Universität Hamburg

Bachelorarbeit
im Studiengang Informatik

Analyse und Optimierung von nicht-zusammenhängende Ein-/Ausgabe in MPI

Autor:
Daniel Schmidtke
Matr.-Nr.: 6250282

Betreuer:
Dr. Julian Kunkel
Michaela Zimmer

Wissenschaftliches Rechnen
am
Fachbereich Informatik
in der
MIN-Faktultät

7. April 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Problembeschreibung

Ein-/ Ausgabeprozesse haben bei der Datenverarbeitung eine große Bedeutung. Je koplexer die Modelle sind, desto mehr Rechenleistung und Zeit wird für ihre Berechnung benötigt. Daher wird hierfür das Hochleistungsrechnen eingesetzt. Dieses ist ein Bereich des computergestützten Rechnens. Er umfasst alle Rechenarbeiten, deren Bearbeitung einer hohen Rechenleistung oder Speicherkapazität bedarf. [1] Das Hochleistungsrechnen (high-performance computing – HPC) findet heutzutage nicht mehr nur auf einer Maschine statt. Es werden Cluster eingesetzt, um die Arbeitsaufträge zu bewältigen. Dies kann auch an der TOP500 [1] Liste gesehen werden. Hier gibt es fast ausschließlich Cluster-Systeme. Die Liste wird derzeit vom Tianhe-2 vom National Super Computer Center in Guangzhou mit 3,1 Mio Kernen in 16.000 Knoten angeführt. Jeder Knoten des Clusters ist über ein Netzwerk mit jedem anderen Knoten verbunden.
In den vergangen Jahren ist die Rechenkapazität gemäß dem Moor'schen Gesetz exponentiell angewachsen. Dabei sind aber die Übertragungsraten der Netzwerke und die Geschwindigkeiten der Speichersysteme nicht adäquat so gesteigert worden. Dies führt zwangsläufig zu einem Ungleichgewicht - einem Flaschenhals im Gesamtsystem.
Moderne Hochleistungsrechner verfügen über ein paralleles Dateisystem um Daten zu speichern. Dieses besteht wiederum selbst aus einem Cluster von Speicherknoten. Es kommen nicht nur viele Speicherknoten zum Einsatz, sondern zusätzlich in jedem Speicherknoten eine Vielzahl an Festplatten und in neuen System auch SSDs, um dem exponentiellen Wachstum der Rechenkapazität Rechnung zu tragen und einen hohen Datendurchsatz zu ermöglichen.
Ein hoher Datendurchsatz allein ist jedoch nicht alles. Ein ineffizienter Datenzugriff auf

[1] http://www.top500.org/ - Ranking der aktuell Leistungsfähigsten Computersysteme

ein sehr effizientes Dateisystem kann auch zu einem schlechten Durchsatz führen. Jeder Speicherzugriff auf das Dateisystem führt zwangsläufig dazu, dass die CPU auf neue Daten warten muss oder andererseits warten muss, bis bereits bearbeitete Daten gesichert werden, um darauf weiter zu arbeiten. Dieses Verhalten führt zu einer geringen Auslastung des Gesamtsystems und somit zu längeren Bearbeitungszeiten der Programme. An dieser Stelle setzt das Projekt Scalable I/O for Extreme Performance (SIOX) an. Es zeichnet die Zugriffe auf das Dateisystem auf und versucht vorhandene Optimierungsmöglichkeiten intelligent zu nutzen, in Kapitel 2 wird diese näher beschrieben.

1.2 Ziele der Arbeit

Das Ziel dieser Arbeit ist es, das Potential von Datasieving zu evaluieren und in Optimierungen nutzbar zu machen. Dazu werden die folgenden Ziele definiert.

1. **Systematische Analyse der erzielbaren Leistung**
    - Zu diesem Zweck wird für verschiedene Zugriffsmuster untersucht, inwiefern Datasieving die Leistungsfähigkeit beeinflusst
2. **Transparente Optimierung**
    - Ein Plugin für SIOX soll entwickelt werden, welches die besten Parameter, die in der systematischen Analyse identifiziert werden können, für eine Anwendung setzt. Hierzu gestattet SIOX es Optimierungsparameter transparent für den Nutzer zu setzten, d.h. ohne Veränderung des Qellcodes
3. **Kontextsensitive Optimierung**
    - Es soll untersucht werden, inwieweit zustandsbehaftete Parameter die Leistung steigern können
    - Ein weiteres Plugin soll entwickelt werden, das die Konzepte aus 2. dahingehend erweitert , dass nun in Abhängigkeit vom Kontext in einer Anwendung die Optimierungen gesetzt werden.

Das Ziel dieser Arbeit ist das Design und die zugehörige Implementation mehrerer PlugIns für das Scalable I/O for Extreme Performance (SIOX) Projekt. Eines dieser PlugIns soll außergewöhnliches Verhalten erkennen und melden. Ein weiteres soll einen aufgezeichneten Programmlauf abspielen und Optimierungsmöglichkeiten geben.
Beide PlugIns sollen für vorhandene Module in SIOX geschrieben werden und setzten

somit auf eine gegebene PlugIn-API auf.
Im letzten Teil der Arbeit soll es eine Auswertung geben, mit der der Leistungsgewinn durch die Nutzung von SIOX und der neu entwickelten Module aufgezeigt und veranschaulicht wird.

1.3 Gliederung

Im ersten Kapitel, der Einleitung, gebe ich einen Überblick über diese Bachelorarbeit.
Das zweite Kapitel beschreibt das SIOX-Projekt näher und wird die Grundlagen von MPI erläutern.
Im dritten Kapitel wird die systematische Analyse der Nutzdatenrate mit und ohne Datasieving durchgeführt
Das vierte Kapitel wird das Design der Module erläutern.
Das fünfte Kapitel wird die wichtigsten Fakten über die Implementation erklären.
Das sechste Kapitel fasst diese Arbeit zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

Im folgenden Kapitel werde ich den Stand der Technik und SIOX näher beschreiben sowie eine Einführung in MPI geben.

Kapitel 2

Stand der Technik

Dieses Kapitel wird das Scalable I/O for Extreme Performance (SIOX) Projekt vorstellen und die benötigten Grundlagen von MPI erklären. Dazu wird bei SIOX auf die vorhandenen Module und deren zusammenwirken eingegangen. Im Abschnitt für MPI wird die Fileview näher beleuchtet, die essentiell für das später zu entwickelnde Plugin ist.

2.1 Das Scalable I/O for Extreme Performance (SIOX) Projekt

Bei SIOX handelt es sich um ein Projekt, dass eine Umgebung zur Verfügung stellt, die alle Informationen über Eingabe/Ausgabe Zugriffe auf allen Schichten sammelt und automatisch auswertet. SIOX wird stets mit mindestens einer instrumentierten Softwareschicht geladen. Bisher implementiert sind folgende Softwareschichten MPI, POSIX, hdf5 und netcdf4. SIOX sammelt E/A Aktivitäten an allen geladenen Softwareschichten. Zusätzlich werden alle relevanten Hardware Informationen und Metriken über den Knoten gesammelt, wie zum Beispiel CPU Auslastung oder auch Netzwerkauslastung.

In SIOX werden online Messung und offline lernen vereint. Das sammeln der Daten geschieht im online Teil von SIOX. Alle Daten werden dabei in einer Datenbank gespeichert um diese später zu analysieren. Das Auswerten und Lernen der gesammelten Daten geschieht offline. Während der Lernphasen wird eine Wissensbasis gepflegt in der Optimierungen gesammelt werden. Jede dieser Optimierungen enthält Informationen über die zu nutzenden Parameter sowie über die Situation in der diese Parameter optimal sind. Während der online Phasen wird auf diese Wissensbasis zugegriffen um gespeicherte Situationen zu erkennen und optimierte Parameter anzuwenden. Mit jedem

Lauf von SIOX wird die Datenbank der Aktivitäten größer und der Lernprozess kann präzisere Ergebnisse liefern.

Beim Start von SIOX (egal ob nur ein Prozess, eine Komponente oder der komplette Deamon gestartet wird) wird stets eine Konfiguration geladen, welche alle benötigten Module beschreibt, die ebenfalls geladen werden müssen. Viele Module besitzen eine Schnittstelle an der Plugins entwickelt werden können, um SIOX anzupassen und zu erweitern. Diese Schnittstellen werden in Kapitel 4.2 weiter spezifiziert, da die von mir entwickelten Module genau auf diesen Schnittstellen aufbauen.

2.1.1 SIOX - Lowlevel API

SIOX wird stets mit mindestens einem instrumentierten Software Layer und der lowlevel C-API gestartet. Währen des Starts der Lowlevel C-API werden Konfigurationsdaten gelesen und alle global benötigten Module geladen. Beim Start einer logischen Komponente wie der POSIX Schicht, werden alle spezifizierten Komponenten der Schicht in der Konfiguration gelesen und die entsprechenden Module geladen. Eine Reihe von Modulen wird direkt von der lowlevel Bibliothek benötigt. Folgende Darstellung orientiert sich an [2] [1]

- *Ontology*: Das Ontologie-Modul bietet Zugriff zu einer beständigen Darstellung aller Aktivitätsattribute wie beispielsweise Funktionsparameter
- *SystemInformationGlobalIDManager*: Dieses Modul bietet eine Abbildung der physikalischen Hardware und der Software Komponenten auf eine globale ID
- *AssosiationMapper*: Die ersten beiden Module enthalten Daten, welche nur sehr selten geändert werden. Schnell wechselnde Daten werden in eine separate Datenstruktur gespeichert. Dieses Modul beinhaltet daher Laufzeitinformationen, es ist mit einem effizienten Update Mechanismus ausgestattet.
- *ActivityMultiplexer*: Der ActivityMultiplexer ist ein Plugin, welches Aktivitäten entgegen nimmt und diese an Plugins weiterleitet, die sich an diesem Modul registriert haben. Dies kann sowohl synchron als auch asynchron geschehen.

Zusätzlich nutzen die Bibliotheken Hilfsklassen um Aktivitäten zu erzeugen, Module zu laden und den internen Overhead zu beobachten. Im folgenden Abschnitt werde ich die Instrumentierung eines Programms beschreiben.

[1] Die Tabellen wurden aus dem Paper übersetzt.

2.1.2 Instrumentierung

Instrumentierung kann zu einem sehr aufwändigen Problem werden. Da SIOX mit fremden Programmen benutzt wird, kann dies zu einem ernstzunehmenden Wartungsaufwand führen. Dies ist nicht erwünscht. Der manuelle Eingriff in den Code soll auf einem Minimum gehalten werden. Daher erzeugt SIOX die Instrumentierung für angegebene Schichten wie zum Beispiel MPI. SIOX nutzt dafür den siox-wrapper-generator, wobei es sich um ein Python Werkzeug handelt.

Es werden Kommentare vor die aufzurufende Funktion gesetzt, die vom siox-wrapper-generator genutzt werden, um mithilfe von Templates den tatsächliche Quellcode zu generieren. Dieses Vorgehen hat den Vorteil, dass durch Austausch der Templates viele verschiedene Resultate erzielt werden können.

2.1.3 Existierende Module

Während der Entwicklung von SIOX wurden bereits sehr viele grundlegende Module und auch eine Reihe von Plugins für jedes dieser grundlegenden Module geschrieben. Im Folgenden werde ich die existierenden vorstellen. Dabei werde ich zuerst auf die grundlegenden Module eingehen, bevor ich die Plugins beschreibe.

Damit SIOX intern kommunizieren kann, wurde das GIOCommunication Modul umgesetzt. Dieses nutzt GLIB IO Sockets für die Kommunikation und bietet TCP/IP und Unix Sockets. Für die Kommunikation in SIOX wird für jeden Kommunikationspartner zwei Threads gestartet, einer zum Senden und einer zum Empfangen von Nachrichten.

2.1.4 Zusammenwirken einer typischen SIOX Konfiguration

Wie bereits in den vorangegangenen Abschnitten beschrieben, wird zum Start von SIOX eine Konfigurationsdatei eingelesen und es werden eine Reihe von Modulen und Plugins gestartet. Eine Übersicht ist in Abbildung 2.1 zu sehen. In der Abbildung sind Module in einem Prozess und im Deamon in ihrem Zusammenwirken dargestellt. Dabei sind die Interaktionen des Monitoring Pfades orange, und die des Optimierungspfades lila dargestellt. Im Folgenden möchte ich das Zusammenspiel dieser einzelnen Teile anschaulich darstellen. Dazu nutze ich eine Beispielkonfiguration die in der Abbildung 2.1 zu sehen ist, bei der der SIOX-Deamon gestartet wird und MPI und POSIX instrumentiert werden. Des weiteren bezieht sich das Beispiel auf eine Ausführung auf nur einer Maschine. Dies vereinfacht es insofern als das keine Kommunikation zwischen Knoten stattfindet.

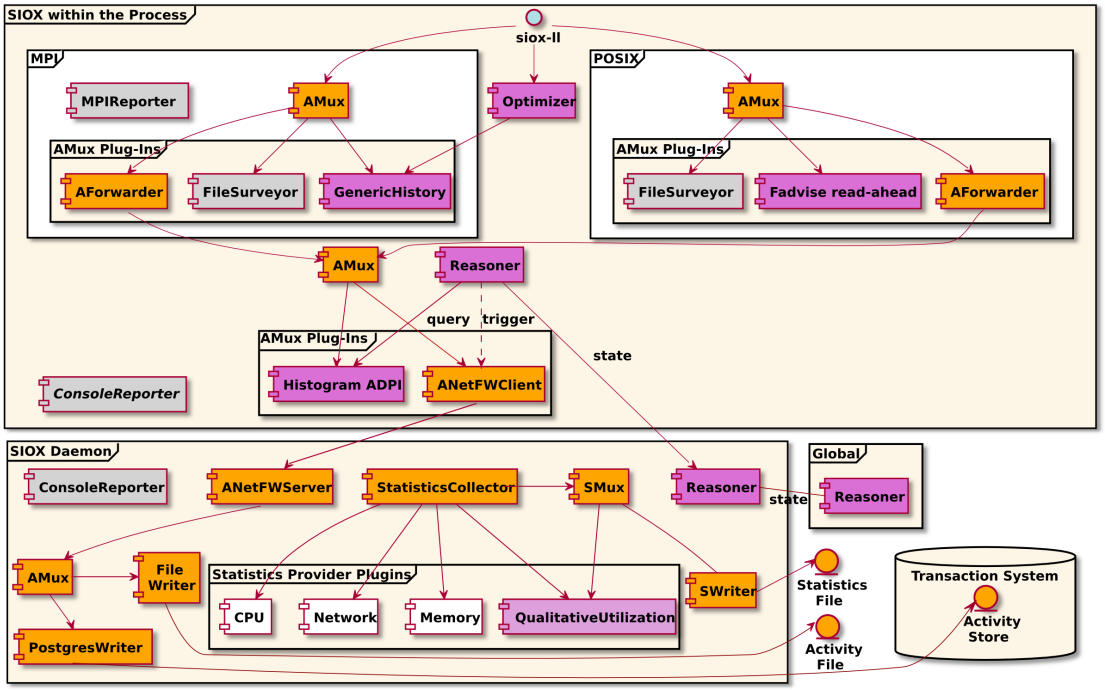

Abbildung 2.1: Beispielkonfiguration von SIOX Modulen. Quelle: [2]

In dem Beispiel überträgt der Client die beobachteten POSIX und MPI Aktivitäten an den Deamon. Dieser sammelt zudem noch Systemstatistiken. Im Folgenden werde ich genauer auf die einzelnen Module eingehen.

- *ConsoleReporter*: Der Reporter schreibt alle internen Statistiken in die Konsole oder eine Datei. Anhand dessen kann der Overhead und das veränderte Systemverhalten, das durch SIOX entsteht, vom Nutzer analysiert werden.
- *MPIReporter*: Dieser Reporter nutzt MPI um numerische Laufzeitinformationen zu sammeln und als Statistik auszugeben.
- *Optimizer*: der Optimizer bietet SIOX eine einfache Schnittstelle, die Anfragen an Module weiterleitet, welche wiederum aktuell optimale Parameter zurückliefert.
- *ActivityMultiplexer*: Der ActivityMultiplexer bietet die Möglichkeit ankommende Aktivitäten synchron und asynchron an registrierte Plugins weiterzureichen. Da bei asynchroner Bearbeitung der ankommenden Aktivitäten auch parallele Verarbeitung stattfinden kann, ist jedes Plugin selbst dafür verantwortlich Threadsafe zu sein.

Im Folgenden werden die implementierten Plugins für den ActivityMultiplexer näher beschrieben.

- *Reasoner*: Der Reasoner ist ein Modul, das verschiedene Aufgaben ausführt. Dabei hängt jede einzelne Aufgabe von seinem Sichtbarkeitsbereich ab. Hierbei wird in die Sichtbarkeitsbereiche Prozess, Knoten und System unterschieden. Die Reasoner fragen regelmäßig alle AnomalyDetectionPlugins ab, die in ihrem Gültigkeitsbereich sind. Diese geben Auskunft über die aufgenommenen Anomalien seit der letzten Abfrage. Somit lassen sich Statistiken über den aktuellen Status des Programmlaufs sammeln. Die Reasoner geben diese Informationen dann an den nächst höheren Reasoner weiter, der dadurch eine globalere Sicht bekommt.

- *ANetFWClient*: Dieser Client bietet einen Ringpuffer, in dem die letzten Activities gespeichert werden. Wenn der verbundene Reasoner ein Signal gibt, sendet der ANetFWClient alle wartenden Activities an einen ANetFWServer. Je nach Serverkonfiguration sendet der Server die empfangenen gesammelten Daten an einen Reasoner in verschiedenen Ebenen oder an den Deamon.

- *GenericHistory*: Das GenericHistory Plugin speichert für jeden Durchlauf eines Programms die aktivierten Optimierungen, so dass in der Lernphase die besten Optimierungen für verschiedene Sequenzen von Operationen gefunden werden können und in die Wissensbasis aufgenommen werden können. Des weiteren kann das Modul erweitert werden durch Nutzer IDs und Nutzer Dateiendungen, so dass für jeden Nutzer spezielle Optimierungen gefunden werden können.

- Histogramm ADPI: Dieses Plugin kann Aktivitäten in verschiedene Kategorien einsortieren und diese dann als Statistik an den Reasoner weiterleiten.

- *FileSurveyor*: Der FileSurveyor überwacht Lese-, Schreib- und Suchanfragen an Dateien, und kategorisiert sie zu sequenziellen und zufälligen Zugriffen. Diese Informationen werden dem Nutzer nach Beenden des Programms dargestellt.

- *FadviceReadAhead*: Hierbei handelt es sich wiederum um ein Plugin für den ActivityMultiplexer, Es überwacht POSIX Aufrufe und kann selbst bei Bedarf POSIX_fadvice() aufrufen.

Nun folgen die *StatisticsProviderPlugins*. In SIOX sind Momentan 5 StatisticsProviderPlugins umgesetzt. Die verschiedenen Plugins sammeln Informationen über die Auslastung von CPU, Arbeitsspeicher, Netzwerk und I/O Last. Das letzte Plugin sammelt Informationen und aggregiert die Informationen der anderen Plugins. Dabei wird folgendermaßen vorgegangen: Die StatisticsProviderPlugins sammeln alle Daten die von SIOX benötigt werden. Der *StatisticsCollector* fragt alle StatisticProvidePlugins nach den aufgenommenen Statistiken ab und leitet diese an den *StatisticsMultiplexer* weiter. Dieser wiederum verteilt die Statistiken an alle angeschlossenen Plugins.

2.1.5 Wissensgenerierung

In SIOX wird Wissen mit den Aktivitätsinformationen aus einer Datenbank generiert. Diese Datenbank wurde während der Programmläufe mit Aktivitäten gefüllt, die aus verschiedenen Parametern besteht, siehe dazu Abbildung 2.2. Dazu wird offline Maschinen lernen benutzt. Die Ergebnisse dieses Maschinenlernens werden zusammen mit Systemparametern in der Wissensbasis gespeichert. [3]

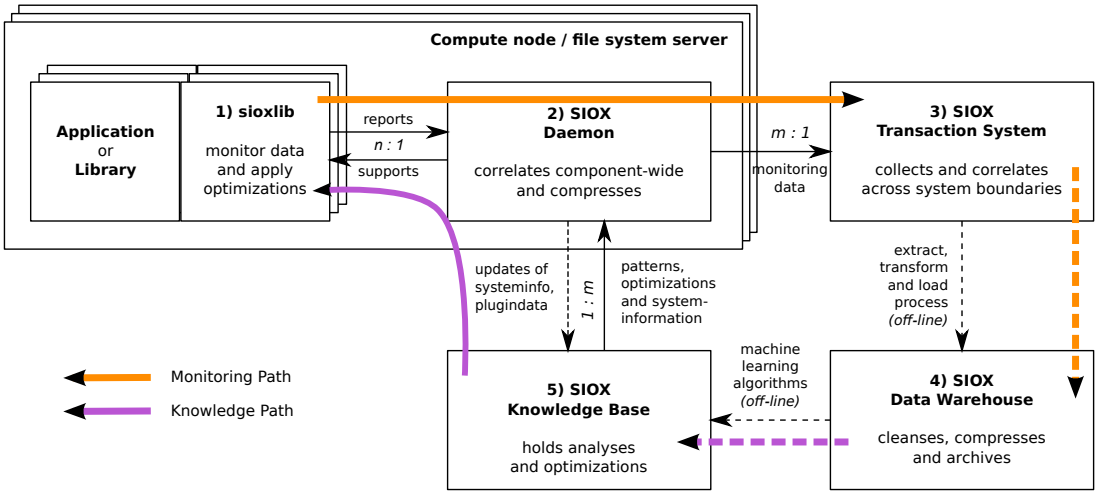

Abbildung 2.2: Die Hauptkomponenten von SIOX. Quelle: [2]

Die Wissensbasis widerum wird von Modulen von SIOX im Programmlauf genutzt, um damit den Programmablauf zu optimieren.

Im folgenden Abschnitt werden die Grundlagen von MPI dargestellt.

2.2 MPI

MPI steht für Massage Passing Interface. Es handelt sich dabei um einen Standard der eine Programmierschnittstelle beschreibt. In der ersten Version dieses Standards wurden nur die Schnittstellen beschrieben, die benötigt werden, um Programme, die mehrere Prozesse nutzen, miteinander kommunizieren zu lassen. Somit gab es in der ersten Version von MPI noch keine Schnittstelle, so dass die Prozesse parallel mit dem Dateisystem kommunizieren können.

In Version 2 von MPI wurde die Schnittstelle für die parallele Kommunikation mit dem Dateisystem hinzugefügt.

Der MPI Standard selbst bietet keine Implementation der Schnittstellen, sondern definiert diese nur. Daher bieten viele große Hersteller für Hochleistungsrechner ihre eigenen Implementationen, die für die jeweilige Maschine zugeschnitten sind und somit die maximale Leistung aus der jeweiligen Maschine holen. Vorteile dieses Vorgehens liegen auf der Hand: Jeder Hersteller passt die Implementation optimal an seine Maschinen an. Nachteile dieses Vorgehens ist, dass jeder Hersteller seine Implementation warten (aktualisieren) muss. Nicht in allen Fällen laufen diese Implemetationen der MPI Schnittstelle fehlerfrei. Dies führt dazu, dass ein Anwendungsprogramm auf einem Hochleistungsrechner sehr gut läuft, auf einem anderen hingegen Fehler auftreten. Dies kann zum Teil an der Implementation des jeweiligen Herstellers liegen. Dadurch ist es notwendig, dass Dateien in einem speziellen Kompatibilitätsmodus geschrieben werden müssen, wenn sie zwischen verschiedenen Maschinen getauscht und benutzt werden. Weiteres dazu werde ich in Kapitel 2.2.1 ausführen.

Damit nicht jeder Hersteller von Hochleistungsrechnern eine eigene MPI Implementation entwickeln muss, wurde MPICH entwickelt. Dabei handelt es sich um eine frei verfügbare Implementation von MPI. Diese Implementation ist portabel und läuft auf vielen Maschinen oder Clustern.

Im nächsten Abschnitt wird es darum gehen was die MPI Fileview ist.

2.2.1 MPI Fileview

Eine MPI Fileview ist die Sicht, die ein Prozess auf eine Datei hat. Diese wird dadurch beeinflusst, wie viele Prozesse auf einer Datei gleichzeitig zugreifen. Des weiteren wird die MPI Fileview davon beeinflusst, um welche Datentypen es sich handelt. Auch muss angegeben werden, welchen Kompatibilitätsgrad die Datei später haben soll.

Wie in Abbildung 2.3 zu erkennen, spielt die Anzahl der Prozesse, die auf einer Datei gleichzeitig zugreifen, in der Fileview eine wichtige Rolle. Dies kommt insofern zum Tragen, als dass im Hochleistungsrechen häufig sehr viele Prozesse gleichzeitig den selben Code nutzen und nur die Daten mit denen jeder Prozess arbeitet unterschiedlich sind. Dies kann beispielsweise über die Fileview gelöst werden. Jeder Prozess kann nur die für ihn relevanten Daten sehen.
Weiterhin wird die Fileview vom Datentyp beeinflusst. Dies wird so umgesetzt, als dass bei der Erzeugung der Fileview angegeben wird, welcher Datentyp betrachtet werden soll.
Zum Schluss wird die Fileview vom Kompatibilitätsgrad beeinflusst. Die verschiedenen Grade der Kompatibilität werden benötigt, wenn Dateien von verschiedenen Implementationen vom MPI genutzt werden sollen. Die Grade sind [4]:

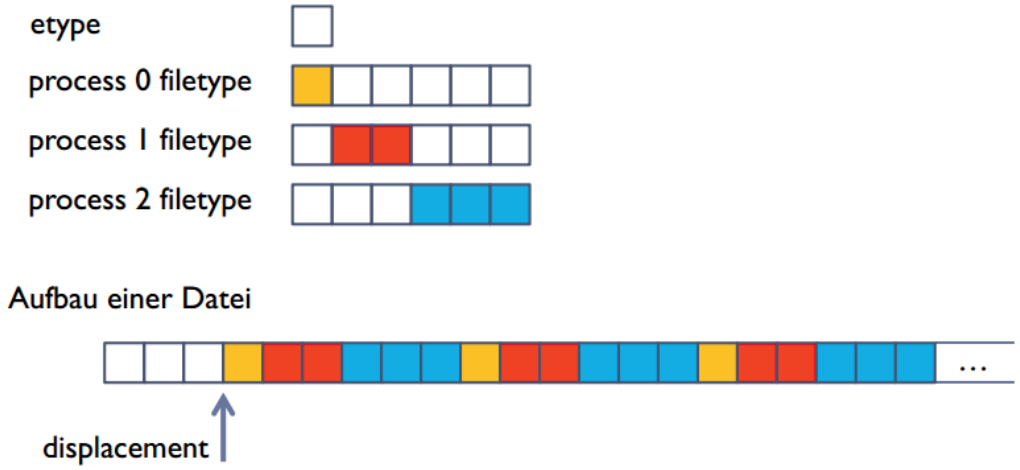

ABBILDUNG 2.3: Ansicht einer Fileview. Quelle: [1]

- *native*: Dies ist der geringste Kompatibilitätsgrad. Daten sind auf der gleichen Art von Maschine zu nutzen. Beim Kompatibilitätsgrad Native wird die Datei so abspeichert, wie sie auch im Arbeitsspeicher gespeichert werden würde. Der Vorteil hierbei ist, dass keine zusätzlichen Schritte zum Speichern benötigt werden, sondern dass der Arbeitsspeicherinhalt direkt geschrieben werden kann.
- *internal*: Hier besteht bereits ein höherer Kompatibilitätsgrad. Die Datei wird spezifisch von der MPI Implementation bearbeitet, so dass es möglich ist, Dateien zwischen verschiedenen Betriebssystemen und Computerarchitekturen auszutauschen. Es ist nur wichtig, dass alle diese Maschinen die selbe MPI Implementation nutzen.
- *external32*: In diesem Modus ist die höchste Kompatibilität erreicht. Es ist möglich Dateien zwischen verschiedenen Computern mit verschiedenen MPI Implementationen und Architekturen sowie Betriebssystemen auszutauschen. Nachteil dieser hohen Kompatibilität ist, dass viele zusätzliche Berechnungen nötig sind, um die Daten zu speichern. Dies verlangsamt den Speichervorgang.

Der nächste Abschnitt beschreibt Datasieving und die Möglichkeiten, die von MPI gegeben werden, um es zu steuern.

2.2.2 Datasieving

Dataseeking ist der Vorgang von einer Stelle in einer geöffneten Datei zur nächsten zu springen. Dies tritt auf, wenn auf eine Datei mehrere Prozesse zugreifen und jeder Prozess durch die Fileview nur den Teil der Datei sieht, der für ihn bestimmt ist. In

diesem Fall muss über die Abschnitte in der Datei gesprungen werden, die für die anderen Prozesse bestimmt sind. Zusätzlich kann dies auftreten, wenn ein Programm nur einzelne Abschnitte einer Datei lesen muss. Hierbei kann durch den MPI-Aufruf MPI_File_read_at festgelegt werden, an welcher Stelle in der Datei als nächstes gelesen werden muss. Zum Schluss gibt es noch die Möglichkeit, dass der zu lesende Datentyp bereits mit Löchern angelegt wurde. Auf diese Art und Weise muss beim Lesen des Datentyps stets über die Löcher gesprungen werden.

Dataseeking führt zu einer verminderten gesamt Datenübertragung der Nutzdaten. Bei modernen SSDs ist das kein Problem, da diese trotz random reads immer noch sehr schnell sind. Hingegen bei HDDs stellen diese Sprünge in der Datei ein sehr großes Problem dar. HHDs haben drehende Scheiben auf denen die Daten gespeichert werden und einen Lesekopf, der dann die Daten von den Scheiben liest. Bei einem sequentiellen Zugriff muss der Lesekopf sich einmal positionieren und kann dann sämtliche Daten sequentiell lesen. Wenn hingegen Sprünge stattfinden, muss der Lesekopf sich jedesmal neu positionieren und darauf warten, dass die rotierende Scheibe an der richtigen Stelle ist, um dann die Daten zu lesen. Dies führt zu einem erheblichen Zeitverlust. Es werden heute in vielen Umgebungen HDDs eingesetzt. Diese bieten mehr Speicherplatz sind bedeutend preiswerter als SSDs. Daher müssen wir auch die Probleme des ineffizienten Zugriffs auf HDDs lösen.

Um dem entgegenzuwirken, wurde Datasieving eingeführt. Hierbei werden die Löcher des Lesezugriffs mitgelesen. Somit werden zwar nicht benötigte Daten gelesen aber eine Neupositionierung des Lesekopfes der Festplatte wird vermieden. Das Datasieving macht einen Zugriff auf eine Datei mit zunehmender Sprungweite ineffizienter, da in diesem Fall sehr viele unnötige Daten gelesen werden, die später verworfen werden.

Um das Datasieving zu steuern, ist in die verbreitetste MPI Implementation MPICH die ROMIO Implentation integriert - eine Implementation des MPI-IO Protokolls. Sie bietet die Möglichkeit eine Reihe von Hinweisen an MPI zu übergeben, die beschreiben, wie die Zugriffe auf die Datei sind. Diese Hinweise werden von der MPI Implementation erkannt, um damit den Zugriff effizienter zu gestalten. Dazu gehören in der ROMIO Implementation auch die folgenden vier Hinweise [5]:

- romio_ds_read: Dieser Hinweis wird genutzt um das Datasieving beim lesenden Zugriff ein- oder auszuschalten, bzw. um die interne Heuristik zu verwenden mit “automatic“
- romio_ds_write: Dieser Hinweis wird so wie der Erste benutzt um Datasieving ein- und auszuschalten. Dabei erfolgt die Optimierung für den schreibenden Zugriff.

- ind_rd_buffer_size: Dies übergibt einen beliebigen Wert für den Puffer in Bytes, der von der Platte pro Leseoperation gelesen wird. Der Standardwert ist 4194304 Bytes oder 4 MiB.
- ind_wr_buffer_size: Mit diesem Hinweis kann die Größe des Schreibpuffers in Bytes übergeben werden. Die angegebene Größe wird in einem Stück auf die Platte geschrieben. Der Standardwert ist 524288 Bytes oder 5 KiB.

Im nächsten Kapitel werde ich eine systematische Analyse der erzielbaren Leistung bei aktiviertem und deaktiviertem Datasieving durchführen.

Kapitel 3

Systematische Analyse

In diesem Kapitel werde ich analysieren welche Leistung mit Datasieving erreicht werden kann. Dazu wurde im Rahmen dieser Arbeit ein Benchmark entwickelt, der die Nutzdatenrate mit und ohne Datasieving für verschiedene Lesegrößen und Muster ermittelt. Beginnen werde ich mit der Beschreibung dieses Benchmarks.

3.1 Benchmark

Der Benchmark ist darauf ausgelegt eine Vielzahl von verschiedenen Zugriffsmustern abzubilden. Dazu bekommt er als Eingabewerte zunächst die Anzahl der Datentypen die pro Zugriff gelesen oder geschrieben werden soll. Der nächste Parameter ist die Menge an Daten, in Megabyte, die vom Dateisystem gelesen oder geschrieben werden soll. Als weiterer Parameter muss das Muster übergeben werden. Dieses ist in folgender Form anzugeben:
Offset,Nutzdatenblockgröße-Offset,Nutzdatenblockgröße-Offset,...
Das Muster kann eine beliebige Anzahl an solchen Blöcken aus Nutzdatengröße und Offset haben. Beide Werte werden in Byte angegeben. Der vierte Parameter ist der Pfad zur Datei, auf die zugegriffen werden soll. Der vorletzte gibt den Zugriffstyp an (“read“ oder “write“). Als letzter Parameter wird die Puffergröße für das Datasieving (in Byte) angegeben, wobei 0 für für deaktiviertes Datasieving steht.

Im Benchmark wird für die Fileview der Kompatibilitätsgrad “nativ“ angenommen um die beste Leistung zu erzielen. Die Anzahl an Dateioperation wird nach Gleichung 3.1. berechnet.

$$\text{Anzahl an Dateioperationen} = \frac{\text{Gesamtzugriffsmenge}}{\text{Anzahl Datentypen pro Zugriff} * \text{Extent des Datentyps}} \tag{3.1}$$

Für die Berechnung der Nutzdatenrate, wird die Zeit für alle Zugriffe auf die Datei gemessen. Diese Zeit wird durch die Menge an Nutzdaten, auf die zugegriffen wurden, geteilt.

3.2 Experimente und Methodik

In Abbildung 3.1 ist der Datentyp dargestellt, der für die Messungen verwendet wurde. Dabei wurden zwei Blöcke Nutzdaten gelesen, zwischen denen sich ein Loch befindet. Zusätzlich befindet sich am Ende des Datentyps ein Loch, dass zu einem Abstand zwischen zwei Lesevorgängen führt. Die Größe eines Nutzdatenblocks ist jeweils bei der Auswertung angegeben. Die Lochgröße wurde so gewählt, dass der Füllstand, der in der Auswertung angegeben wird, erreicht wird. Der Extent des Datentyps ist die Gesamtgröße. Diese berechnet sich bei diesem Muster als Summe aus zwei mal der Größe eines Nutzdatenblocks und zwei mal der Größe eines Loches.

So ergeben sich bei einem Nutzdatenblock von 1024 Byte und einem gewünschten Füllstand von 98,43% eine Lochgröße von 16 Byte und ein Extent von 2080 Byte. Es wird pro Lesezugriff einmal der angegebene Datentyp gelesen.

Abbildung 3.1: Zugriffsmuster des Benchmark

Für die Messungen wurde zwischen jedem Durchlauf der Cache geleert, so dass keine Verfälschungen zustande kommen, die durch bereits vorhandene Daten entstehen. Jede Messung wurde fünf mal durchgeführt und daraus dann der Mittelwert gebildet.

Messungen wurden für Nutzdatenmengen in 10er Potenzen (1KiB, 10KiB, 100KiB, 1000KiB und 10000KiB), sowie für Nutzdatenmengen in 2er Potenzen (16KiB, 64KiB, 256KiB, 1024KiB, 4096KiB und 16384KiB) durchgeführt. Dies hat den Hintergrund, dass das Dateisystem Blockgrößen in 2er Potenzen hat, was möglicherweise zu Optimierungen führt. Die Lochgröße im Datentyp wurde zwischen dem 0.0X und dem 12 fachen der Nutzdatenmenge, die tatsächlich zugegriffene Menge an Bytes eines Nutzdatenblocks, variiert. Für die Auswertung wird hieraus der Füllstand, entsprechend Gleichung 3.2, abgeleitet.

$$Füllstand = \frac{Datenmenge}{Lochgröße + Datenmenge} = \frac{Datenmenge}{ExtentdesDatentyps} \tag{3.2}$$

Darstellung Die Achsen der Diagramme zeigen auf der Y-Achse die erzielte Nutzdatenrate und auf der X-Achse den Füllstand und den Extent des jeweils zu lesenden Blocks.

Die Nutzdatenrate ist die Datenrate, die angibt, mit welcher Geschwindigkeit Nutzdaten übertragen wurden. Beim Datasieving werden die nicht benötigten Daten mitgelesen um sequentiell lesen zu können Dabei wird dann ein Teil der gelesenen Daten wieder verworfen. Die Auswirkungen auf die Nutzdatenrate wird in diesem Kapitel untersucht. Der Extent gibt die Gesamtgröße eines zu lesenden Blocks an. Somit sind in diesem auch die nicht benötigten Teile enthalten, die zum Teil gelesen aber später verworfen werden.

Durchgeführte Messungen Die Messungen wurden mit lesenden Zugriffen auf eine vorher erstellte Datei durchgeführt. Dabei wurde sowohl mit als auch ohne Datasieving gemessen. Die Messung wurde für verschiedene Nutzdatenmengen durchgeführt. Dabei wurde jede Konfiguration fünf mal wiederholt und daraus ein Mittelwert gebildet.

Beschreibung des Testsystems Die Tests wurden am Cluster des Arbeitsbereichs Wissenschaftliches Rechnen am DKRZ (Deutsches Klimarechenzentrum) durchgeführt. Dieses ist wie in Abbildung 3.2 aufgebaut. Die Westmere Knoten sind die Rechenknoten und die Sandy Bridge sind die Speicherknoten in dieser Konfiguration.

- 10 Westmere Knoten
  - Prozessor: Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz
  - Arbeitsspeicher: 12 GiB RAM
  - Betriebssystem: Ubuntu 12.04
  - MPI-Version: OpenMPI 1.6.5
- 10 Sandy Bridge Knoten
  - Prozessor: Intel(R) Xeon(R) CPU E31275 @ 3.40 GHz
  - Arbeitsspeicher: 16 GiB RAM
  - Festplatten: WDC WD20EARS-07MVWB0 (1 Festplatte pro Server)
  - Betriebssystem: Ubuntu 12.04
  - Dateisystem: Lustre 2.5

Der erste Abschnitt dieses Kapitels beschäftigt sich mit den Lesegrößen der 10er Potenzen.

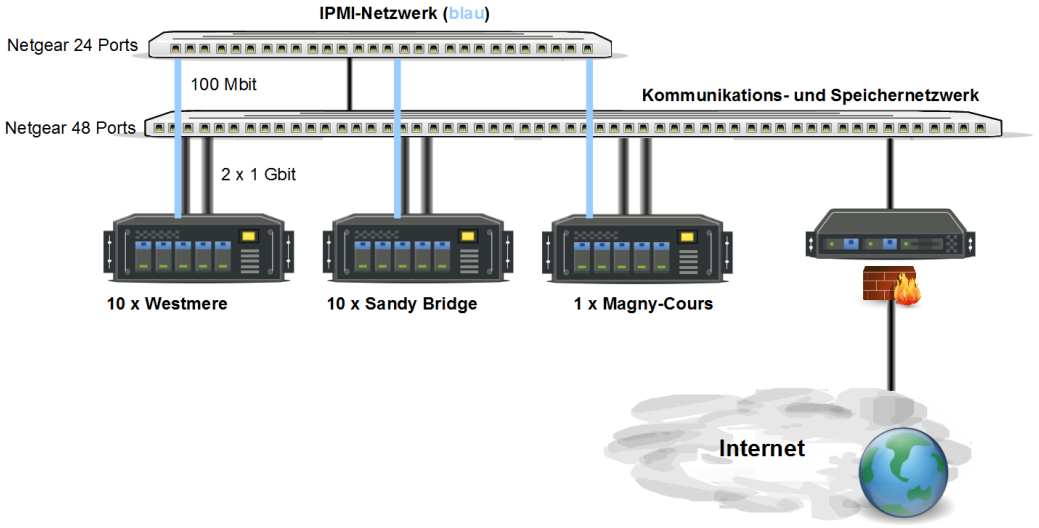

ABBILDUNG 3.2: Physikalische Sicht des Clusters [1]

3.3 Beobachtungen bei Nutzdatenmengen in 10er Potenzen

Abbildung 3.3 zeigt die kleinste gemessene Größe. Dabei wurden zwei mal 1KiB vom Dateisystem gelesen. Zwischen den zwei Blöcken liegt ein verschieden großes Loch von 0 Byte bis zu 11 KiByte. Hierbei ist sehr gut zu erkennen, dass:

- Datasieving bringt einen starken Leistungsgewinn
  - Bei kleinen Löchern in der Datenstruktur müssen die Festplatten des Dateisystems, bei deaktiviertem Datasieving, häufig Teile auslassen und den Lesekopf neu positionieren, dies führt zu geringeren Datenraten
- Bei sinkendem Füllstand sinkt die Datenrate bei aktiviertem Datasieving
  - Die Menge an verworfenen Daten beim Datasieving wird größer
- aktiviertes Datasieving ist auch bei geringen Füllständen effizienter als deaktiviertes Datasieving

In Abbildung 3.4 wurde mit einer Nutzdatengröße von zwei mal 10 KiByte gemessen. Dabei ist zu erkennen, dass:

- die maximalen Datenraten der einzelnen Kurven gegenüber Abbildung 3.3 sind angestiegen

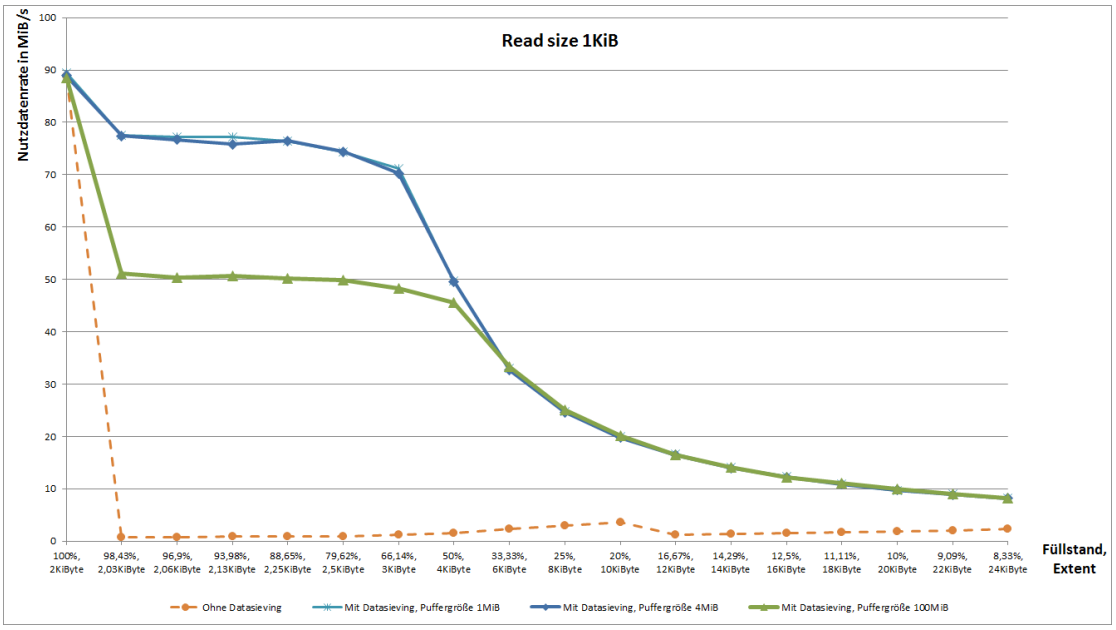

Abbildung 3.3: Blocklesegröße 2x1KiB

  - die Menge der gesamt gelesenen Daten ist gestiegen
  - bei einer größeren sequentiellen Menge an gelesenen Daten steigt grundsätzlich die Datenrate, bis sie das Maximum der Leistung der Festplatte erreicht hat.
  - die Datenmenge ist 10 mal so groß wie in der Abbildung 3.3, daher der Leistungsanstieg, der im Fall von deaktiviertem Datasieving etwa um Faktor 10 größer ist als in Abbildung 3.3

- aktiviertes Datasieving führt bei allen gemessenen Puffergrößen zu einem Leistungsvorteil gegenüber abgeschaltetem Datasieving

Da auch im Bereich von 33% und 25% Füllstand kein Unterschied sichtbar ist, könnte Datasieving mit einer kleinen Puffergröße dauerhaft eingeschaltet sein um die Maximale Leistung zu erhalten.

Bei der Messung der wiederum 10 fachen Menge an Nutzdaten, die in Abbildung 3.5 zu sehen ist, ist feststellbar, dass:

- die Maximaldatenrate wiederum leicht angestiegen ist
- In diesem Fall kann bei einem Füllstand größer als 50% kein Unterschied in der Leistung zwischen aktiviertem und deaktiviertem Datasieving erkannt werden
- Erstmals ist bei Füllständen kleiner als 50% deaktiviertes Datasieving effizienter als aktiviertes Datasieving

Abbildung 3.4: Blocklesegröße 2x10KiB

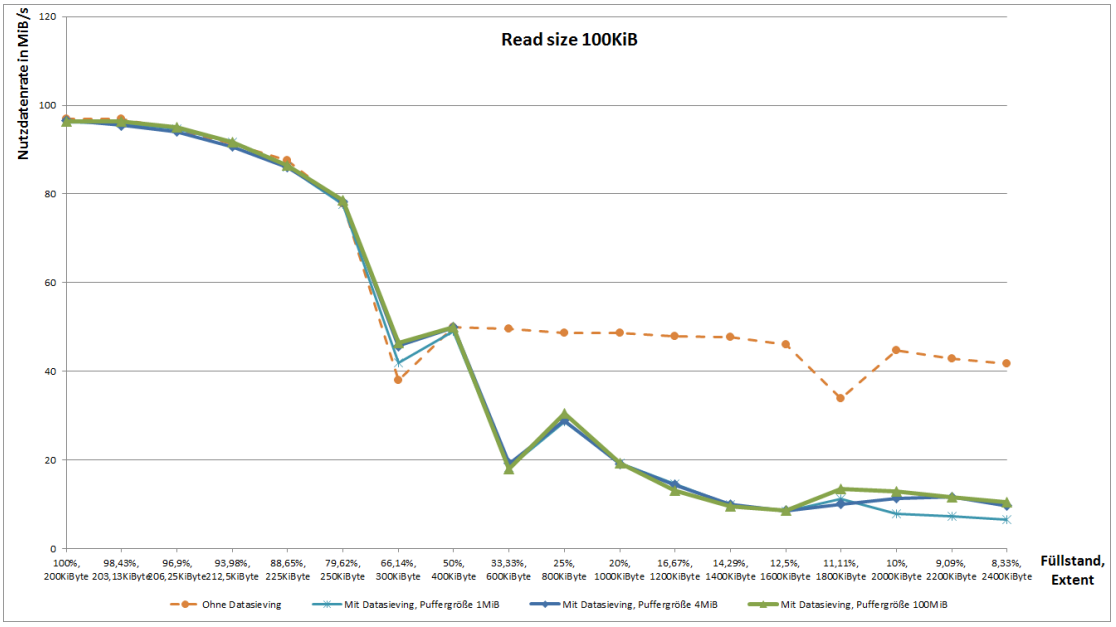

Abbildung 3.5: Blocklesegröße 2x100KiB

Die Messreihe mit einer Leseblockgröße von 1000 KiByte ist in Abbildung 3.6 zu sehen. Hier ergeben sich folgende Resultate:

- erste Unterschiede in den verwendeten Puffergrößen bei aktiviertem Datasieving sind erkennbar

  - der Extent ist größer als die Puffergröße, daher müssen bei einer Puffergröße von 1MiByte mehrere Blöcke gelesen werden
  - bei einer Puffergröße von 1MiByte werden Löcher bei geringeren Füllständen ausgelassen, daher die erhöhte Nutzdatenrate im Bereich eines Füllstandes kleiner als 50%

- bei Füllständen größer als 50% sollte Datasieving mit einem Puffer von 4MiByte aktiviert sein
- bei Füllständen kleiner als 50% sollte Datasieving deaktiviert werden

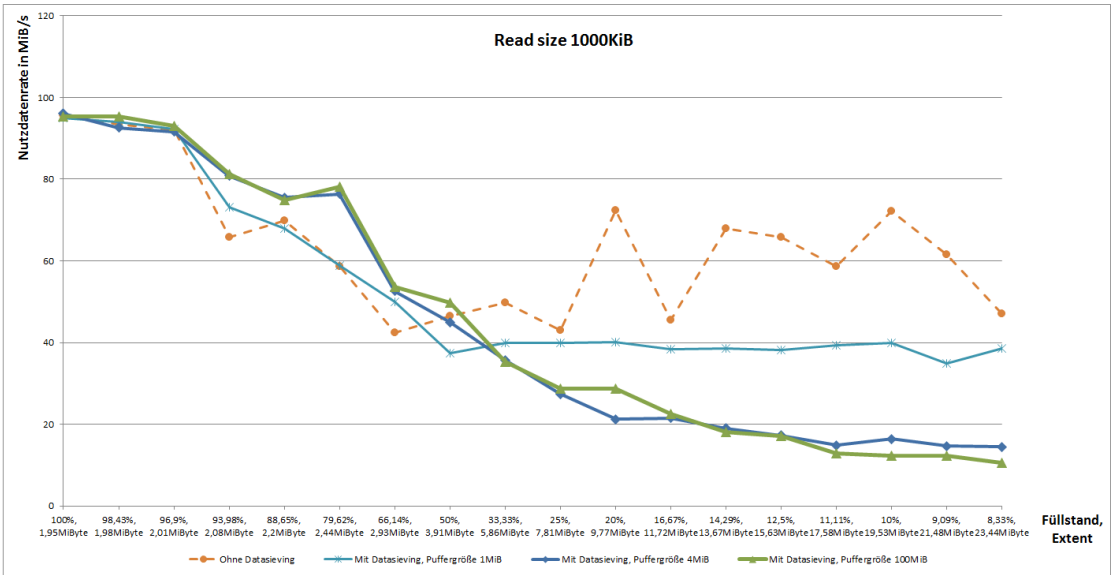

Abbildung 3.6: Blocklesegröße 2x1000KiB

Die letzte Messreihe dieses Abschnitts zeigt eine Leseblockgröße von 10.000 MiByte, wie in Abbildung 3.7 zu sehen.

- Datasieving ist für alle Füllstände empfehlenswert ist. Die Puffergröße muss variiert werden, um maximale Leistung zu erreichen.
  - Bei großen Füllständen ist eine Puffergröße von 4MiByte optimal, da kleine Löcher mitgelesen werden und so sequentiell gelesen wird.
  - Bei kleinen Füllständen, kleiner als 80% ist eine Puffergröße von 1MiByte empfehlenswert. Es werden so Teile der Löcher ausgelassen. Trotzdem können Optimierungen durch das Dateisystem, das Blöcke in 2er Potenzen hat, genutzt werden.
  - Bei kleinen Füllständen ist eine geringe Puffergröße zusätzlich vorteilhaft, da die verwendete Nutzdatenblockgröße etwas weniger als 10MiByte groß ist.

Dadurch werden bei aktiviertem Datasieving und einer Puffergröße von 1MiByte genau 10MiByte gelesen und das Loch wird übersprungen. Erst dann wir der zweite Block an Nutzdaten gelesen. Hingegen bei einer Puffergröße von 4MiByte werden nicht nur 10MiByte Nutzdaten pro Block gelesen sondern 12MiByte, was dazu führt, dass 2MiByte verworfen werden bevor zum nächsten Nutzdatenblock gesprungen wird.

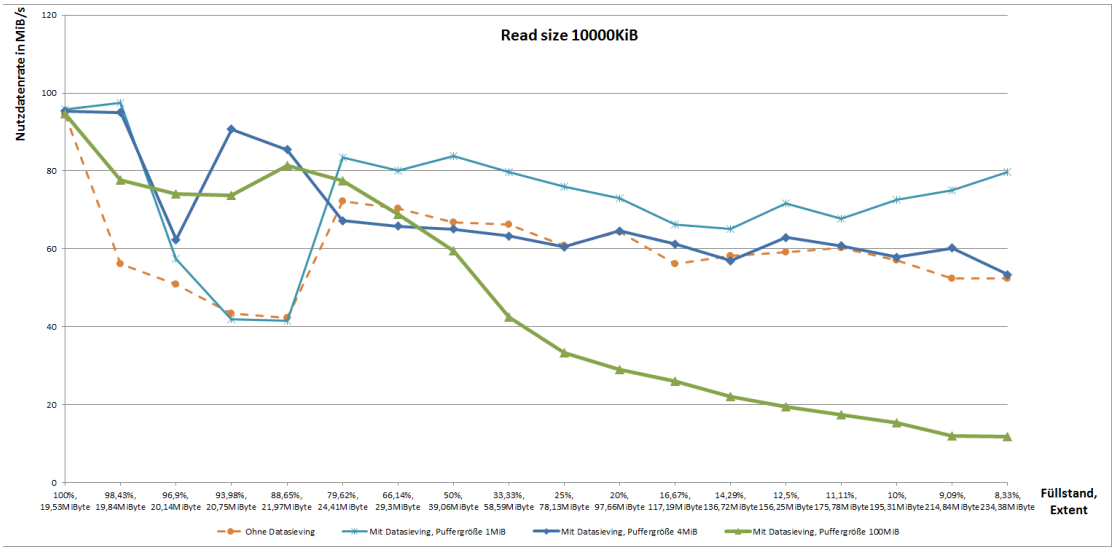

Abbildung 3.7: Blocklesegröße 2x10000KiB

Im ersten Teil dieser Analyse ist erkennbar, dass große Nutzdatenblockgrößen zu besseren Nutzdatenraten führen. Weiterhin kann geschlussfolgert werden, dass große Füllstände dazu führen, dass bei aktiviertem Datasieving bessere Ergebnisse erreicht werden als bei deaktiviertem Datasieving.

Im zweiten Teil dieses Kapitels werde ich eine Analyse der erzielbaren Leistung bei Leseblockgrößen, die ein vielfaches der 2er Potenzen sind, durchführen.

3.4 Beobachtungen bei Nutzdatenmengen in 2er Potenzen

Im Weiteren führe ich Messungen durch, bei denen Lesegrößen in 2er Potenzen verwendet werden.

Die erste Abbildung 3.8 zeigt die Leistung bei einer Nutzdatenblockgröße von 2 mal 16 KiByte. Erkennbar ist:

- die Leistung mit aktiviertem Datasieving ist maximal

- die Differenz zwischen aktiviertem und deaktiviertem Datasieving ist besonders groß bei einem Füllstand größer als 60%

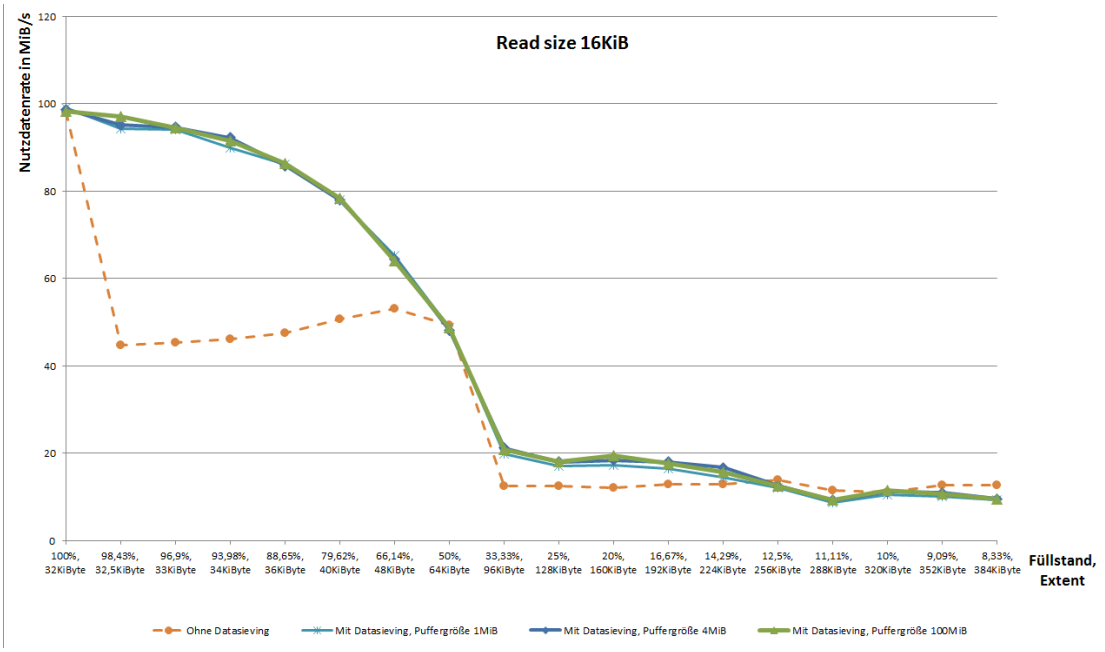

Abbildung 3.8: Blocklesegröße 2x16KiB

Abbildung 3.9 zeigt eine Nutzdatengröße von zwei mal 64 KiByte.

- Bei einem hohen Füllstand, größer als 33%, ist der Nutzdatendurchsatz von aktiviertem und deaktiviertem Datasieving gleich hoch
- Ab einem Füllstand kleiner als 33% ist deaktiviertes Datasieving effizienter als aktiviertes

In der nächsten Messung wurde die Blockgröße weiter erhöht, so dass ein zu lesender Block nun 256 KiByte groß ist. So werden pro Leseaufruf zwei mal 256 KiByte gelesen. Dies ist in Abbildung 3.10 zu sehen.

- ähnlich zu Abbildung 3.9 ist es leistungssteigernd Datasieving ab einem Füllstand größer als 40% zu nutzten
- bei allen Füllständen, geringer als 40%, sollte Datasieving deaktivieren werden, um maximale Leistung zu erreichen.
- aktiviertes Datasieving mit einer Puffergröße von 1MiByte erzielt eine höhere Nutzdatenrate als aktiviertes Datasieving mit größeren Puffern

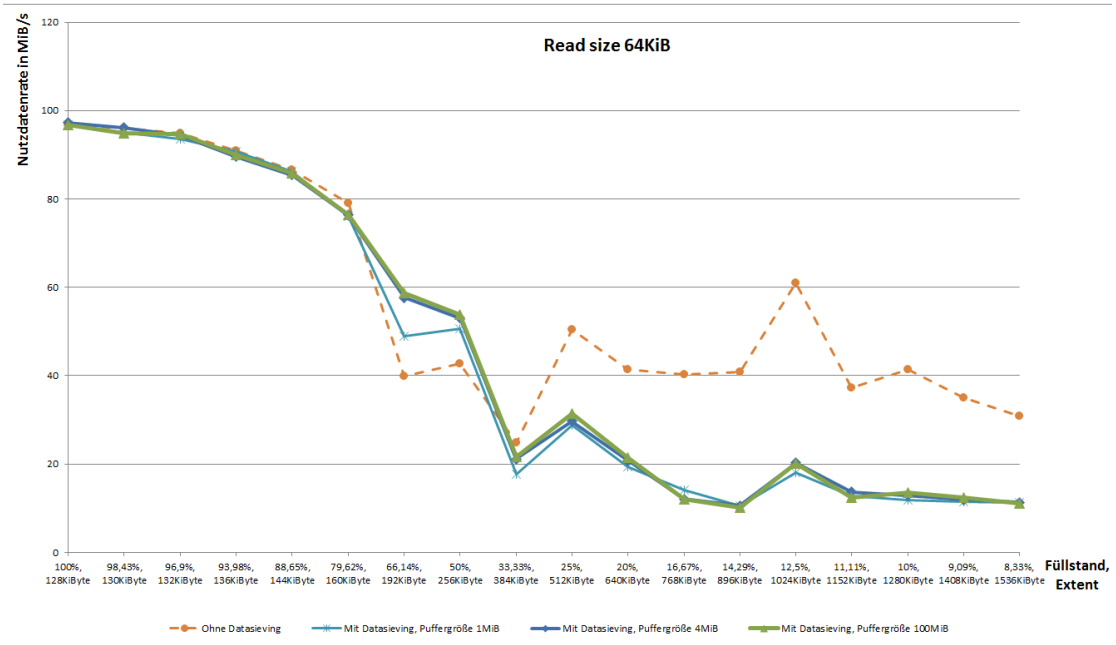

Abbildung 3.9: Blocklesegröße 2x64KiB

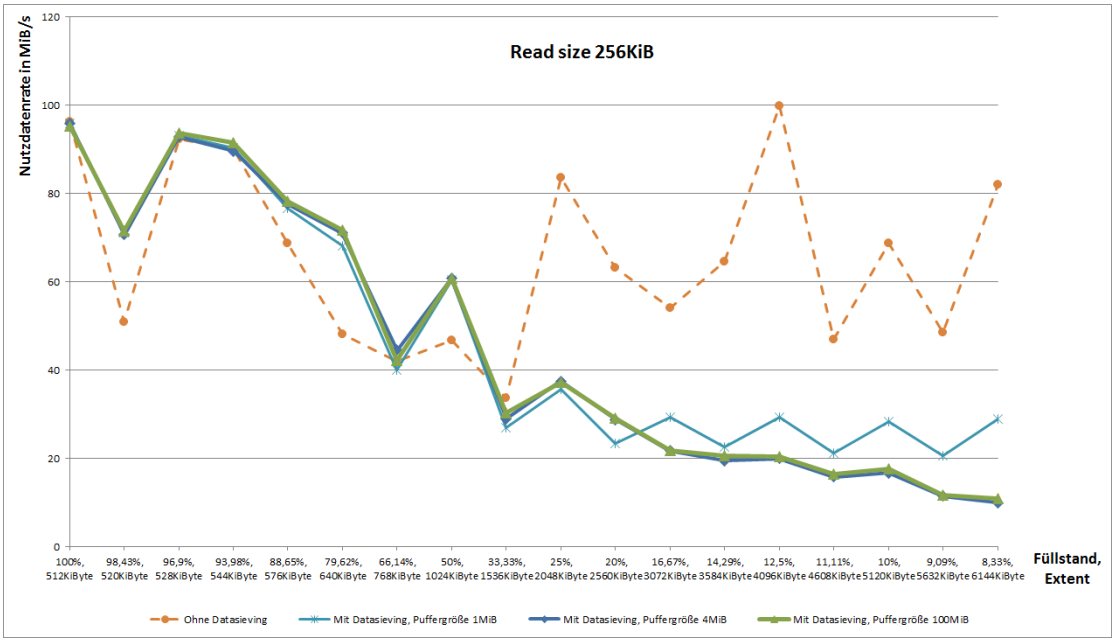

Abbildung 3.10: Blocklesegröße 2x256KiB

Nun folgt eine Messung mit einer Lesegröße von zwei mal 1 MiByte, siehe Abbildung 3.11. In dieser Messung ist zu erkennen, dass:

- aktiviertes Datasieving mit einem Puffer von 1 MiByte die selbe Datenrate hat wie deaktiviertes Datasieving
  - bei einer Zugriffsgröße von 1 MiByte wird in beiden Fällen die selbe Menge an Daten gelesen, da die Nutzdatengröße gerade 1MiByte beträgt
- bei einem hohen Füllgrad ist Datasieving mit einem Puffer der größer als 1 Mibyte ist (4 MiByte und 100 Mibyte) tendentiell effizienter
- Bis zu einem Füllgrad kleiner als 50% ist Datasieving mit 1 MiByte und deaktiviertes Datasieving deutlich effizienter als aktiviertes Datasieving mit großen Puffern

Abbildung 3.11: Blocklesegröße 2x1024KiB

In Abbildung 3.12 ist die Messung dargestellt, in der die Nutzdatengröße pro Zugriff zwei mal 4 MiByte groß ist. Hierbei kann festgestellt werden:

- die Effizienz von deaktiviertem und aktiviertem Datasieving mit einem Puffer bis 4 MiByte ist gleich groß
- ab einem Füllstand größer als 85% ist Datasieving mit einem Puffer von 100 MiByte effizienter als bei den anderen Messungen

  - ein kleines Loch existiert und nur eine Puffergröße größer als 4 MiByte liest dieses Loch mit und hat somit durch den Vorteil des Sequentiellen lesens eine höhere Datenraten

- ab eine Füllstand geringer als 85% ist aktiviertes Datasieving mit einer Puffergröße von 1 MiByte optimal

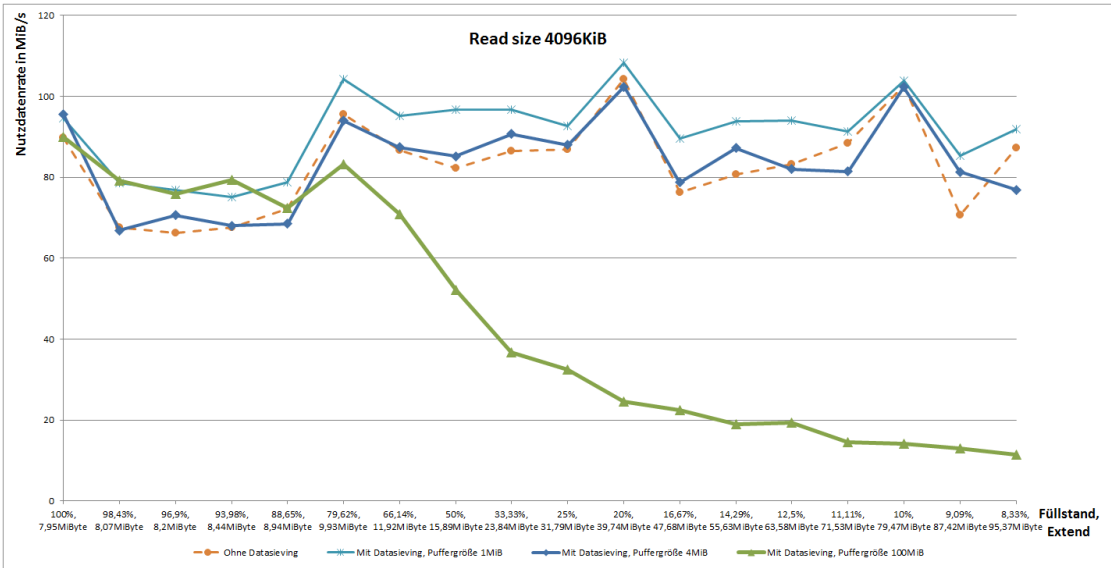

Abbildung 3.12: Blocklesegröße 2x4096KiB

Die letzte Messung (Abbildung 3.13) stellt die größte Datenmenge dar, die in diesem Test vom Dateisystem gelesen wurde.

- bei einem hohen Füllgrad, größer als 85%, ist aktiviertes Datasieving mit einem Puffer von 100 MiByte optimal

- bei geringeren Füllgraden ist deaktiviertes Datasieving mit einer Puffergröße von 4 MiByte optimal.

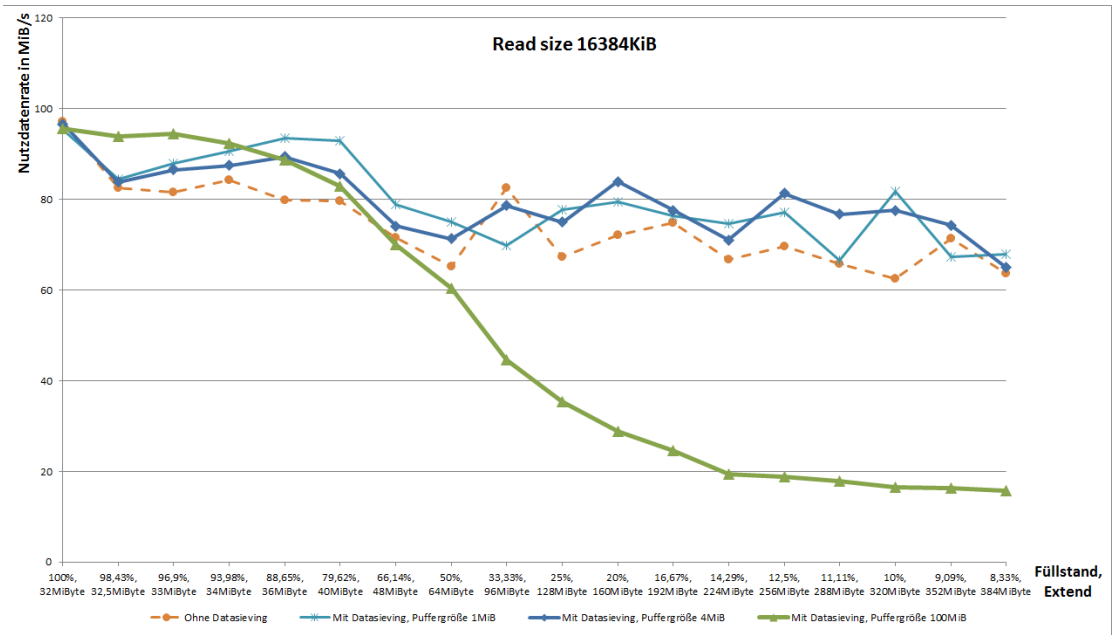

Abbildung 3.13: Blocklesegröße 2x16384KiB

3.5 Zusammenfassung der Analyse

Die Messreihen haben gezeigt, dass es bei kleinen Nutzdatenmengen tendenziell effizienter ist Datasieving einzusetzen. Hingegen bei großen Nutzdatenmengen ist es vorteilhafter Datasieving mit geringen Puffergrößen zu nutzen. Bei dem gemessenen Muster ist aufgefallen, dass es bei hohen Füllgraden und damit verbundenen kleinen Löchern in der Datenstruktur optimal ist, wenn Datasieving eingesetzt wird. Dies gilt bis zu einem gewissen Wert an Füllgrad. Dieser liegt in den meisten Fällen zwischen 40% und 60%. Wenn der Füllgrad geringer ist als dieser Werte, wird ein höherer Nutzdatendurchsatz erzielt, wenn Datasieving ausgeschaltet wird. Selbstverständlich gilt dies nicht in allen Fällen. Gerade bei kleinen Blockgrößen tritt dies nicht auf und es ist effizienter Datasieving zu nutzen.

Es ist weiterhin aufgefallen, dass die Puffergröße keinen Einfluss auf die Leistung hatte, wenn der Extent klein war. So erzielen Puffergrößen von 1 MiByte, 4 MiByte und 100 MiByte den selben Durchsatz, wenn der Extent kleiner ist als 1 MiByte. Erst wenn der Extent größer wurde, gingen die Ergebnisse auseinander. So gab es für die Puffergröße von 1 MiByte große Unterschiede ab einer zu lesenden Blockgröße von 1000 KiByte für die 10er Potenz Messungen und bei den Messungen mit 2er Potenzen begannen die Unterschiede bereits bei der Messung der 256 KiByte zu lesenden Blockgröße. Diese Unterschiede zeigten sich insofern, als das der Durchsatz der kleinen Puffergröße sich vom Durchsatz der größeren Puffer entfernte und dichter zum deaktivierten Datasieving zustrebte. Dies geschah sowohl positiv als auch negativ.

Allgemein war zu erkennen, dass Nutzdatenmengen in 2er Potenzen effizienter waren als Nutzdatenmengen in 10er Potenzen.

Kapitel 4

Design

Dieses Kapitel wird die Designfragen die vor der Implementierung zu beantworten waren klären. Dazu werde ich im ersten Teil auf die Unterschiede zwischen online und offline Plugins eingehen und später auf die getroffenen Entscheidungen eingehen.

4.1 Integration in SIOX

Ein Plugin, welches mit Aktivitäten arbeiten soll, kann in SIOX an verschiedenen Modulen angeschlossen werden. Hierbei gibt es zwei Möglichkeiten, die vorher bedacht werden müssen: soll das Plugin online mit den Aktivitäten arbeiten oder offline. Bei der Wahl zu einem Plugin für den Online Teil in SIOX muss das Plugin an der ActivityMultiplexer angeschlossen werden. Für den Offline Teil gibt es den TraceReader.

4.2 Vergleich von online und offline Plugins

Grundsätzlich werden online Plugins, die mit Aktivitäten arbeiten an den ActivityMultiplexer angeschlossen und offline Plugins an den TraceReader.

Aufbau von online Plugins für den ActivityMultiplexer Der ActivityMultiplexer ist ein Modul von SIOX, dass beim Start des Deamon und des Prozesses geladen wird. Zusätzlich werden zu diesem Zeitpunkt auch alle registrierten Plugins geladen.

Während des Programmlaufs nimmt der ActivityMultiplexer jede Aktivität entgegen und reicht sie an alle registrierten Plugins sowohl synchron als auch asynchron weiter. Dies hat den Vorteil, das jedes Plugin entscheiden kann, welche Möglichkeit es nutzt.

Wenn ein Plugin sich für die asynchrone Variante entscheidet, muss es selbst dafür sorgen alle notwendigen Informationen sicher zu speichern mit denen es arbeitet, denn Aktivitäten werden nach deren Auftreten an das Plugin weitergereicht.

Die Plugins für dieses Modul müssen spezielle Methoden zur Verfügung stellen:

- initPlugin Funktion, diese wird beim Starten des Deamon oder Prozesses aufgerufen. In dieser Funktion wird das Plugin initialisiert. Das Plugin kann auf systemweite Services zugreifen und benötigt in der init Funktion keine Parameter.
- finalize Funktion, diese wird beim Beenden des Plugins aufgerufen und meldet das Plugin vom ActivityMultiplexer ab
- zum Empfang von Aktivitäten während eines Durchlaufs wird die Notify Funktion aufgerufen. Diese Funktion bekommt als Argument eine Aktivität mit der das Plugin arbeiten kann.

Aufbau von offline Plugins für den TraceReader Der TraceReader ist ein Tool von SIOX, welches es ermöglicht aufgezeichnete Programmläufe abzuspielen. Dazu wird während eines Programmlaufs für jeden Prozess eine Activities Datei angelegt. In dieser Datei werden sämtliche aufgetretenen Aktivitäten serialisiert gespeichert. Der TraceReader startet alle wichtigen Module und somit eine SIOX Umgebung. Diese ist sehr ähnlich zu der Umgebung, die der SIOX-Deamon erzeugt. Der Unterschied zwischen den Umgebungen liegt nur darin, dass der Deamon Aktivitäten aufzeichnet. Der TraceReader liest solch eine Aufzeichnung ein. Dazu deserialisiert er eine Activities Datei und kann dann Aktivität für Aktivität an die Plugins übergeben.

Der TraceReader nutzt spezielle Methoden, die von den Plugins zur Verfügung gestellt werden müssen. Diese sind:

- init(): die init Methode wird beim Start des TraceReader aufgerufen. Diese Methode initialisiert das Plugin. Die Argumente dieser Methode sind zum einen die gesetzten Programmoptionen und zum anderen der TraceReader selbst, der über getter Funktionen Module zur Verfügung stellt um Parameter der Aktivitäten zu analysieren.
- processNextActivity(): die processNextActivity Methode wird bei jeder neu gelesenen Aktivität aufgerufen. Innerhalb dieser Methode finden alle Berechnungen statt, die nötig sind, damit das Plugin seine Funktion erfüllt. Dabei können Funktionen auch ausgelagert werden.

Die Plugins des TraceReader müssen alle benötigten Module bei der Initialisierung vom TraceReader bekommen. Im Gegensatz dazu, greifen die Plugins des ActivityMultiplexers auf eine global verfügbare facade zu, um die benötigten Module lokal zur Verfügung zu stellen. Der nächste Unterschied ist, dass die Plugins des ActivityMultiplexers eine finalize Funktion zur Verfügung stellen. Dies müssen die Plugins, die auf den TraceReaders aufsetzen, nicht. Letzter kleiner Unterschied ist der Funktionsname des Aufrufs, um eine Aktivität mitgeteilt zu bekommen. Diese ist beim TraceReaderPlugin processNextActivity und beim ActivityMultiplexer notify.

Somit sind die Differenzen zwischen den online und offline Plugins sehr gering und es ist durchaus möglich ein Plugin für das eine Modul zu erstellen und später durch kleine Abwandlungen auch im anderen Modul zu nutzten. Dies führt zu einer sehr effizienten Methode, in der Plugins für den offline und online Teil von SIOX erstellt werden.

4.3 Entscheidungsfindung

Zu Beginn der Entwicklung mussten einige Entscheidungen getroffen werden. Dazu zählen:

- Codierung der Wissensbasis im Plugin
- soll es ein Plugin für den online oder offline Teil werden
- Auf welcher Softwareschicht arbeitet das Plugin

Codierung der Wissensbasis Hierfür mussten geeignete Datenstrukturen gefunden werden

- geschachtelte Maps
- endliche Automaten

Die Wahl fiel auf geschachtelte Maps mit der Option später endliche Automaten umzusetzen, da diese eine deutlich größere Flexibilität besitzen.

Das Plugin funktioniert mit einer gegebenen Wissensbasis. Dies hat den Vorteil, dass während des Programmlaufs kein neues Wissen gesammelt werden muss und das Programm schneller läuft. Daher wird in der init() Methode des Plugins die Wissensbasis eingelesen. Aktuell wird die Wissensbasis in Dateien gespeichert. Im Plugin selbst wird die Wissensbasis in HashMaps gehalten. HashMaps sind sehr effiziente Datenstrukturen

auf die ein einfacher Zugriff möglich ist. Die Wissensbasis besteht aus zwei Teilen. Das hat den Vorteil, dass konstante Informationen nur selten nachgeschlagen werden müssen: Der erste Teil bildet die Informationen über die aktuelle Fileview auf einen Token ab, der dann gespeichert werden kann. Dabei handelt es sich um eine sehr statische Information, da die Fileview nur selten geändert wird. Die zweite Map bildet die Parameter aus der aktuellen Aktivität auf eine Optimierung ab, die dann ausgegeben oder live angewandt wird.

Im ersten Teil der Wissensbasis, den Informationen der Fileview, werden drei Parameter genutzt, um einen eindeutigen Token zu generieren.

- *Füllstand*: Dieser gibt wie in Kapitel 3.2 beschrieben das Verhältnis von Nutzdatenmenge zu Gesamtdatenmenge an. Diese lässt sich sehr leicht berechnen, da MPI Methoden zur Verfügung stellt, um diese auszugeben. Diese sind MPI_Type_Size, welche die Nutzdatenmenge im Datentyp angibt, und MPI_Type_Extent, welche den Extent des Datentyps angibt.
- *Durchschnittliche Lochgröße*: Da ein Datentyp beliebig komplex werden kann, muss eine Reduzierung dieser Komplexität gefunden werden, um die Wissensbasis einfach zu halten.
- *Extent*: Da dieser von MPI ausgegeben wird und verschiedenen Extents, wie in Kapitel 3 nachgewiesen, zu unterschiedlichen Ergebnissen führen.

Diese drei Werte wurden genutzt, um eine Signatur von komplexen Datentypen zu erstellen.

Online oder Offline Plugin Weil eine einfache Konvertierung zwischen online und offline Teil von SIOX möglich ist, habe ich mein Plugin im offline Teil von SIOX entwickelt. Somit können alle Optimierungen wärend des Abspielens eines Traces ausgegeben werden. Um die Optimierungen transparent während eines Programmlaufs zu setzten, muss das Plugin in den online Teil konvertiert werden.

Wahl der Softwareschicht Die Wahl der Softwareschicht begrenzte sich auf MPI, da mit Datasieving gearbeitet werden soll.

4.4 Design der Codierung der Wissensbasis

Die Wissensbasis basiert, wie in Tabelle 4.1 zu sehen, auf dem Füllstand der zu lesenden Datentypen. Diese bestimmen auch den Extent. Zu dem hängt die Wissensbasis von der

Füllstand	Durchschnittliche Lochgröße	Extent	Tokentyp
98.43	16	2080	98-1
88.65	128	2304	88-1
50.00	10240	40960	50-10

Tabelle 4.1: Auszug aus der Wissensbasis Teil1, in der das Mapping von Parametern auf einen Token steht.

durchschnittlichen Lochgröße der Fileview und des Datentyps ab. Diese grundlegenden sich selten ändernden Werte generieren einen Token, auf den die Reaktionen beruhen.

Token	Anzahl der Zugriffe	Zugriffstyp	Reaktion
98-1	1	read	enable/1048576
88-1	1	read	enable/1048576
50-10	1	read	enable/1048576

Tabelle 4.2: Auszug aus der Wissensbasis Teil2, in der das Mapping vom Token und den Parametern der Aktivität auf eine Reaktion steht.

Die Reaktionen die durchgeführt werden müssen, basieren aus dem Token der Wissensbasis, die sich selten ändernde Werte enthält und den Werten der Aktivität, wie in Tabelle 4.2 zu sehen. Zu diesen zählt die Anzahl der zu lesenden oder zu schreibenden Datentypen. Des weiteren basiert die Reaktion auf dem Zugriffstyp, lesend oder schreibend.

Im Folgenden Kapitel werde ich genauer auf die Implementierung eingehen und dabei vorstellen, welche Datenstrukturen ich dazu genutzt habe.

Kapitel 5

Implementierung

5.1 Datenstrukturen

In der Implementierungphase dieser Arbeit ging es darum, die in Kapitel 4 gefallenen Designentscheidungen umzustezen. Dabei wurde zunächst das Plugin mit geschachtelten Maps als Wissensbasis umgesetzt. Später sollte es mit endlichen Automaten erweitert werden, um eine größere Flexibilität zu gewährleisten. Dazu eignete sich eine StateMachine sehr gut. Ich werde im Folgenden zuerst auf die Maps eingehen und später den Ansatz mit StateMachines beschreiben.

5.1.1 geschachtelte Maps

Eine Map ist eine Datenstruktur, die aus einem Schlüssel - Wert Paar besteht. Dabei kann der Schlüssel ein einfacher Datentyp sein und der Wert ein komplexer. Bei den von mir verwendeten geschachtelten Maps ist der Schlüssel ein Interger oder String und der Wert wiederum eine Map.

```
1 map<double,map<int,map<int,string>>> tokenMap;
```

Listing 5.1: Definition der Map für die Tokens

Der erste Schlüssel steht für den Füllstand, der zweite für die durchschnittliche Lochgröße und der dritte für den Extent. Der innerste Wert steht für den Token mit dem dann im zweiten Teil der Wissensbasis weiter gearbeitet wird.

```
1 currentViewToken = ((tokenMap.find(fillsize)->second).
2         find(avgHoleSize)->second).find(extent)->second;
```

Listing 5.2: Abfrage der Map mit vorher errechneten Werten um den Token zu erhalten

Im folgenden werde ich eine Bibliothek erläutern, mit der es möglich ist, den aktuellen Ansatz mit Maps in StateMachines umzuwandeln, um auch auf Folgen von Aktivitäten reagieren zu können.

5.1.2 StateMachine

In meinem Plugin benutze ich einen deterministischen endlichen Automaten.
Ein deterministischer endlicher Automat wird durch ein Quintupel $A := (Q, \Sigma, \delta\, q_0, F)$ beschrieben, wobei

- Q eine endliche Menge von Zuständen ist,
- Σ ein endliches Alphabet von Eingabesymbolen ist,
- $\delta : Q \times \Sigma \to Q$ die nicht notwendigerweise totale Überführungsfunktion ist,
- $q_0 \in Q$ der Startzustand ist und
- $F \subseteq Q$ die Menge der Endzustände ist

Dies wird durch die Bibliothek FSMPP [1] abgedeckt. Sie bietet eine einfache und schnelle Möglichkeit endliche Automaten zu erzeugen und zu nutzen.

Dafür wird eine eigene Klasse erstellt, die den späteren endlichen Automaten erzeugt.

```
1 class MyFSM : public FSM<char>
2 {
3 public:
4     MyFSM(void);
5 };
```

Listing 5.3: Definition der Klasse für den endlichen Automaten

Alle Zustände werden in einer Map gespeichert, die aus einem Namen für den Zustand und dem Zustand selbst besteht. Diese Map stellt das Q aus der Definition dar.

```
6     map<string, State<char>*> states;
```

Listing 5.4: Die Map für die Zustände wird erstellt

Im Anschluss daran werden die Zustände erzeugt und der Map mit einem eindeutigen Namen zugeordnet.

[1] Die Bibliothek stammt von folgender Quelle: http://fsmpp.sourceforge.net/

```
7     states["s1"] = new State<char>(this, 1);
8     states["s2"] = new State<char>(this, 2);
9     states["s3"] = new State<char>(this, 3);
10    states["s4"] = new State<char>(this, 4);
11    states["s5"] = new State<char>(this, 5);
12    states["s6"] = new State<char>(this, 6);
```

Listing 5.5: Die Zustände werden der Map zugeordnet

Nun können Übergänge zwischen den Zuständen festgelegt werden. Diese stellen die Übergangsfunktion δ aus der Definition dar. Dazu wird ein Token benötigt, der an den endlichen Automaten übergeben wird. Dieses Token sorgt dafür, dass die Transition aktiviert wird und ein Zustandsübergang stattfindet. In FSMPP ist dies so gelöst, dass die Klasse der Zustände eine Methode, addTransition, anbietet, die diese Aufgabe übernimmt. Das Alphabet von Eingabesymbolen muss in FSMPP nicht direkt definiert werden. Es werden sämtliche Eingabesymbole die möglich sind per addTransition übergeben.

```
13    states["s1"]->addTransition('r', states["s2"]);
14    states["s2"]->addTransition('w', states["s3"]);
15    states["s3"]->addTransition('w', states["s4"]);
16    states["s2"]->addTransition('r', states["s5"]);
17    states["s5"]->addTransition('w', states["s4"]);
```

Listing 5.6: jedem Zustand werden seine Übergänge zu anderen Zuständen zugeordnet

Wenn der endliche Automat sich in einem der definierten Zustand befindet, erwartet er die Symbole die in den Übergängen vorher definiert wurden. Damit auch mit vorher nicht angegeben Symbolen umgegangen werden kann, kann jedem Zustand eine defaultTransition zugeordnet werden. Diese Transition wird an das Ende der Transitionen angehängt. Wenn keine andere Transition auf das Eingabesymbol reagiert, werden die defaultTransition genutzt um den Automaten in einen definierten Zustand gehen zu lassen. Wird diese Transition nicht angegeben, dann bleibt der Automat in seinem aktuellen Zustand.

In dem Beispiel wurde ein Zustand “s6“ eingeführt, der als ein Fehlerzustand gelten kann, denn aus ihm führen keine Zustände heraus.

```
18    states["s2"]->defaultTransition(states["s6"]);
19    states["s3"]->defaultTransition(states["s6"]);
20    states["s5"]->defaultTransition(states["s6"]);
```

Listing 5.7: Jede Transition bekommt einen Übergang in einen Fehlerzustand

Als letzter Schritt bei der Einrichtung des Automaten wird genau ein Startzustand gesetzt. In diesen Zustand geht der Automat beim Initialisieren und Zurücksetzen des Automaten. Dieser Zustand entspricht dem q_0 aus der Definition. Da es sich um einen deterministischen Automaten handelt, kann es nur einen Startzustand geben. Andernfalls wäre es ein nichtdeterministischer Automat, da bereits der Startzustand nicht eindeutig bestimmt wäre.

```
20      setStartState(states["s1"]);
```

Listing 5.8: Der Startzustand wird festgelegt

Aus der Definition eines deterministischen endlichen Automaten geht hervor, dass es eine endliche Menge von Endzuständen gibt. Diese ist nicht direkt in die FSMPP Bibliothek integriert. Das Programm, das die Bibliothek nutzt, kann mit der Funktion getCurrentStateId abfragen, in welchem Zustand sich der Automat befindet. Diese Information kann dann vom Programm genutzt werden um zu entscheiden, ob es sich beim aktuellen Zustand um einen Endzustand handelt.

Als Abschluss dieser Arbeit werde ich eine Zusammenfassung und einen Ausblick auf Folgearbeiten geben.

Kapitel 6

Schlussfolgerungen und Ausblick

In diesem Kapitel wird eine Zusammenfassung, Schlussfolgerungen und ein Ausblick auf zukünftige Arbeiten gegeben.

6.1 Zusammenfassung

In dieser Arbeit habe ich die Leistungsfähigkeit der Optimierungen durch Datasieving untersucht. Dabei war feststellbar, dass ein komplexer Zusammenhang zwischen den verschiedenen Parametern der Anfrage besteht und der dabei erreichten Leistung. Das Optimierungspotential lag bei sehr kleinen Nutzdatenblockgrößen, 1KiByte, bei bis zu 80MiB/s. Bei größeren Nutzdatenblockgrößen, 1000KiByte, waren es noch bis zu 25MiB/s gegenüber den Standardeinstellungen. Daher war es nötig nicht nur eine allgemeingültige Heuristik anzunehmen, sondern ein Plugin zu entwickeln, dass eine komplexe Wissensbasis nutzten kann, um damit Optimierungen durchzuführen. Das Plugin wurde entwickelt und es ist in der Lage mithilfe des TraceReaders Aktivitäten Dateien abzuspielen und Optimierungsvorschläge auf der Konsole auszugeben. Da der Parameterraum umfangreicher als zunächst angenommen ist, konnten nicht alle Plugins umgesetzt werden.

6.2 Schlussfolgerung

SIOX macht es möglich, mithilfe von Plugins moderne Dateisysteme optimal zu nutzen. Dabei spielt das Dateisystem selbst keine Rolle, da es möglich ist für beliebige Dateisysteme Instrumentierungen zu erstellen und dann mit vorhandenen Plugins zu nutzen. Die Steigerung der Effizienz mit der Dateien vom Dateisystem gelesen werden ist, wie

in Kapitel 3 zu erkennen war, zum Teil sehr hoch. Zudem ist es einfach möglich Plugins für den online und offline Teil von SIOX zu erstellen und auf den jeweils anderen Teil anzupassen, wie in Kapitel 4 dargestellt wurde.

6.3 Ausblick

In dieser Arbeit wurden die Untersuchungen ohne SIOX durchgeführt. Da SIOX auch einen gewissen Overhead erzeugt, ist es nötig, die in dieser Arbeit gefunden idealisierten Ergebnisse mit SIOX zu prüfen und auszuwerten.

Für weiterführende Arbeiten könnten folgende Punkte betrachtet werden.

- Plugin für den online Teil in SIOX schaffen, mit dem die Optimierungen live genutzt werden
- den Benchmark mit mehreren Prozessen starten und die Ergebnisse analysieren
- entsprechende Plugins mit dem Benchmark auf Leistungsfähigkeit testen und somit SIOX overhead in die Analyse mitaufnehmen
- offline Plugin entwickeln, dass Wissen für die Plugins aus der Datenbank der Aktivitäten automatisch generiert.

Literaturverzeichnis

[1] Thomas Ludwig. Hochleistungsrechnen. Vorlesungsskript. URL `http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2011_2012/hr-1112.pdf`.

[2] Julian Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, and Johann Weging. The siox architecture – coupling automatic monitoring and optimization of parallel i/o. In *Supercomputing*, number 8488 in Lecture Notes in Computer Science, Berlin, Heidelberg, 2014 – to-appear. Springer.

[3] Michaela Zimmer, Julian Kunkel, and Thomas Ludwig. Towards self-optimization in hpc i/o. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, number 7905 in Lecture Notes in Computer Science, pages 422–434, Berlin, Heidelberg, 06 2013. Springer. ISBN 978-3-642-38749-4. doi: http://dx.doi.org/10.1007/978-3-642-38750-0_32.

[4] URL `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.pe.v1r3.pe500.doc%2Fam107_ifsetv.htm`.

[5] Ewing Lusk William Gropp Robert Latham Rajeev Thakur, Robert Ross. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Mathematics and Computer Science Division, ARGONNE NATIONAL LABORATORY 9700 South Cass Avenue Argonne, IL 60439, April 2010. URL `http://www.mcs.anl.gov/research/projects/romio/doc/users-guide.pdf`.

Eidesstattliche Erklärung

Ich versichere, dass ich die Bachelorarbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.
Hamburg, den 7. April 2014

Nachname:	Schmidtke
Vorname:	Daniel
Matrikelnummer:	6250282
Unterschrift:	_____