Vortical Flows by Means of Helicity“, AIAA Journal, Jg. 28, Ausg. 8, S. 1347 – 1352, 1990.

[DIJK65] Edsger W. Dijkstra. „Cooperating Sequential Processes“. Technical Report EWD-123, Technische Universität Eindhoven, 1965.

[DOUG99] Joseph Doug, Dirk Grunwald, „Prefetching Using Markov Predictors“, IEEE Transaction on Computers, Jg. 48, Ausg. 2, S. 121 – 133, 1999.

[FROE96] Kevin W. Froese, Richard B. Bunt, „The Effect of Client Caching on File Server Workloads“, Proceedings, 29. Hawaii International Conference on System Science, HICSS, Band 1: Software Technology and Architecture, S. 150 – 159, 1996.

[FRUE97] Thomas Frühauf, „Graphisch-Interaktive Strömungsvisualisierung", Dissertation, TH Darmstadt, Springer Verlag, 1997.

[GAMM96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, „Entwurfsmuster, Elemente wieder verwendbarer objektorientierter Software“, Addison Wesley Longman, 1996.

[GERL94] Carsten Andreas Gerlhof, Alfons Kemper: „A Multi-Threaded Architecture for Prefetching in Object Bases“, Matthias Jarke, Janis A. Bubenko Jr., Keith G. Jeffery (Hrsg.), 4. International Conference on Extending Database Technology, EDBT ’94, Cambridge, UK, Springer Verlag, S. 351 – 364, 28. – 31. März 1994.

[GERN00] Andreas Gerndt, Thomas van Reimersdahl, Torsten Kuhlen, Christian Bischof, „A Parallel Approach for VR-based Visualization of CFD Data with PC Cluster“, Proceedings, IMACS 20000, Lausanne, Schweiz, 2000.

[GERN02] Andreas Gerndt, Thomas van Reimersdahl, Torsten Kuhlen, Christian Bischof, „Large Scale CFD Data Handling with Off-The-Shelf PC-Clusters in a VR-based Rhinological Operation Planning System“, in: P. Wilders, A. Ecer, J. Periaux, N. Satofuka, P. Fox

(Hrsg.), Proceedings, Parallel Computational Fluid Dynamics - Practice and Theory, Elsevier Science B.V., S. 135 – 142, 2002.

[GERN03] Andreas Gerndt, Thomas van Reimersdahl, Torsten Kuhlen, Christian Bischof, Ingolf Hörschler, Matthias Meinke, Wolfgang Schröder, „Large-Scale CFD Data Handling in a VR-Based Otorhinolaryngological CAS-System using a Linux-Cluster“, in: Hamid R. Arabnia (Hrsg.): The Journal of Supercomputing - An International Journal of High-Performance Computer Design, Analysis, and Use, Kluwer Academic Publishers, Jg. 25, Ausg. 2, S. 143 – 154, 2003.

[GERN04a] Andreas Gerndt, Torsten Kuhlen, Thomas van Reimersdahl, Matthias Haack, Christian Bischof, „VR-based Interactive CFD Data Comparison of Flow Fields in a Human Nasal Cavity“, Proceedings, SPIE Medical Imaging 2004, San Diego, CA, USA, 2004.

[GERN04b] Andreas Gerndt, Stefan Lankes, Mark Asbach, Torsten Kuhlen, Thomas Bemmerl, Christian Bischof, „Conceptual Design and Implementation of a Pipeline-Based VR-System Parallelized by CORBA, and Comparison with Existing Approaches“, Proceedings, International Conference on Virtual Reality Continuum and its Applications in Industry, VRCAI 2004, Singapur, ACM Siggraph, S. 368 – 369, 2004.

[GERN04c] Andreas Gerndt, Bernd Hentschel, Marc Wolter, Torsten Kuhlen, Christian Bischof, „VIRACOCHA: An Efficient Parallelization Framework for Large-Scale CFD Post-Processing in Virtual Environments”, Proceedings, The International Conference for High Performance Computing and Communications, SC2004, Pittsburgh, PA, USA, 6. – 12. November 2004.

[GERN05] Andreas Gerndt, Marc Schirski, Torsten Kuhlen, Christian Bischof, „Parallel Calculation of Accurate Path Lines in Virtual Environments through Exploitation of Multi-Block CFD Data Set Topology”, George Gravvanis, Hamid Arabnia (Hrsg.), Journal of Mathematical Modelling and Algorithms, Jg. 4, Ausg. 1, Springer Science & Business Media B.V., März 2005.

[GHAR00] Morteza Gharib, Dana Daribi, „Digital Particle Image Velocimetry“, Alexander J. Smits, Tee Tai Lim (Hrsg.), „Flow Visualization – Techniques and Examples“, Imperial College Press, S. 123 – 147, 2000.

[GLOB91] Al Globus, Creon Levit, Thomas Lasinski, „A Tool for Visualization the Topology of Three-Dimensional Vector Fields“, Gregory M. Nielson, Larry Rosenblum (Hrsg.), Proceedings, IEEE Visualization, San Jose, CA, S. 33 – 40, 22. – 25. Oktober 1991.

[GRAY03] Kris Gray, „The Microsoft DirectX 9 Programmable Graphics Pipeline”, Microsoft Press, 2003.

[GRIM89] Anders Grimsrud, Gray Lorig, „Implementing a Distributed Process between Workstation and Supercomputer“, C. A. Brebbia, A. Peters (Hrsg.), Proceedings, 1. International Conference on Applications of Supercomputers in Engineering, Southampton, UK, Elsevier Science & Technology, Amsterdam, S. 133 – 144, 5. – 7. September 1989.

[GROP99] William Gropp, Ewing Lusk, Athony Skjellum, „Using MPI: Portable Parallel Programming with the Message Passing Interface“, 2. Ausgabe, MIT Press, 1999.

[GROS96] Roberto Grosso, Martin Schulz, Jan Kraheberger, Thomas Ertl, „Flow Visualization for Multiblock Multigrid Simulations”, Proceedings, Eurographics Workshop on Virtual Environments and Scientific Visualization, Monte Carlo, Monaco, Springer Verlag, S. 296 – 307, 1996.

[HAIM91] Robert Haimes, Michael B. Giles, „VISUAL3: Interactive Unsteady Unstructured 3D Visualization“, Proceedings, 29. AIAA Aerospace Science Meeting and Exhibit, AIAA-91-0794, Reno, NV, Januar 1991.

[HAIM94] Robert Haimes, „pV3: A Distributed System for Large-Scale Unsteady CFD Visualization“, Proceedings, 32. Aerospace Scienced Meeting and Exhibit, AIAA 94-0321, 1994

[HAMA97] Bernd Hamann, Robert J. Moorhead II, „A Survey of Grid Generation Methodologies and Scientific Visualization Efforts”, Gregory M. Nielson, Hans Hagen, Heinrich Müller (Hrsg.), „Scientific Visualization – Overviews, Methodologies, Techniques“, IEEE Computer Society Press, S. 59 – 101, 1997.

[HEID99] Wolfgang Heidrich, Rüdiger Westermann, Hans-Peter Seidel, Thomas Ertl, „Applications of Pixel Textures in Visualization and Realistic Image Synthesis“, ACM Symposium on Interactive 3D Graphics, I3D ’99, Atlanta, GA, S. 127 – 134, ACM Press, New York, 1999.

[HELM91] James L. Helman, Lambertus Hesselink, „Visualization Vector Field Topology in Fluid Flows“, IEEE Computer Graphics and Applications, Jg. 11, Ausg. 3, S. 36 – 46, 1991.

[HIBB90] Bill Hibbard, Dave Santek, „The VIS-5D System for Easy Interactive Visualization", Arie Kaufmann (Hrgs.), Proceedings, IEEE Visualization, San Francisco, CA, S. 28 – 35, 23. – 26. Oktober 1990.

[HOPP97] Hugues Hoppe, „View-Dependent Refinement of Progressive Meshes", Proceedings, ACM Siggraph, Los Angeles, CA, S. 189 – 198, 3. – 8. August 1997.

[INTE97] Victoria Interrante, Chester Grosch, „Strategies for Effectively Visualizing 3D Flow with Volume LIC", Roni Yagel, Hans Hagen (Hrsg.), Proceedings, IEEE Visualization, Phoenix, AZ, IEEE Computer Society Press, Los Alamos, CA, S. 421 – 425, 18. – 24. Oktober 1997.

[JEON95] Jinhee Jeong, Fazle Hussain, „On Identification of a Vortex", Journal of Fluid Mechanics, Jg. 285, S. 69 – 94, 1995.

[KENW96] David N. Kenwright, David A. Lane, „Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition", IEEE Transactions on Visualization and Computer Graphics, Jg. 2, Ausg. 2, S. 120 – 129, 1996.

[KNIS01] Joe Kniss, Patrick McCormick, Allen McPherson, James Ahrens, Jamie Painter, Alan Keahey, Charles Hansen, „Interactive Texture-Based Volume Rendering for Large Data Sets", IEEE Computer Graphics and Applications, Jg. 21, Ausg. 4, 2001.

[KOBB00] Leif Kobbelt, „$\sqrt{3}$ Subdivision", Proceedings, ACM Siggraph, ACM Press, S. 103 – 112, 2000.

[KONR01] R. Konrath, Wolfgang Schröder, Wolfram Limberg, „Three-Dimensional Flow Measurements within the Cylinder of a Motored Four-Valve Engine using Holographic Particle-Image Velocimetry", Proceedings, SAE International Fall Fuels & Lubricants Meeting, San Antonio, TX, September 2001.

[KOTZ01] David Kotz, „Disk-Directed I/O for MIMD Multiprocessors", Hai Jin, Tonni Cortes, Rajkumar Buyya (Hrsg.), „High Performance Mass Storage and Parallel I/O: Technologies and Applications", IEEE Computer Society Press und John Wiley & Sons, S. 513 – 535, 2001.

[KRAU01] Martin Kraus, Thomas Ertl, „Topology-Guided Downsampling", Proceedings, International Workshop on Volume Graphics, VG01, Stony Brook, NY, S. 139 – 147, 21. – 22. Juni 2001.

[KREY03] Oliver Kreylos, Wes Bethel, Terry J. Ligocki, Bernd Hamann, „Virtual-Reality-Based Interactive Exploration of Multiresolution Data", Gerald Farin, Bernd Hamann, Hans Hagen (Hrsg.), "Hierarchical and Geometrical Methods in Scientific Visualization", Springer Verlag, Heidelberg, S. 205 – 224, 2003.

[KROE96] Thomas M. Kroeger, Darrel D. E. Long, „Predicting Future File-System Actions From Prior Events", Proceedings, USENIX Annual Technical Conference, San Diego, California, USA, 22. – 26. Januar 1996.

[KROG93] Michael F. Krogh, Charles Hansen, „Visualization on Massively Parallel Computers using CM/AVS", AVS Users Conference, Orlando, FLorida, Mai 1993.

[KURC99] Tahsin Kurc, Chialin Chang, Renato Ferreira, Alan Sussmann, Joel Saltz, „Quering Very Large Multi-Dimensional Datasets in ADR", Proceedings, ACM/IEEE Conference on High Performance Networking and Computing, SC '99, Portland, OR, ACM Press, New York, 14. – 19. November 1999.

[KURC01] Tahsin Kurc, Ümit Çatalyürek, Chialin Chang, Alan Sussmann, Joel Saltz, „Visualization of Large Data Sets with the Active Data Repository", IEEE Computer Graphics and Applications, Jg. 21, Ausg. 4, S. 24 – 33, 2001.

[LABS02] Ulf Labsik, Kai Hormann, Martin Meister, Günther Greiner, „Hierarchical Iso-Surface Extraction", Journal of Computing and Information Science in Engineering, Jg. 2, Ausg. 4, S. 323 – 329, Dezember 2002.

[LACR95] Philippe G. Lacroute, „Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation", Dissertation, Departments of Electrical Engineering and Computer Science, Stanford University, CA, 1995.

[LANE94] David A. Lane, „UFAT: A Particle Tracer for Time-Dependent Flow Fields", Dan Bergeron, Arie Kaufmann (Hrsg.), Proceedings, IEEE Visualization, Washington D.C., IEEE Computer Society Press, S. 257 – 264, 17. – 21. Oktober 1994.

[LANE97] David A. Lane, „Scientific Visualization of Large-Scale Unsteady Fluid Flows", Gregory M. Nielson, Hans Hagen, Heinrich Müller (Hrsg.), „Scientific Visualization – Overviews, Methodologies, Techniques", IEEE Computer Society Press, S. 125–145, 1997.

[LAW99] C. Charles Law, Kenneth M. Martin, William J. Schroeder, Joshua E. Temkin, „A Multi-Threaded Streaming Pipeline Architecture for Large Structured Data Sets", Proceedings, IEEE Visualization, Washington D.C., IEEE Computer Society Press, 1999.

[LEEU98] Wim de Leeuw, Robert van Liere, „Comparing LIC and Spot Noise“, Proceedings, IEEE Visualization, Research Triangle Park, NC, S. 359 – 365, IEEE Computer Society Press, Los Alamos, CA, 1998.

[LEEU99] Wim de Leeuw, Robert van Liere, „Visualization of Global Flow Structures Using Multiple Levels of Topology”, E. Gröller, H. Löffelmann, W. Ribarski (Hrsg.), Proceedings, Joint Eurographics and IEEE TCVG Symposium on Visualization, VisSym99, Wien, Österreich, S. 45 – 52, 26. – 28. Mai 1999.

[LEEU00] Wim de Leeuw, Robert van Liere, „Multi-Level Topology for Flow Visualization“, Computer & Graphics, Jg. 24, Ausg. 3, S. 325 – 331, Juni 2000.

[LIN02] Ching-Feng Lin, Don-Lin Yang, Yeh-Ching Chung, „Parallel Shear-Warp Factorization Volume Rendering Using Efficient 1-D and 2-D Partitioning Schemes for Distributed Memory Multicomputers“, The Journal of Supercomputing, Jg, 22, Ausg. 3, Kluwer Academic Publiher B. V., S. 277 – 302, Juli 2002.

[LOMA98] Harvard Lomax, Thomas H. Pulliam, David W. Zingg, „Fundamentals of Computational Fluid Dynamics”, Online-Veröffentlichung: 3. September 1998; Buch-Veröffentlichung: Springer Verlag, Berlin, 2001.

[LORE87] William E. Lorensen, Harvey E. Cline, „Marching Cubes: A High Resolution 3D Surface Construction Algorithm“, Proceedings, ACM Siggraph, ACM Press, S. 163 – 169, 1987.

[LUEB97] David Luebke, Carl Erikson, „View-Dependent Simplification of Arbitrary Polygonal Environments“, Proceedings, ACM Siggraph, Los Angeles, CA, August 1997.

[LUEB01] David Luebke, Amitabh Varshney, Jon Cohen, Martin Reddy, Ben Watson, „Advanced Issues in Level of Detail”, Course Notes, Kurs 45, ACM Siggraph, 2001.

[MOON01] Bongki Moon, Hosagrahar. V. Jagadish, Christos Faloutsos, Joel H. Saltz, „Analyses of the Clustering Properties of the Hilbert Space-Filling Curve“, IEEE Transaction on Knowledge and Data Engineering, Jg. 13, Ausg. 1, S. 124 – 141, 2001.

[NEID93] Jackie Neider, Tom Davis, Mason Woo, „OpenGL Programming Guide – The Official Guide to Learning OpenGL, Release 1“, OpenGL Architecture Review Board, Addison Wesley, 1993.

[NICH96] Bradford Nichols, Dick Buttlar, Jacqueline P. Farrell, „Pthread Programming – A POSIX Standard for Better Multiprocessing“, O'Reilly & Associates, 1996.

[NIEL91] Gregory M. Nielson, Bernd Hamann, „The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes“, Proceedings, IEEE Visualization, San Diego, CA, IEEE Computer Society Press, Los Alamitos, CA, S. 83 – 91, 22. – 25. Oktober 1991.

[NIEL03] Gregory M. Nielson, „On Marching Cubes“, IEEE Transactions on Visualization and Computer Graphics, Jg. 9, Ausg. 3, S. 283 – 297, Juli / September 2003.

[NO03] Jaechun No, Rajeev Thakur, Alok Choudhary, „High-Performance Scientific Data Management System”, Journal for Parallel and Distributed Computing, Jg. 63, Ausg. 4, S. 434 – 447, Elsevier Inc., 2003.

[OLBR99] Stephan Olbrich, Helmut Pralle, „Virtual Reality Movies – Real-Time Streaming of 3D Objects“, Proceedings, TERENA-NORDUnet Networking Conference, TNNC 1999, Lund, Schweden, 1999.

[OLBR01] Stephan Olbrich, Helmut Pralle, „Using Streaming and Parallelization Techniques for 3D Visualization in a High-Performance Computing and Networking Environment”, Proceedings, International Conference on High Performance Computing and Networking Europe, HPCN Europe 2001, Amsterdam, Niederlande, 2001.

[OMG02] „OMG Technical Document formal/02-06-01“, „The Common Object Request Broker -- Architecture and Specification“, Version 3.0, 2002.

[PACH96] Peter Pacheco, „Parallel Programming with MPI“, Morgan Kaufmann Publishers, 1996.

[PAGE93] Hans-Georg Pagendarm, „HIGHEND, A Visualization System for 3D Data with Special Support for Postprocessing of Fluid Dynamics Data“, M. Grave, Y. LeLous, W. Hewitt (Hrsg.), Visualization in Scientific Computing, Springer Verlag, Heidelberg, 1993.

[PAJA01] Renato Pajarola, „FastMesh: Efficient View-dependent Meshing“, Proceedings, Pacific Graphics Conference, IEEE Computer Society Press, S. 22 – 30, 2001.

[PAJA04] Renato Pajarola, Christopher DeCoro, “Efficient Implementation of Real-Time View-Dependent Multiresolution Meshing , IEEE Transaction on Visualization and Computer Graphics, Jg. 10, Ausg. 3, May / Juni 2004.

[PARK99] Steven Gregory Parker, „The SCIRun Problem Solving Environment and Computational Steering Software“, Dissertation, Department of Computer Science. University Utha, 1999.

[PASC00] Valerio Pascucci, Chandrajit L. Bajaj, „Time Critical Isosurface Refinement and Smoothing“, Proceedings, IEEE Symposium on Volume Visualization, VVS ’00, Salt Lake City, Utah, , S. 33 – 42, 9. – 10. Oktober 2000.

[PRES02] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, „Numerical Recipes in C++ – The Art of Scientific Computing“, 2. Auflage, Cambridge University Press, 2002.

[QUER91] Valerie Quercia, Tim O’Reilly, „X Windows System User’s Guide – OSF/Motif Edition“, Band 3, O’Rheilley & Associates, 1991.

[RANT98a] Dirk Rantzau, Karin Frank, Ulrich Lang, Daniela Rainer, Uwe Wössner, „COVISE in the CUBE: An Environment for Analyzing Large and Complex Simulation Data“, Proceedings, 2. Workshop on Immersive Projection Technology, IPT98, Ames, IA, 1998.

[RANT98b] Dirk Rantzau, Ulrich Lang, „A Scalable Virtual Environment for Large Scale Scientific Data Analysis“, Future Generation Computer Systems, Jg. 14, Ausg. 3 – 4, Elsevier Science B. V., S. 215 – 222, 1998.

[REIM00] Thomas van Reimersdahl, Torsten Kuhlen, Andreas Gerndt, Jörg Henrichs, Christian Bischof, „ViSTA: A Multimodal, Platform-Independent VR-Toolkit based on WTK, VTK, and MPI“, 4. International Immersive Projection Technology Workshop, IPT 2000, Ames, IA, 2000.

[REIM01] Thomas van Reimersdahl, Ingolf Hörschler, Andreas Gerndt, Torsten Kuhlen, Matthias Meinke, Georg Schlöndorff, Wolfgang Schröder, Christian Bischof, „Airflow Simulation inside a Model of the Human Nasal Cavity in a Virtual Reality-based Rhinological Operation Planning System“, Proceedings, Computer-Assisted Radiology and Surgery, CARS 2001, 15. International Congress and Exhibition, Berlin, Deutschland, S. 85 – 90, 2001.

[REIN01] Koob Frederik Jan Reinders, „Feature-Based Visualization of Time-Dependent Data“, Dissertation, Universität Delft, Niederlande, 2001.

[RESC99] Michael M. Resch, Dirk Rantzau, Robert Stoy, „Metacomputing Experience in a Transatlantic Wide Area Application Test-Bed“, Future Generation Computer Systems, Jg. 15, Ausg. 5 – 6, Elsevier Science B. V., S. 807 – 816, 1999.

[RISQ98] Carlos Pérez Risquet, “Visualizing 2D Flows: Integrate and Draw”, Dirk Bartz (Hrsg.), Proceedings, 9. Eurographics Workshop on Visualization in Scientific Computing, Springer Verlag, Wien, S. 57 – 67, 1998.

[ROBI90] John T. Robinson, Murthy V. Devarakonda, „Data Cache Management Using Frequency Based Replacement“, Proceedings, Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS, S. 134 – 142, Mai 1990.

[ROGE92] David Rogelberg, Joanne Clapp Fullagar (Hrsg.), „OpenGL Reference Manual – The Official Reference Document for OpenGL, Release 1“, OpenGL Architecture Review Board, Addison Wesley, 1992.

[ROSA93] Juan Miguel del Rosario, Rajesh Bordawekar, Alok Choudhary, „Improved Parallel I/O via a Two-Phase Run-Time Access Strategy“, Proceedings, Workshop on I/O in Parallel Computer Systems at IPPS93, S. 56 – 70, 1993.

[ROTH00] Martin Roth, „Automatic Extraction of Vortex Core Lines and Other Line-Type Features for Scientific Visualization“, Dissertation, ETH No. 13673, ETH Zürich, Institute of Scientific Computing, Computer Graphics Group, Hartung-Gorre Verlag, Juni 2000.

[SADA94] I. Ari Sadarjoen, Theo van Walsum, Andrea J. S. Hin, Frits H. Post, „Particle Tracing Algorithms for 3D Curvilinear Grids“, Proceedings, 5. Eurographics Workshop on Visualization in Scientific Computing, Rostock, 1994.

[SADA97] I. Ari Sadarjoen, Theo van Walsum, Andreas J. S. Hin, Frits H. Post, „Particle Tracing Algorithms for 3D Curvilinear Grids“, in: Gregory M. Nielson, Hans Hagen, Heinrich Müller (Hrsg.), „Scientific Visualization – Overview, Methodologies, Techniques“, IEEE Computer Society Press, S. 311 – 335, 1997.

[SCHI03] Marc Schirski, Andreas Gerndt, Thomas van Reimersdahl, Torsten Kuhlen, Philipp Adomeit, Oliver Lang, Stefan Pischinger, Christian Bischof, „ViSTA FlowLib – A Framework for Interactive Visualization and Exploration of Unsteady Flows in Virtual Environments“, Proceedings, 7. International Immersive Projection Technologies Workshop / 9. Eurographics Workshop on Virtual Environments, ACM Siggraph, Zürich, Schweiz, S. 77 – 85, Mai 2003.

[SCHI05] Marc Schirski, Torsten Kuhlen, Martin Hopp, Philipp Adomeit, Stefan Pischinger, Christian Bischof, „Virtual Tubelets - Efficiently Visualizing Large Amounts of Particle Trajectories“,Computers & Graphics, Jg. 29, Ausg. 1, S. 17 – 27, Februar 2005.

[SCHL02] Kirk Schloegel, George Karypis, Vipin Kumar, „Graph Partitioning for High-Performance Scientific Simulations“, Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, Andy White (Hrsg.), „The Sourcebook of Parallel Computing“, Kapitel 18, Morgan Kaufmann, 2002.

[SCHM02] Frank Schmuck, Roger Haskin, „GPFS: A Shared-Disk File System for Large Computing Clusters“, Proceedings, Conference on File and Storage Technologies (FAST'02), Monterey, CA, USENIX, Berkeley, CA, S. 231 – 244, 28. – 30. Januar 2002.

[SCHR98] William J. Schroeder, Ken Martin, Bill Lorensen, „The Visualization Toolkit – An Object-Oriented Approach to 3D Graphics”, 2. Auflage, Prentice Hall, NJ, 1998.

[SCHR00a] William J. Schroeder, Lisa S. Avila, William Hoffmann, „Visualization with VTK: A Tutorial“, IEEE Computer Graphics and Application, Jg. 20, Ausg. 5, S. 20 – 27, 2000.

[SCHR00b] Wolfgang Schröder, „Fluidmechanik“, Aachener Beiträge zur Strömungsmechanik (ABS), Band 3, 2. Auflage, Wissenschaftsverlag Mainz, Aachen, 2000.

[SCHR01] William J. Schroeder (Hrsg.), „The VTK User's Guide”, Kitware Inc., 2001.

[SCHU00] Heidrun Schumann, Wolfgang Müller, „Visualisierung – Grundlagen und allgemeine Methoden“, Springer Verlag, 2000.

[SEAM95] Kent E. Seamons, Ying Chen, Phyllis Jones, John Jozwiak, Marianne Winslett, „Server-Directed Collective I/O in Panda“, Proceedings, ACM / IEEE Conference on Supercomputing, San Diego, CA, IEEE Computer Society Press, 1995.

[SIEG00] Jon Siegel, „CORBA 3 – Fundamentals and Programming“, 2. Ausgabe, John Wiley & Sons, 2000.

[SILV02] Cláudio T. Silva, Yi-Jen Chiang, Jihad El-Sana, Peter Lindstrom, „Out-Of-Core Algorithm for Scientific Visualization and Computer Graphics“, Course Notes, IEEE Visualization, Boston, MA, Oktober 2002.

[SMIT89] Merrit H. Smith, William R. van Dalsem, F. Carroll Dougherty, Pieter G. Buning, „Analyse and Visualization of Complex Unsteady Three-Dimensional Flows“, Proceedings, 27. AIAA Aerospace Sciences Meeting, AIAA-1989-0139, Reno, NV, 9. – 12. Januar 1989.

[SNIR96] Marc Snir, Steve Otto, Steven Huss-Ledermann, Jack Dongarra, David W. Walker, „MPI: The Complete Reference“, MIT Press, Januar 1996.

[SPUR96] Joseph H. Spurk, „Strömungslehre – Einführung in die Theorie der Strömungen“, 4. Auflage, Springer Verlag, 1996.

[STAL95] Detlev Stalling, Hans-Christian Hege, „Fast and Resolution Independent Line Integral Convolution“, Proceedings, ACM Siggraph, ACM Press, New York, S. 249 – 256, 1995.

[STAL04] Detlev Stalling, Hans-Christian Hege, Malte Westerhoff, „Amira – A Highly Interactive System for Visual Data Analysis“, in: C. R. Johnson, C. D. Hansen (Hrsg.), „Visualization Handbook“, Kapitel 38, Academic Press, S. 749 – 767, 2004.

[STEF01] Roland Andreas Steffan, „Multimodale Interaktionstechniken für die Simulation von Montagevorgängen in virtuellen Umgebungen“, Dissertation, Fakultät für Elektrotechnik und Informationstechnik, RWTH Aachen, Mensch & Buch Verlag, Berlin, 2001.

[STOL95a] Eric J. Stollnitz, Tony D. DeRose, David H. Salesin, „Wavelets for Computer Graphics: A Primer Part 1“, IEEE Computer Graphics and Applications, Jg. 15, Ausg. 3, S. 76 – 84, Mai/Juni 1995.

[STOL95b] Eric J. Stollnitz, Tony D. DeRose, David H. Salesin, „Wavelets for Computer Graphics: A Primer Part 2“, IEEE Computer Graphics and Applications, Jg. 15, Ausg. 4, S. 75 – 85, Juli/August 1995.

[TANE02] Andrew S. Tanenbaum, Maarten van Steen, „Distributed Systems – Principles and Paradigms“, Prentice Hall, 2002.

[THAK99a] Rajeev Thakur, William Gropp, Ewing Lusk, „Data Sieving and Collective I/O in ROMIO“, Proceedings, 7. Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, S. 182 – 189, 1999.

[THAK99b] Rajeev Thakur, William Gropp, Ewing Lusk, „On Implementing MPI-IO Portably and with High Performance“, Proceedings, 6. Workshop on Input/Output in Parallel and Distributed System, S. 23 – 32, 1999.

[THOM03] Oliver Thomer, „Numerische Untersuchung der Wechselwirkung von Längswirbeln mit senkrechten und schrägen Verdichtungsstößen”, Dissertation, Fakultät für Maschinenbau, RWTH Aachen, 2003.

[TRIC01] Xavier Tricoche, Gerik Scheuermann, Hans Hagen, „Continuous Topology Simplification of Planar Vector Fields“, Proceedings, IEEE Visualization, S. 159 – 166, 2001.

[UPSO89] Craig Upson, Thomas Faulhaber Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, Andries van Dam, „The Application Visualization System: A Computational Environment for Scientific Visualization“, IEEE Computer Graphics and Applications, Jg. 9, Ausg. 4, 1989.

[VITT01] Jeffrey Scott Vitter, „External Memory Algorithms and Data Structures: Dealing with Massive Data“, ACM Computing Survey, Jg. 33, Ausg. 2, S. 209 – 271, Juni 2001.

[VOLL91] Heinrich Vollmers, „The Recovering of Flow Features from Large Numerical Data Bases“, VKI Lecture Series, Computer Graphics and Flow Visualization in Computational Fluid Dynamics, von Karman Institute for Fluid Dynamics, Brüssel, Belgien, 1991.

[WELL96] Friedrich Weller, „Numerische Mathematik für Ingenieure und Naturwissenschaftler – Eine Einführung für Studium und Praxis“, Vieweg, 1996.

[WEST97] Rüdiger Westermann, Thomas Ertl, „A Multiscale Approach to Integrated Volume Segmentation and Rendering“, Computer Graphics Forum, Jg. 16, Ausg. 3, (Proceedings Eurographics), S. 117 – 128, 1997.

[WEST01] Rüdiger Westermann, Christopher Johnson, Thomas Ertl, „Topology-Preserving Smoothing of Vector Fields“, IEEE Transactions on Visualization on Computer Graphics, Jg. 7, Ausg. 3, S. 222 – 229, 2001.

[WIEL96] Steve van der Wiel, David J. Lilja, „A Survey of Data Prefetching Techniques“, Technischer Bericht, HPPC-96-05, University of Minnesota, High Performance Parallel Computing Research Group, 1996.

[WIER01] Andreas Wierse, „Eine objekt-orientierte Datenverwaltung für eine verteilte Visualisierungsumgebung“, Dissertation, Fakultät für Energietechnik, Universität Stuttgart, 2001.

[WILH92] Jane Wilhelms, Allen van Gelder, „Octrees for Faster Isosurface Generation“, ACM Transactions on Graphics, Jg. 11, Ausg. 3, S. 201 – 227, Juli 1992.

[WILL93] Darryl L. Willick, Derek L. Eager, Richard B. Bunt, „Disk Cache Replacement Policies for Network Fileservers“, Proceedings, International Conference on Distribute Computing Systems, S. 2 – 11, 1993.

[ZORI00] Denis Zorin, Peter Schröder, Tony DeRose, Leif Kobbelt, Adi Levin, Wim Sweldens, „Subdivision for Modeling and Animation“, Course Notes, Kurs 23, ACM Siggraph, ACM Press, 2000.

Index

Lebenslauf des Verfassers

Daten zur Person

Name	Andreas Gerndt
Geburtsdatum	11. Dezember 1965
Geburtsort	Berlin

Werdegang

1972 bis 1973	Goetheschule, Mühlheim
1973 bis 1976	Wilhelm-Busch-Schule, Grundschule in Jügesheim
1976 bis 1983	Georg-Büchner-Schule, Gesamtschule in Jügesheim, Gymnasialzweig
1983 bis 1986	Gewerblich-Technische Schulen, Offenbach Schwerpunkt: Elektrotechnik
13. Juni 1986	Erlangung der allgemeinen Hochschulreife
01.10.1986 bis 31.01.1988	Zivildienst beim Deutschen Roten Kreuz, Rettungsdienst, Kreisverband Offenbach
01.02.1988 bis 30.09.1988	Rettungssanitäter beim DRK-Offenbach
03. September 1990	Anerkennung zum Rettungsassistenten
18.10.1988 bis 31.09.1993	Studium der Informatik, Technische Hochschule Darmstadt
12. Oktober 1993	Datum des Diplomzeugnisses
bis 31.04.1994	Wissenschaftliche Hilfskraft, Fraunhofer Institut für Graphische Datenverarbeitung, Darmstadt
19.08.1994 bis 06.12.1994	Angestellter bei der Firma Videocomp
01.03.1995 bis 31.09.1997	Angestellter bei der Firma Cegelec AEG Anlagen- und Automatisierungstechnik GmbH
01.10.1997 bis 30.10.1999	Angestellter bei der Firma DIAL GmbH
Seit 01.11.1999	Wissenschaftlicher Angestellter, Abteilung Virtuelle Realität, Rechen- und Kommunikationszentrum der RWTH Aachen
20. Januar 2006	Mündliche Doktorprüfung