Hochschule Esslingen
University of Applied Sciences

Forschungsprojekt

# Tracing-Tool zur Analyse von IO auf HPC-Systemen

im Studiengang Angewandte Informatik der Fakultät Informationstechnik

Wintersemester 2018/2019

Philipp Köster und Johannes Maisch

**Datum:** 6. März 2019
**Betreuer:** Prof. Dr.-Ing. Rainer Keller

# Ehrenwörtliche Erklärung

Hiermit versicheren wir, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 6. März 2019 _____

Unterschriften

# Abstract

Ziel dieses Berichtes ist es über den Stand des Forschungsprojektes „Tracing-Tool zur Analyse von IO auf HPC-Systemen“ zu informieren. Im Rahmen des Forschungsprojektes wird ein Werkzeug zur Protokollierung von File-IO geschaffen. Die protokollierten Daten können dann von einem ebenfalls im Forschungsprojekt erstellten Werkzeug analysiert werden.

Der Bericht zeigt auf, wie nach einer kurzen Betrachtung bestehender Werkzeuge zur Analyse von File-IO ein eigenes Konzept entworfen wurde. Dabei liegt der Fokus auf dem Wrappen von File-IO-Funktionsaufrufen, da dies der im Projekt bereits in Arbeit befindliche Aufgabenbereich ist. Für das Wrappen und Protokollieren werden die möglichen Ansätze sowie unterschiedliche Architekturen aufgezeigt und gegeneinander abgewogen. Abschließend wird der aktuelle Stand zusammen mit Beispielen für die Wrapper dargestellt und die geplanten künftigen Schritte dargelegt.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Das High-Performance-Computing (HPC) beschäftigt sich mit dem Hochleistungsrechnen. Im Bereich des High-Performance-Computing geht es darum, dass Computer mit möglichst grosser Leistung und m̈oglichst vielen parallelen Prozessen operieren. Dabei ist es unerlässlich, dass zum einen CPU-Berechnungen als auch Speicher-Zugriffe möglichst schnell sind.

## 1.1 BWHPC

Die Arbeiten in diesem Projekt werden am Hochleistungscluster BWHPC durchgeführt. Das BWHPC ist ein Hochleistungscluster des Landes Baden Württemberg, welcher an der Universität Karlsruhe steht und für Forschungszwecke eingesetzt wird.

## 1.2 Mooresches Gesetz

Die Leistung von Prozessoren wird immer schneller. Die Gewschwindigkeit des Wachstums kann durch das sog. Mooresche Gesetz beschrieben werden. Die Definition dieses Gesetzes ist im folgenden gegeben.

> Die Anzahl an Transistoren, die in einen integrierten Schaltkreis festgelegter Grösse passen, verdoppelt sich etwa alle zwei Jahre. [16]

Das Mooresche Gesetz sagt im Umkehrschluss also aus, dass sich die Prozessorleistung etwa alle zwei Jahre verdoppelt. Dieses Wachstum ist in Abbildung 1.1 für Intelprozessoren beispielhaft dargestellt.

Auffallend in Abbildung 1.1 ist, dass die Frequenz der einzelnen Kerne seit einigen Jahren nicht mehr zunimmt. Dies bedeutet, dass das Wachstum nicht mehr durch das Steigern von Leistung, sondern durch das Parallelisieren von CPU-Kernen bestimmt wird. Gerade im Bereich des Hochleistungsrechnen ergibt sich daraus, dass viele Anwendungen parallel ausgeführt werden.

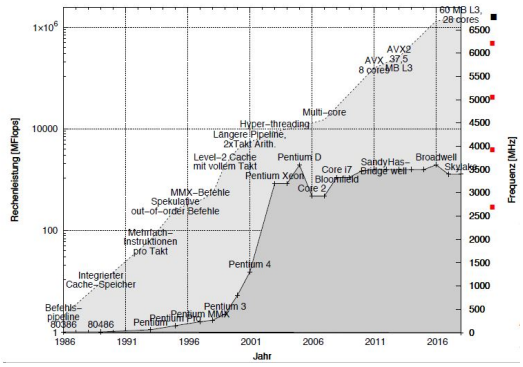


**Abb. 1.1:** Mooresches Gesetz

## 1.3 Speicherzugriffe

Mit Speicherzugriffen kann ein Prozessor Daten aus einem Speicher holen und auch in ihn schreiben. Dabei kann grob zwischen Registern, Caches, Hauptspeicher und Festplatte unterschieden werden. Wohingegen der Zugriff auf Register ohne grosse Latenzen möglich ist, ist der Zugriff auf andere Speicher deutlich langsamer. Die Zugriffszeiten sind in Tabelle 1.1 vergleichend dargestellt.

| **Speicher** | **Zugriffszeit** |
|---|---|
| CPU zu L3-Cache | 10 mal schneller |
| CPU zu Hauptspeicher | 100 bis 1000 mal schneller |
| CPU zu Festplatte | 1000 bis 1000000 mal schneller |

**Tab. 1.1:** Speicherzugriffe

Aufgrund dieser Geschwindigkeitsunterschiede ist es notigwendig den Zugriff auf Speicher möglichst effizient zu gestalten, da es sonst zu erheblichen Engpässen im Programmabläufen kommen kann.

## 1.4 Dateisysteme

Der Zugriff auf Speicher geschieht über Dateisysteme. Ein Dateisystem ist notwendig, damit Programme in einem Betriebssystem auf Dateien zugreifen können, welche auf der Festplatte liegen. Bei Dateisystemen kann im Wesentlichen zwischen seriellen und parallelen Dateisystemen unterschieden werden. Die Unterscheidung hierbei liegt darin, wie parallel ausgeführte Programme auf Dateien zugreifen.

### 1.4.1 Serielles IO

Beim seriellen IO läuft der komplette IO über einen Masterprozess. Dies bedeutet, dass Programme nicht gleichzeitig auf eine Datei zugreifen können. Dies ist in Abbildung 1.2 ersichtlich.

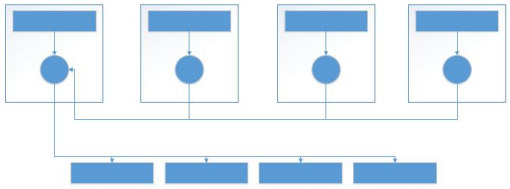

**Abb. 1.2:** Serial IO [10]

In Abbildung 1.2 ist zu erkennen, dass es zu starken Engpässen kommen kann, wenn mehrere Programme gleichzeitig auf eine Datei zugreifen wollen. Diese Art des IO stellt daher auf kleinen Desktop-Computern mit nur wenigen parallelen Programmen kein Problem dar, im Bereich des High-Performance-Computing mit vielen parallelen Programmen sollte aber auf andere Methoden zurückgegriffen werden.[10]

### 1.4.2 Paralleles IO

Im Gegensatz zum seriellen IO ist es beim parallelen IO möglich, dass mehrere Prozesse zeitgleich auf eine Datei zugreifen können. Dies ist in 1.3 dargestellt. Darin wird ersichtlich, dass der IO nicht mehr über einen Masterprozess läuft, sondern, dass die einzelnen Prozesse ihren IO unabhängig voneinander durchführen. Der Vorteil hierbei liegt darin, dass die

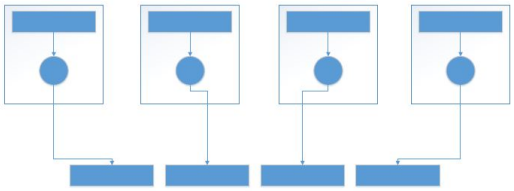

**Abb. 1.3:** Parallel IO [10]

einzelnen Prozesse parallel auf Dateien zugreifen können. Gerade im Bereich des High-Performance-Computing mit sehr vielen parallelen Prozessen stellt dies einen wichtigen Vorteil dar.[10]

### 1.4.3 POSIX IO

POSIX-IO ist der IO-Part des POSIX-Standards. Der POSIX-Standard ist ein Standard fÜr die Kommunikation von Prozessen mit dem Betriebssystem. POSIX-IO beschreibt dabei verschiedene Funktionen, über welche Programme in einem POSIX-Betriebssystem auf Speicher zugreifen können. POSIX-IO ist eine Form des seriellen IO, was im Bereich des High-Performance-Computings zu Problemen führen kann, welche im Folgenden kurz erläutert werden sollen.

#### Metadaten

Dateien auf einem POSIX-Dateisystem müssen eine Vielzahl an Metadaten besitzen. Sollen Informationen über eine Datei angezeigt werden, müssen diese Metadaten ausgelesen werden. Auf HPC-Systemen kann dies einen Nachteil darstellen, da das Auslesen der Metadaten von vielen Dateien mitunter sehr lange dauert[13].Darüber hinaus sind die Metadaten in POSIX-IO sehr unflexibel, da sämtliche Dateien alle Metadaten besitzen müssen und nicht bspw. Metadaten für alle Dateien in einem Ordner gelten können.

#### File-Descriptoren

Bevor eine Datei in POSIX gelesen werden kann, muss diese zunächst geöffnet werden, um einen File-Descriptor zu erhalten. Mit diesem File-Descriptor wird sichergestellt, dass immer nur ein Prozess auf eine Datei zugreifen kann. Wird eine Datei wieder geschlossen, wird der File-Descriptor wieder freigegeben.
Der Nachteil von File-Descriptoren kommt zum Tragen, wenn viele Prozesse gleichzeitig auf ein Dateisystem zugreifen wollen. Dann muss das Betriebssystem sehr viele File-Descriptoren parallel verwalten, wodurch bspw. das Öffnen einer Datei immer langsamer wird, je mehr Prozesse parallel auf das Dateisystem zugreifen. In Abbildung 1.4 ist dabei ersichtlich, dass das Öffnen einer Datei linear langsamer wird, je mehr parallele Prozesse auf das Dateisystem zugreifen.

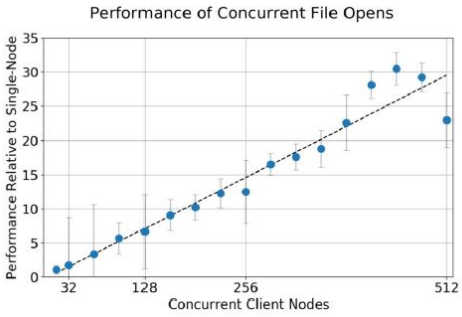


**Abb. 1.4:** Öffnen einer Datei bei POSIX-IO [14]

### Konsistenz

Das Schreiben in eine Datei muss in POSIX konsistent sein. Dies bedeutet, dass das Schreiben die Ausführung einer Applikation so lange blockiert, bis sichergestellt ist, dass ein Lesezugriff das neu geschriebene sieht. Dies hat wiederum den Nachteil, dass im AusfÃ¼hren von Applikationen durch das Schreiben in eine Datei starke Latenzzeiten entstehen. Im HPC-Bereich mit vielen parallelen Applikationen stellt dies ein grosses Problem dar, wenn viele Prozesse gleichzeitig in Dateien schreiben wollen. [14]

## 1.4.4 MPI-IO

MPIO-IO ist der IO-Part des Message Passing Interface (MPI). MPIO-IO ist dabei eine sog. Middleware, welche i.d.R. nicht direkt von Anwendungen sondern nur indirekt durch höhere Schichten genutzt wird. Es definiert somit einen Standard für parallele IO-Operationen in einer MPI-Applikation. Im Gegensatz zu POSIX-IO ist der Zugriff auf Dateien hierbei nicht Bytestrom- sondern elementorientiert. Der Aufbau einer Datei in MPI-IO ist in Abbildung 1.5 und in Abbildung 1.6 ersichtlich. Eine Datei wird dabei in sog. Fliessen aufgeteilt. Auf diese Fliessen kann über einen Dateityp zugegriffen werden. Ein Dateityp beschreibt ein Muster an Fliessen, welches sich über Teile der Datei oder über die ganze Datei wiederholt. Ein solches Muster ist in Abbildung 1.5 dargestellt. Jede Fliesse im Muster besteht wiederum aus einem elementaren Typ. Der elementare Typ ist der Datentyp über welchen auf die Datei zugegriffen werden kann. Ein Prozess, der auf die Datei über diesen Dateityp zugreift kann somit auf alle Fliessen zugreifen, welche in diesem Muster liegen.

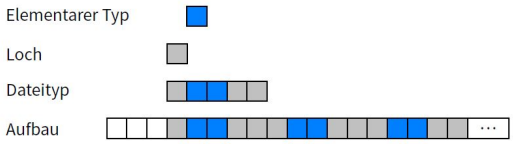


**Abb. 1.5:** Dateityp bei MPI-IO [12]

In Abbildung 1.6 ist der Aufbau einer Datei aus Sicht von Prozessen dargestellt. Dies wird auch als Dateisicht bezeichnet. Jeder Prozess greift damit über einen anderen Dateityp auf die Datei zu, wodurch mehrere Prozesse gleichzeitig auf die Datei zugreifen können.

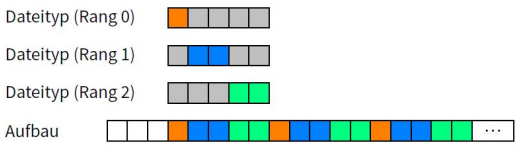


**Abb. 1.6:** Dateisicht bei MPI-IO [12]

MPI-IO stellt im High-Performance-Bereich eine gute Alternative zu POSIX dar, da damit mehrere Prozesse zeitgleich auf eine Datei zugreifen können. Die populärste Implementierung von MPI-IO ist ROMIO. MPI-IO bildet darüber hinaus die Basis vieler IO-Systeme wie bspw. HDF.[11][12]

# 2 Ziel des Projekts

Ziel des Projekts ist es, eine Software zu entwickeln, die den IO von Anwendungen analysiert. Die Software soll sich dabei zwischen das zu analysierende Programm und das Betriebssystem schalten und sämtlichen IO abfangen. Der IO des Programms soll anschliessend gespeichert und graphisch aufbereitet werden. Die Software soll dabei interaktiv sein. Das bedeutet, es soll mit der Software möglich sein, gezielt nach IO-Engpässen in einer Anwendung zu suchen.

Im ersten Schritt sollen dabei bestehende Softwarelösungen evaluiert werden. Im zweiten Schritt geht es dann darum, eine eigene Software zu entwickeln, welche die oben genannten Forderungen erfüllt. Anforderungen an die Software sind sowohl eine portable Entwicklung, als auch Thread-Sicherheit.

# 3 Stand der Technik

Im Rahmen der Forschungsarbeit erfolgte zunächst eine Marktrecherche, welche Softwarelösungen zum Tracing von IO bereits auf dem Markt sind. Darüber hinaus wurden diese Lösungen bezüglich ihrer Funktionalität evaluiert. Wichtigstes Kriterium bei der Marktrecherche ist, dass es sowohl möglich ist POSIX-IO zu untersuchen, als auch MPI-IO. Darüber hinaus soll die Analyse zur Laufzeit ohne Recompilieren des Codes möglich sein.

## 3.1 Darshan

Darshan ist ein Programm zur Analyse von POSIX-IO und MPI-IO. Mit Darshan kann ein PDF-Report des IO von Programmen erstellt werden. Bei dynamisch gelinkten Programmen ist dies zur Laufzeit möglich, bei statisch gelinkten Programmen ausschliesslich beim Bau des Programms.
Darshan besteht aus zwei Programmen. Mit Darshan-Runtime werden die Informationen über den IO eines Programms gesammelt und gespeichert, mit Darshan-Util werden diese aufbereitet und dargestellt.

### 3.1.1 Funktionsweise

Das Sammeln von Informationen zur Laufzeit von Programmen geschieht über die Systemvariable LD_PRELOAD. Mit dieser ist es möglich Features in ein Prorgamm einzuschleusen. Beim Laden von Shared Libraries wird dabei zunächst nicht die eigentliche Bibliothek geladen, sondern diese, welche unter LD_PRELOAD angegeben wurde. Damit wird dann die Darshan-Bibliothek geladen, welche die IO-Befehle speichert und diese anschliessend an die eigentlichen Bibliotheken weitergibt. Die Funktionsweise von Darshan für dynamisch gelinkte Programme ist in Abbildung 3.1 dargestellt. Die Bibliothek libdarshan.so wird dabei vom zu untersuchenden Programm über LD_PRELOAD geladen. Diese speichert alle MPI-IO- und POSIX-IO-Befehle in Log-Dateien. Diese Log-Dateien können anschliessend mit Darshan-Util ausgewertet werden. Dabei wird ein PDF-Report kreiert in welchem in Diagrammen u.a. dargestellt wird, wieviele IO-Operationen jeweils durchgeführt wurden und welche Datenmengen dabei verarbeitet wurden.

Das Analysieren von statisch gelinkten Programmen funktioniert ähnlich wie bei Vampir-Trace mit Compiler-Wrappern. Diese werden beim Bau anstatt der eigentlichen Compiler

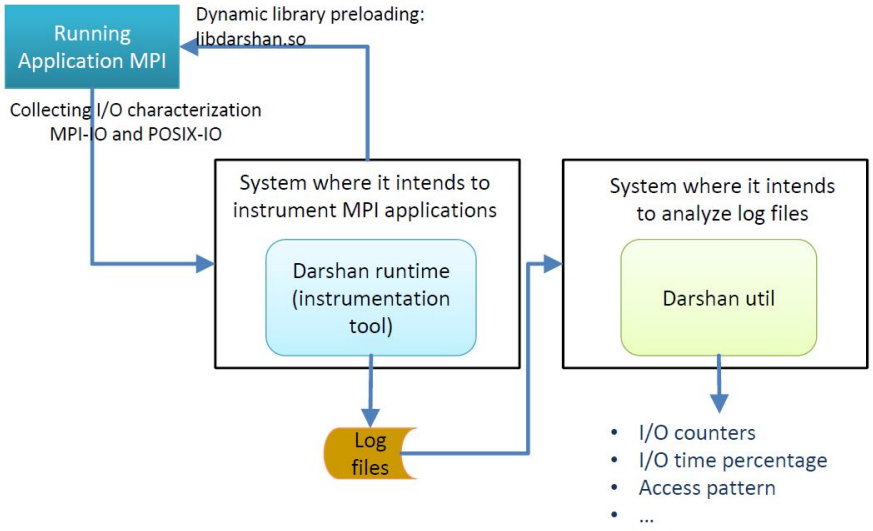


**Abb. 3.1:** Darshan Aufbau [15]

aufgerufen. Die Wrapper werten das Programm dann aus und schreiben die Log-Dateien. Anschliessend werden die eigentlichen Compiler aufgerufen. [3][1]

### 3.1.2 Fazit

Darshan ist ein hervorragendes Programm zur Analyse des IO von dynamisch gelinkten Programmen. Der Nachteil liegt dabei jedoch darin, dass der graphische Output nicht interaktiv ist. Es wird zwar ein PDF-Report kreiert, es ist jedoch nicht möglich interaktiv gezielt nach Schwachstellen im Programm zu suchen. Darüber hinaus können statisch gelinkte Programme mit Darshan nicht zur Laufzeit ohne erneuten Bau untersucht werden, was ebenfalls einen gravierenden Nachteil darstellt.

## 3.2 VampirTrace

VampirTrace ist ein Programm, welches von der Universität Dresden zur ursprünglich Analyse von MPI-Programmen entwickelt wurde. Mittlerweile ist es ein Tool-Set zur Analyse von parallelen Programmen im HPC-Bereich. Mit VampirTrace kann sowohl MPI-IO als auch POSIX-IO untersucht werden. Für die Analyse von Programmen ist es notwendig, diese mithilfe von VampirTrace-Compiler-Wrappern neu zu bauen. Im Makefile müssen dabei die Compiler durch die Compiler-Wrapper von VampirTrace ersetzt werden. Diese rufen dann wiederum die eigentlichen Compiler auf. Die gebauten Programme können anschliessend zur Laufzeit mit VampirTrace analysiert werden. Eine Untersuchung zur Laufzeit ohne neuen Bau ist nicht ohne weiteres möglich.

Die gewonnenen Daten werden von VampirTrace in einer Log-Datei im Open-Trace-Format (OTF) gespeichert. Diese Log-Dateien können anschliessend mit Tools, die den Umgang mit OTF beherrschen, visualisiert werden. Am besten eignet sich hierzu das Tool Vampir, welches ebenfalls von der Universität Dresden zu diesem Zweck entwickelt wurde. [2]

## 3.3 Ludalo

Mit Ludalo ist es möglich Lustre-Metadaten-Operationen zu analysieren. [9]

## 3.4 TAU

Tau ist eine Software, entwickelt von der University of Oregeon, zur Analyse von parallelen Applikationen. Eine IO-Analyse ist dabei sowohl für POSIX-IO, als auch für MPI-IO möglich. Die von TAU generierten Daten können im OTF-Format gespeichert und anschliessend mit Vampir visualisiert werden. Die Analyse erfolgt entweder durch das Recompilieren des Codes oder durch das Laden einer Bibliothek mit LD_PRELOAD. Hinsichtlich der Funktionalität wurde TAU in diesem Projekt nicht weiter evaluiert. [17][18]

## 3.5 Fazit

Keines der untersuchten Programme enthält alle Features, welche in diesem Projekt gewünscht sind. Darshan trifft die Anforderungen jedoch am ehesten. Damit kann sowohl MPI-IO als auch POSIX-IO analysiert werden. Allerdings ist dabei keine interaktive Bedienung möglich. Dasselbe ist bei VampirTrace auch der Fall. Damit kann zwar ebenfalls MPI-IO und POSIX-IO analysiert werden, jedoch ist auch hierbei keine interaktive Bedienung vorhanden. Aus diesem Grund soll in diesem Projekt eine Software entwickelt werden, welche die Features von Darshan und VampirTrace mit einer interaktiven Bedienung vereint.

# 4 Entwicklung einer eigenen Software

Zur Entwicklung einer eigenen Lösung für die Protokollierung und anschließenden Analyse von File-IO sind mehrere Schritte notwendig. Zunächst muss festgelegt werden, welche Daten ausgewertet werden sollen und wie diese erfasst werden können. Nachgelagert folgt dann die Konzeption des gesamten Systems bis hin zur visuellen Aufbereitung der Analyseergebnisse. Mit diesen Schritten befassen sich die folgenden Abschnitte.

## 4.1 File-IO-Konstellationen

Im Folgenden werden unterschiedliche Konstellationen von Dateizugriffen betrachtet, die bei einer Analyse unterschieden werden müssen. Dementsprechend bilden diese Konstellationen auch die Grundlage für die zu protokollierenden Daten. Es müssen ausreichend Daten protokolliert werden um diese Konstellationen erkennen zu können.

### 4.1.1 Konsistenz und Synchronisation

Alle folgende Konstellationen beziehen sich auf die Laufzeit eines untersuchten Programms. Zugriffe auf Dateien vor Programmstart und nach Programmende werden nicht betrachtet. Eine Unterscheidung in Threads und Prozesse wird bei der Beschreibung der einzelnen Konstellationen nicht vorgenommen, da beide Varianten sich bezüglich der Synchronisation gleich verhalten. Vereinfachend steht der Begriff Prozess daher für Thread oder Prozess.

#### Prozess 0 schreibt einmalig in eine Datei

Ein Prozess schreibt einmalig in eine Datei. Die Datei wird durch keinen anderen Prozess beschrieben und durch keinen Prozess gelesen.

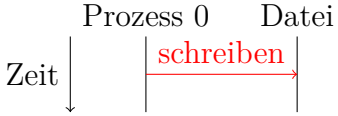


**Abb. 4.1:** Prozess 0 schreibt einmalig in eine Datei

Um diese Konstellation zu erkennen, muss für alle Dateizugriffe des Programms die Datei und die Art des Zugriffs protokolliert werden. Dadurch ist es möglich zu überprüfen, ob nur einmalig schreibend auf eine Datei zugegriffen wird.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen

Bei dieser Konstellation ist keine Synchronisation während der Laufzeit des Programms erforderlich. IO kann gefahrlos optimiert werden. Lediglich vor Programmende muss eventuell geprüft werden, ob der Schreibvorgang abgeschlossen ist, damit nachfolgenden Programmen die Daten zur Verfügung stehen. Hierfür muss auch ein Schließen der Datei (POSIX close) protokolliert werden.

- Art des Zugriffs: schließen

#### Prozess 0 schreibt wiederholt in eine Datei

Ein Prozess schreibt wiederholt in eine Datei. Die Datei wird durch keinen anderen Prozess beschrieben und durch keinen Prozess gelesen.

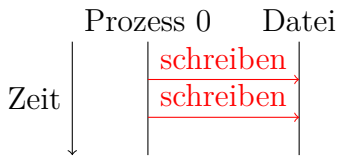


**Abb. 4.2:** Prozess 0 schreibt wiederholt in eine Datei

Um diese Konstellation zu erkennen sind die gleichen Daten wie bei einem einmaligen Schreiben in eine Datei notwendig (siehe Abschnitt 4.1.1, Seite 11). Neben der Datei selbst und der Art des Zugriffs ist hier allerdings auch noch die Zeit von Interesse. Für Optimierungen des Zugriffs ist die Reihenfolge der Zugriffe entscheidend. Nur wenn die Reihenfolge protokolliert wird, kann bei einer Optimierung das ursprüngliche Ergebnis sichergestellt werden. Hierfür ist auch die genaue Position der geschriebenen Bytes in der Datei wichtig. Nur mit dieser Information kann ein Überschreiben in der Datei oder ein sequenzielles Anhängen an ein Dateiende erkannt und bei einer Optimierung beachtet werden. Um Möglichkeiten zur Optimierung zu erkennen, ist zusätzlich die Dauer eines einzelnen Schreibvorgangs notwendig. Über diese Information kann geprüft werden, ob ein nachfolgender Schreibvorgang auf den vorhergehenden warten muss. Hierfür muss zu den Schreibvorgängen auch noch das eigentliche Öffnen (POSIX open) protokolliert werden. Dies betrifft allerdings die Art des Zugriffs und entspricht daher den oben bereits erwähnten Informationen.

- Datei: Pfad und Dateiname

- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)

Bei dieser Konstellation ist eine Synchronisation zwischen den einzelnen Schreibvorgängen notwendig, sofern diese sich gegenseitig überschreiben oder ein vorhergehender Schreibzugriff das Dateiende verschiebt und der aktuelle Vorgang an dieses anknüpft. Im zweiten Fall kann eventuell auf eine Synchronisation über Blockieren nachfolgender Schreibzugriffe verzichtet werden, wenn die Bytelänge und Anzahl der vorhergehenden Schreibvorgänge bekannt ist.

### Prozess 0 liest einmalig aus einer Datei

Ein Prozess liest einmalig aus einer Datei. Die Datei wird durch keinen anderen Prozess gelesen und durch keinen Prozess beschrieben.

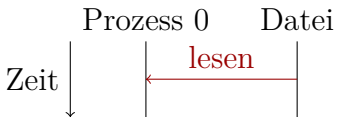


**Abb. 4.3:** Prozess 0 liest einmalig aus einer Datei

Um diese Konstellation zu erkennen, sind die gleichen Daten wie bei einem einmaligen Schreiben in eine Datei notwendig (siehe Abschnitt 4.1.1, Seite 11). Das Schließen der Datei ist dabei ebenfalls nur zur Sicherung des Zugriffs durch nachfolgend arbeitende Programme notwendig.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen

Bei dieser Konstellation ist keine Synchronisation während der Laufzeit des Programms erforderlich. IO kann gefahrlos optimiert werden.

### Prozess 0 liest wiederholt aus einer Datei

Ein Prozess liest wiederholt aus einer Datei. Die Datei wird durch keinen anderen Prozess gelesen und durch keinen Prozess beschrieben.

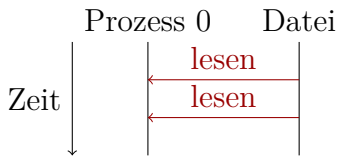


**Abb. 4.4:** Prozess 0 liest wiederholt aus einer Datei

Diese Konstellation verhält sich analog zum einmaligen Lesen aus einer Datei (siehe Abschnitt 4.1.1, Seite 13), da das mehrfache Lesen einer sich nicht verändernden Datei weder beim Erkennen der Konstellation noch zur Synchronisation zusätzliche Anforderungen stellt.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen

#### Prozess 0 liest und schreibt eine Datei

Ein Prozess liest und schreibt wiederholt eine Datei. Die Datei wird durch keinen anderen Prozess gelesen und durch keinen anderen Prozess beschrieben.

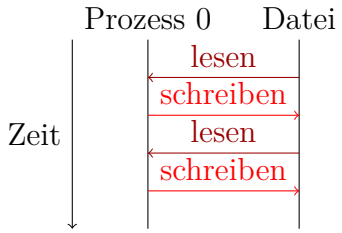


**Abb. 4.5:** Prozess 0 liest und schreibt eine Datei

Um zwischen Lesen und Schreiben unterscheiden zu können, sind mindestens die Informationen zum einmaligen Lesen (siehe Abschnitt 4.1.1, Seite 13) und zum einmaligen Schreiben (siehe Abschnitt 4.1.1, Seite 11) notwendig. Um konkurrierende und sich gegenseitig blockierende Schreibvorgänge zu identifizieren sind zudem die gleichen Daten wie beim wiederholten Schreiben nötig (siehe Abschnitt 4.1.1, Seite 12).

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)

Bei dieser Konstellation ist eine Synchronisation sowohl zwischen den einzelnen Schreibvorgängen als auch zwischen Schreibvorgängen und darauf folgenden Lesevorgängen notwendig. Hier muss also zusätzlich zu den im Abschnitt über wiederholtes Schreiben (siehe Abschnitt 4.1.1, Seite 12) genannten Prüfungen noch überprüft werden, ob ein lesender Zugriff auf zuvor durch Schreibvorgänge veränderte Bytes erfolgt.

### Prozess 0 und Prozess 1 schreiben in eine Datei

Mehrere Prozesse schreiben wiederholt in die gleiche Datei. Die Datei wird durch keinen Prozess gelesen.

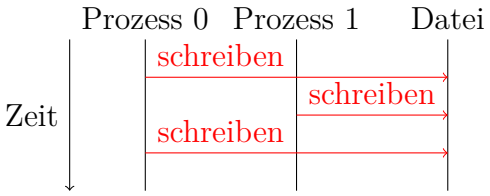


**Abb. 4.6:** Prozess 0 und Prozess 1 schreiben in eine Datei

Neben den Daten zum Erkennen mehrerer Schreibvorgänge (siehe Abschnitt 4.1.1, Seite 12) ist noch eine Information über den jeweiligen Prozess notwendig.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)
- Prozess/Thread: Prozess-ID und Thread-Nummer

Wie beim wiederholten Schreiben durch einen Prozess (siehe Abschnitt 4.1.1, Seite 12) ist auch in dieser Konstellation eine Synchronisation nötig.

### Prozess 0 und Prozess 1 lesen aus einer Datei

Mehrere Prozesse lesen wiederholt aus der gleichen Datei. Die Datei wird durch keinen Prozess beschrieben.

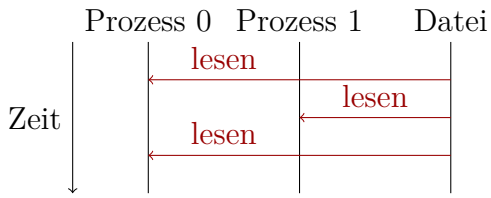


**Abb. 4.7:** Prozess 0 und Prozess 1 lesen aus einer Datei

Neben den Daten zum Erkennen mehrerer Lesevorgäng (siehe Abschnitt 4.1.1, Seite 13) ist noch eine Information über den jeweiligen Prozess notwendig.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Prozess/Thread: Prozess-ID und Thread-Nummer

#### Prozess 0 und Prozess 1 lesen und schreiben eine Datei

Mehrere Prozesse lesen und schreiben wiederholt eine Datei.

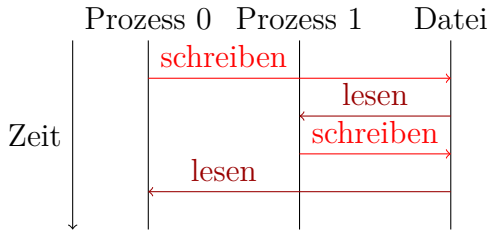


**Abb. 4.8:** Prozess 0 und Prozess 1 lesen aus einer Datei

Diese Konstellation stellt eine Kombination aller vorhergehenden Konstellationen dar. Dementsprechend werden alle Daten der einzelnen Konstellationen benötigt.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)
- Prozess/Thread: Prozess-ID und Thread-Nummer

Hier ist eine Synchronisation zwischen den Schreibvorgängen aller beteiligter Prozesse notwendig. Zudem müssen Lesevorgänge nach Schreibvorgängen gegebenenfalls auch synchronisiert erfolgen (wenn ein zuvor geschriebener Dateiinhalt ausgelesen wird).

### 4.1.2 Metadaten

Wenn Metadaten von einem überwachten Programm ausgelesen werden, so muss dies protokolliert werden. Falls kein derartiger Zugriff auf Metadaten protokolliert wurde, kann möglicherweise auf die Erstellung und Aktualisierung von Metadaten verzichtet werden. Dementsprechend könnten Methoden und Dateisysteme ohne Metadaten genutzt werden. Da Metadaten über POSIX-Methoden nicht explizit sondern implizit über jeden ändernden Zugriff auf eine Datei geschrieben werden, muss vor einem Verzicht auf Metadaten allerdings geklärt werden, ob neben dem überwachten Prozess noch nachgelagerte Zugriffe auf die Metadaten stattfinden. Dies kann nicht automatisch geschehen. Daher kann in einem solchen Fall nur eine Empfehlung durch eine Analyse erstellt werden.

## 4.2 Architektur

Wichtig für die Architektur ist neben dem Blick auf das gesamte System auch die Entscheidung für ein geeignetes Datenformat zur Protokollierung und die genaue Gestaltung der Wrapper. Mit einem Wrapper ist in diesem Zusammenhang das Abfangen eines File-IO-Funktionsaufrufes gemeint. Hierfür wird ein Funktionsaufruf an eine andere Funktion (den Wrapper) delegiert und erst von dort aus die ursprünglich gewünschte Funktion aufgerufen. Somit besteht die Möglichkeit im Wrapper zusätzliche Funktionalität auszuführen. Dies kann beispielsweise die Protokollierung von Informationen über den ursprünglichen Funktionsaufruf sein.

### 4.2.1 Datenformat

Für die im Abschnitt zu Konsistenz und Synchronisation (siehe Abschnitt 4.1.1, Seite 11) aufgeführten Konstellationen müssen durch die Wrapper unterschiedliche Informationen protokolliert werden. Aus diesen Unterschieden ergeben sich folgende Satzarten:

| Art des Zugriffs | Prozess-ID | Thread-Nr. | Pfad und Datei | Startzeit | Endzeit | Position | Länge |
|---|---|---|---|---|---|---|---|
| öffnen | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| schließen | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| lesen | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| schreiben | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

✓ Datum wird benötigt
✗ Datum wird nicht benötigt

**Tab. 4.1:** Übersicht Satzarten

Es werden neben den oben aufgeführten noch weitere Daten benötigt (z.B. Optionen beim Öffnen einer Datei oder MPI-spezifische Parameter).

Um die benötigten Daten nach dem Protokollieren in eine Datei durch beliebige Anwendungen nutzen zu können (siehe Abschnitt 4.2.3, Seite 24) ist ein binäres Format nicht ausreichend. Es wird ein Datenformat benötigt, welches die einzelnen Werte eines Datensatzes selbst beschreibt. Somit setzt das Nutzen der Daten nicht die genaue Kenntnis um Bytelängen und Kodierung der Werte voraus, wie dies bei einem Binärformat der Fall wäre. Als Format kommt somit beispielsweise XML in Frage. Da beim Schreiben der Protokolldatei jedoch auch die Performance wichtig ist, wird ein Format mit möglichst geringem zusätzlichen Aufwand benötigt. Hier erfordert XML durch die Wiederholung der Beschreibung in öffnenden und schließenden Tags zu viele Bytes beim Schreiben der Datei. Dies wirkt sich negativ auf die Dateigröße und die zum Schreiben benötigte Zeit aus. Unter den Formaten mit hoher Verbreitung erfüllt JSON am ehesten die Anforderungen. Die in JSON-Dateien enthaltenen Werte sind als Key-Value-Paare abgelegt und enthalten somit immer einen beschreibenden Text als Key. Zusätzlich sind die eigentlichen Werte im Value-Teil immer als String-Repräsentation kodiert. Weiterhin ist der Daten-Overhead durch die Nutzung weniger Sonderzeichen anstelle von beispielsweise Tags in XML möglichst gering.

### 4.2.2 Wrapper

Wesentlich für die Funktion der Protokollierung ist ein effizientes und vollständiges Abfangen des File-IOs. Dies geschieht über Wrapper, welche die jeweiligen Funktionsaufrufe abfangen und anschließend alle benötigten Daten protokollieren, um danach die ursprünglich gewünschte Funktion aufzurufen. Im Folgenden wird das Wrappen von Funktionsaufrufen von C-Bibliotheken genauer erläutert. Dies geschieht anhand von POSIX-IO und der glibc. Zudem wird die Möglichkeit von direkten Aufrufen des Linux-Kernels ohne vorgeschaltete

C-Bibliothek erwähnt, aber aufgrund der geringen Relevanz für dieses Projekt nicht weiter erläutert. Anschließend werden drei mögliche Architekturvarianten für die Wrapper näher erläutert und gegeneinander abgewogen.

Eine Liste der vom Linux-Kernel angebotenen Systemfunktionen kann der Dokumentation in den „man-Pages“ entnommen werden[8]. Hier finden sich auch die von POSIX definierten Funktionen, welche ebenfalls in der glibc-Bibliothek als C-Funktionen zur Verfügung stehen.

Bei der Architektur der Wrapper liegt der Fokus insbesondere auf der Performance. Da der zu untersuchende File-IO einen Engpass in den überwachten Anwendungen darstellen kann, muss das Protokollieren des IOs mit möglichst geringem Aufwand an Speicher und Laufzeit erfolgen.

### POSIX über glibc

Die POSIX Implementierung in Linux stellt Systemfunktionen entsprechend der einzelnen POSIX-Funktionen bereit. Diese werden meist nicht direkt, sondern über die entsprechende c-Bibliothek glibc aufgerufen[6]. Für die Fälle, in denen die Systemfunktionen über glibc aufgerufen werden, können Wrapper für die einzelnen Funktionen bereitgestellt werden.

Dabei ist zu beachten, dass einige Funktionen innerhalb der glibc wiederum andere Funktionen aufrufen. So ruft beispielsweise „printf“ zunächst „puts“ auf. In „puts“ wird wiederum „write“ aufgerufen. Da mit den folgend beschriebenen Mitteln nur Aufrufe von außen an die glibc-Bibliothek gewrappt werden können, Aufrufe innerhalb von glibc aber nicht, müssen für das Wrappen aller „write“-Aufrufe auch „puts“ und „printf“ gewrappt werden.

Die folgenden Vorgehensweisen setzen voraus, dass die glibc genutzt wird. Dies ist abhängig von der genauen Implementierung des jeweiligen Programms. Es gibt Programme, welche direkt die Systemfunktionen aufrufen ohne die glibc zu verwenden.

Dynamische gelinkt: Unter Linux existiert die Umgebungsvariable „LD_PRELOAD“. Diese kann zur Angabe eines Pfades zu einer shared library genutzt werden. Die entsprechend über diesen Pfad angegebene Bibliothek wird dann vor allen anderen Bibliotheken geladen. Werden in dieser Bibliothek Funktionen der glibc-Bibliothek (C-Interface zu Systemfunktionen unter Linux)[8] überschrieben, so werden anstelle der glibc-Funktionen die überschriebenen Funktionen ausgeführt.

Um innerhalb der überschriebenen Funktionen die ursprünglich gerufene Funktion aus der glibc-Bibliothek aufzurufen, kann nicht direkt der Funktionsname genutzt werden, da dies zu einem Namenskonflikt mit der überschriebenen Funktion führt. Anstelle eines Aufrufs über den Namen kann allerdings mit der Funktion dlsym[4] die Adresse der gewünschten Funktion ermittelt werden. Über diese Adresse kann dann die Funktion in glibc ausgeführt werden.

Die Funktion dlsym muss mit der Konstante „RTLD_NEXT“ (findet erste Routine innerhalb der geladenen Module) aufgerufen werden. Aus Performancegründen muss dabei über einen Init Hook (Linker -init) sichergestellt sein, dass einmalig nach dem Laden von glibc der jeweilige Funktionspointer ermittelt wird.

Ein Beispiel für einen dynamischen Wrapper kann dem Anhang entnommen werden (siehe Abschnitt A.2, Seite 30).

Statisch gelinkt: Werden Funktionen nicht zur Laufzeit dynamisch ermittelt, sondern sind statisch fest eingebunden, so kann ein Wrapper nur zum Zeitpunkt des Linkens eingebunden werden. Hierfür kann im GNU Linker die Option „ld –wrap=symbol“ [7] genutzt werden. Über den gcc kann die Option „-Wl“genutzt werden, damit intern der Linker mit „ld –wrap“ aufgerufen wird.

Ein Beispiel für einen statischen Wrapper (siehe Abschnitt A.3, Seite 32) und einen dazu passenden Aufruf des Linkers (siehe Abschnitt A.4, Seite 33) kann dem Anhang entnommen werden.

#### POSIX über Kernel Entry Point

Im Projekt vorerst nicht relevant, da üblicherweise über glibc und nicht direkt über System Calls mittels Kernel Entry Point gearbeitet wird. Sollte der Projektfokus entsprechend erweitert werden, so sind die Wrapper (siehe Abschnitt 4.2.2, Seite 18) nicht ausreichend. Ansätze für mögliche Lösungen können dem Linux-Programm ptrace oder Programmen wie Plash, Systrace, Subterfugue, Chrome sandbox und Pink trace entnommen werden. Ein möglicher Ansatz ist „system call interposition“. Dieser Ansatz macht aber eventuell bei jedem System Call einen zusätzlichen Kontextwechsel notwendig und wirkt sich damit stark auf die Performance aus.

#### Thread Local Storage

Über Thread Local Storage (TLS) sicherstellen, dass beim Schreiben in den Speicher keine Synchronisation notwendig ist und somit auch keine Wartezeiten anfallen. Im TLS für jeden Thread einer Anwendung einen Buffer zum Schreiben reservieren. Sobald ein Buffer voll ist, die enthaltenen Daten in eine eigene Datei schreiben. Dabei die Prozess-ID und die Thread-Nummer im Dateinamen vermerken. Auf diese Weise ist auch beim Schreiben in die Datei keine Synchronisation notwendig.

Um den TLS beim Start eines Threads zu reservieren und ihn vor dem Ende des jeweiligen Threads abschließend in eine Datei zu übernehmen, muss das Starten und das Beenden eines Threads erkannt werden. Zusätzlich zum Reservieren und Leeren des TLS muss auch das Öffnen und Schließen der jeweiligen Datei abhängig von der Laufzeit des jeweiligen Threads erfolgen (siehe hierzu Abbildung 4.9).

Da die Funktionalität zum Überprüfen, ob der Buffer noch ausreichend leeren Platz enthält, und zum Leeren des Buffers in eine Datei, in jedem Wrapper einer POSIX-/MPI-IO-Funktion benötigt wird, muss sie in separate Funktionen ausgelagert werden. Dabei kann die Funktion zum Prüfen des Buffers die Funktion zum Leeren intern nutzen/aufrufen. Somit muss in den Wrappern nur eine Funktion genutzt werden und das Leeren steht dennoch als separate Funktion für das Beenden eines Threads zur Verfügung.

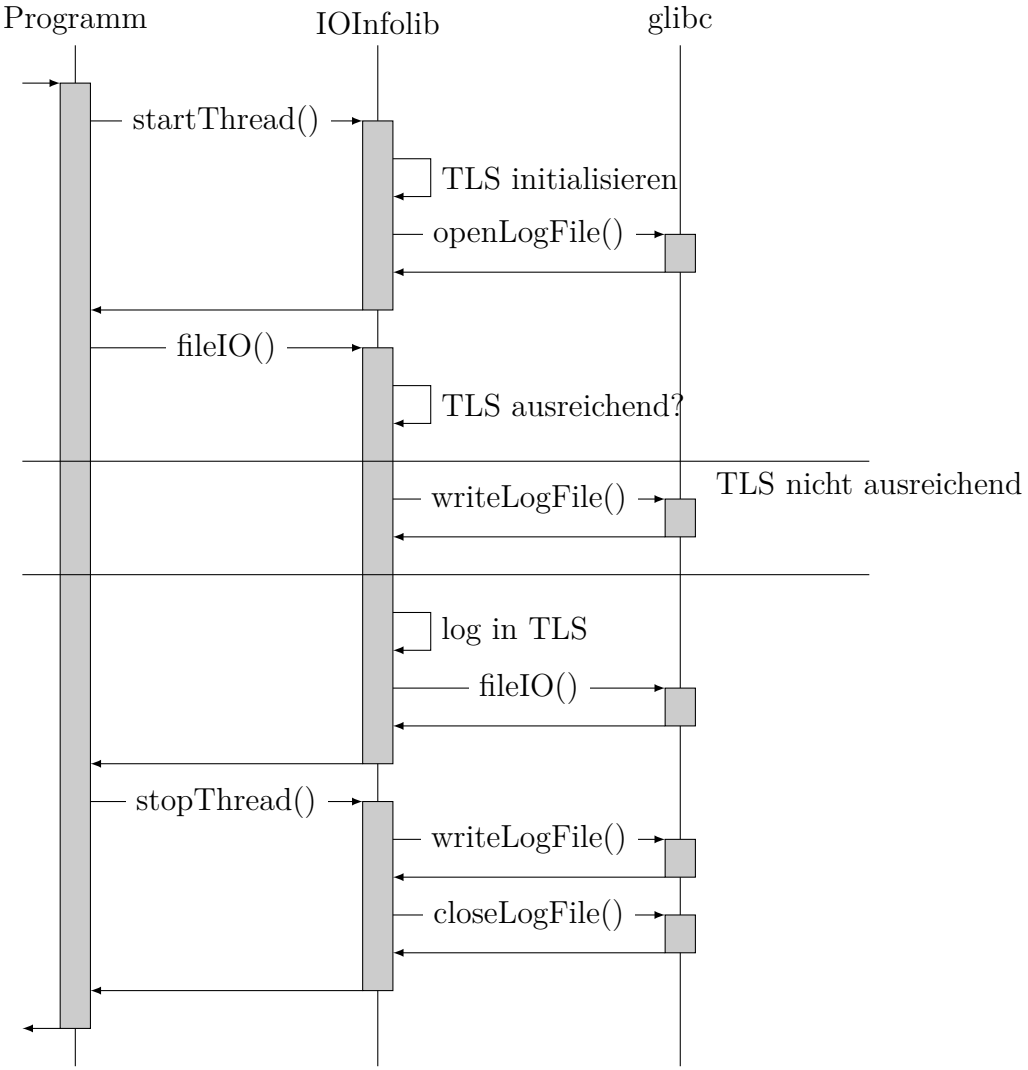


**Abb. 4.9:** Sequenzdiagramm Wrapper mit TLS

Je nach Art der Parallelisierung unter Linux kann das Starten und das Stoppen eines nebenläufigen Vorgangs unterschiedlich abgefangen werden. Bei einem Vorgehen über fork() und exit() werden separate Prozesse (heavy-weight process) gestartet. Geschieht dies über die entsprechenden Funktionen in der glibc, so kann es über Wrapper abgefangen werden. Zusätzlich müssen noch alle nicht mittels fork() sondern über einen Befehl zum Ausführen einer Datei gestarteten Prozesse abgefangen werden. Dies betrifft die Funktionen execl(),

execlp(), execv() und execvp(). Wird anstelle von fork() die Funktion clone() genutzt, so wird kein Prozess, sondern ein Thread (light-weight process) gestartet. Daher muss diese Funktion ebenfalls abgefangen werden. In diesem Fall gibt es keinen exit() für die einzelnen Threads. Stattdessen kann ein waitpid() mit der Prozess-ID des Threads genutzt werden, um das Ende des Threads zu erkennen. In diesem Fall ist der Speicher des Threads allerdings bereits freigegeben und nicht mehr sicher nutzbar. Das finale Schreiben des Buffers in eine Datei kann also auf diesem Weg nicht ermöglicht werden. Das gleiche gilt für exit_group(). Über diese Funktion werden mehrere laufende Prozesse/Threads in einer Prozess-Gruppe gemeinsam beendet. Auch hierbei kann nicht sichergestellt werden, ob der Speicher eines Threads bereits freigegeben wurde.

Wurde alternativ über die POSIX-Threads in der glibc parallelisiert, so kann über das Makro pthread_cleanup_push() eine weitere Funktion zum finalen Cleanup übergeben werden. Da dieses Makro allerdings immer mit einem weiteren Makro (pthread_cleanup_pop()) innerhalb der gleichen umschließenden Klammern kombiniert werden muss, lässt sich dieser Ansatz bei einer event-orientierten Vorgehensweise nicht nutzen. Zudem funktioniert dieses Vorgehen nur bei Threads, die mit den entsprechenden Funktionen der glibc erstellt wurden. Somit lassen sich beispielsweise über openMP parallelisierte Programme so nicht instrumentieren.

Bei dieser Architektur muss also zwischen unterschiedlichen Parallelisierungsarten unterschieden werden. Zudem gibt es Vorgehensweisen, die nicht über die glibc-Funktionen gehen. So nutzen manche Programmme direkt die Funktionen des Kernels und reduziert auf diese Weise den Overhead der Parallelisierung auf ein Minimum. Leider gibt es bei diesen Programmen somit keine Funktionen in der glibc oder einer anderen Bibliothek, die durch einen Wrapper abgefangen werden können.

Ein weiterer Nachteil dieser Architektur ist der erhöhte Speicherverbrauch. Um Verzögerungen durch dynamische Reservierung von Speicher (alloc() bzw. malloc()) zu vermeiden soll der benötigte Speicher einmalig reserviert werden. Beim dynamischen Reservieren müsste zum einen das Betriebssystem prüfen, ob und wenn ja wo der angefragte Speicherplatz verfügbar ist. Zum Anderen müssten Strategien für den Fall, dass nicht ausreichend Speicherplatz zur Laufzeit verfügbar ist, implementiert werden. Es findet daher kein nachträgliches Vergrößern des Speichers statt. Dies deckt sich mit den Anforderungen an TLS. Hier wird einmalig zum Start eines Threads der benötigte Speicher reserviert. Daraus ergibt sich allerdings das Problem, dass für jeden Thread bereits zum Start des Threads ein großer Buffer reserviert werden muss. Zu diesem Zeitpunkt steht noch nicht fest, wie viel File-IO der jeweilige Thread tatsächlich ausführt. Im Extremfall führt ein Thread überhaupt keinen File-IO aus und benötigt daher eigentlich keinen Buffer im TLS. Dies kann der jeweilige Wrapper zwar während der Ausführung des Threads analysieren und bei Beenden des Threads final feststellen, da die Reservierung des Speichers aber schon zum Start des Threads erfolgen muss, wird unnötigerweise Speicherplatz blockiert. Wird aufgrund dieses Umstandes ein kleiner Buffer gewählt, so wirkt sich dies negativ auf alle Threads aus, die tatsächlich viel File-IO ausführen. Bei diesen ist der kleine Buffer ständig voll und muss vor erneutem Protokollieren des IOs erst durch Schreiben in eine

Datei geleert werden. Dies bremst den Thread aus und steht somit im Widerspruch zu den Anforderungen an die Wrapper-Architektur.

### Zentraler Buffer

Ein einziger zentraler Buffer umgeht die Probleme bei der Nutzung von TLS (siehe Abschnitt 4.2.2, Seite 20). Es ist keine Unterscheidung in verschiedene Arten der Parallelisierung und kein unnötiges Reservieren von Speicher notwendig (siehe hierzu Abbildung 4.10). Allerdings erfordert das Schreiben in den zentralen Buffer eine Synchronisation zwischen den einzelnen Threads einer Anwendung. Hierdurch entsteht ein Engpass, durch den sich Threads gegenseitig ausbremsen können. Daher ist diese Architekturvariante zwar leicht zu implementieren, verfehlt aber die Vorgabe möglichst performant zu sein.

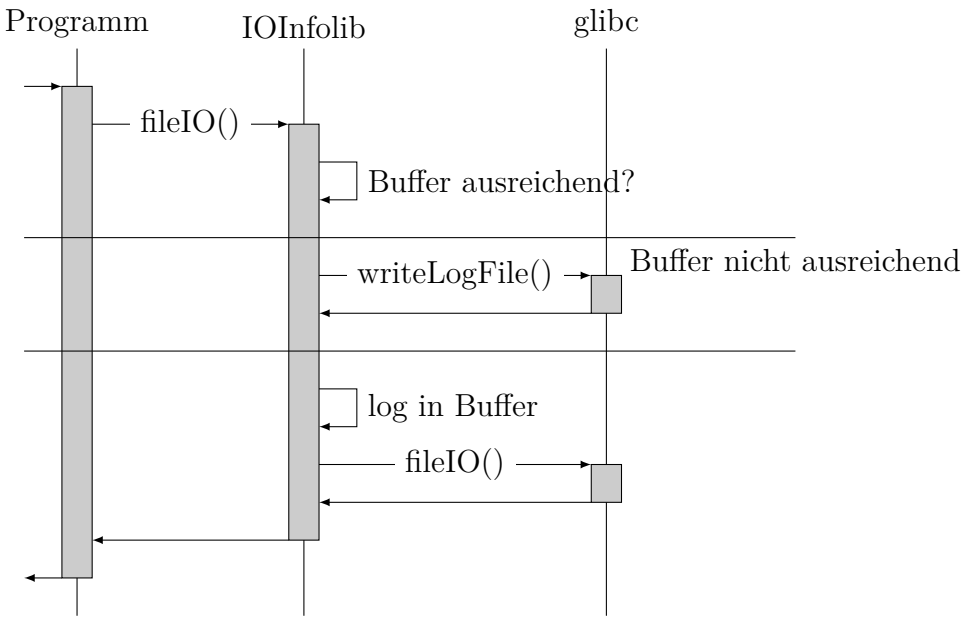


**Abb. 4.10:** Sequenzdiagramm Wrapper mit zentralem Buffer

### Lock-free Bag

Um die Vorteile der Architekturvarianten Thread Local Storage (siehe Abschnitt 4.2.2, Seite 20) und zentraler Buffer (siehe Abschnitt 4.2.2, Seite 23) zu kombinieren und gleichzeitig die jeweiligen Nachteile zu vermeiden kann ein lock-free Bag genutzt werden. Dabei wird in einen zentralen Bag geschrieben (siehe hierzu Abbildung 4.11). Es ist also nicht notwendig das Verwalten von Threads zu überwachen. Zur Synchronisation der Zugriffe nutzt der Bag atomare Instruktionen. Hierbei wird lediglich ein Pointer in den Bag atomar inkrementiert. Das Schreiben in den durch den Pointer definierten Speicherbereich erfolgt nach dem Inkrementieren unsynchronisiert. Somit wird die Synchronisation auf eine einzige Instruktion beschränkt. Sie ist somit auf das absolut mögliche Minimum reduziert.

Hierfür muss die zugrunde liegende Architektur des Prozessors jedoch entsprechende atomare Instruktionen zur Inkrementierung eines Pointers anbieten. Diese Architekturvariante funktioniert daher nicht mit jedem Prozessor.

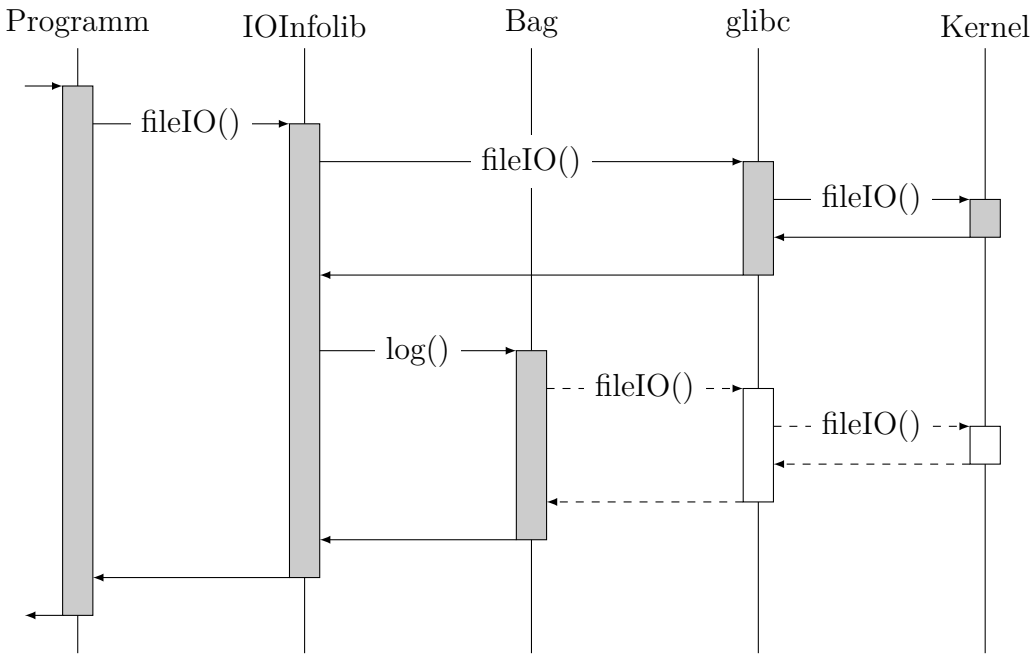


**Abb. 4.11:** Sequenzdiagramm Wrapper mit lock-free Bag

### 4.2.3 System

Die libiotrace schaltet sich zwischen den Prozess und die glibc. Dadurch kann jeder Aufruf einer Funktion für File-IO an die glibc abgefangen und protokolliert werden. Die Protokollierung erfolgt in einem JSON-File. Jeder mittels libiotrace überwachte Prozess führt somit pro Ausführung zu einer Datei mit Daten im JSON-Format (siehe hierzu Abbildung 4.12). Das Ablegen der Rohdaten in Dateien hat mehrere Gründe. Zum einen wird der überwachte Prozess nicht noch zusätzlich durch eine Analyse der Daten ausgebremst. Zum anderen stehen die Daten für die Analyse durch verschiedene Anwendungen zur Verfügung. Es können neue Anwendungen auf Basis der Dateien erstellt werden, ohne dass hierfür die libiotrace angepasst oder aus der neuen Anwendung mit der libiotrace kommuniziert werden muss.

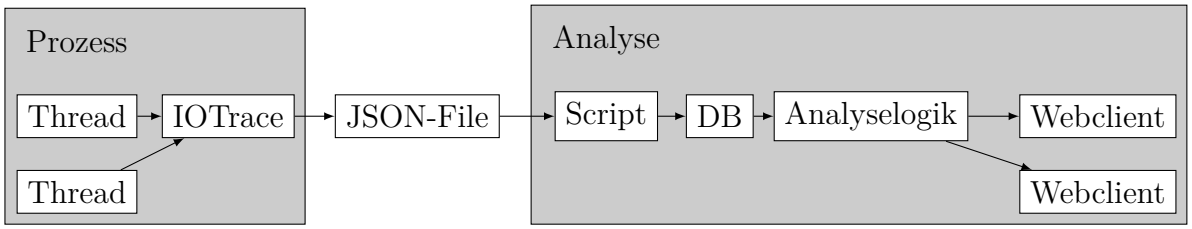


**Abb. 4.12:** Systemarchitektur

Ein überwachter Prozess kann, abhängig vom jeweiligen File-IO, sehr viele Protokolldaten erzeugen. Um in diesen die zur Analyse und Visualisierung interessanten Konstellationen zu finden, müssen sie performant durchsucht werden können. Sollen zudem die Protokolle mehrerer Prozesse in Abhängigkeit zueinander durchsucht werden, so wird die Datenmenge und somit das Problem größer. Für eine performante Verarbeitung der Protokolldaten muss daher eine schnelle Suche in den Daten möglich sein. Hierfür werden die einzelnen Protokolldateien über ein Script in eine Datenbank importiert. Die Datenbank stellt über die Generierung von Indizes performante Suchen zur Verfügung.

Um verschiedenen Frontends über die gleiche Datenbank versorgen zu können, ohne hierbei für jedes Frontend erneut grundlegende Analysefunktionen zu implementieren, werden Analysen in eine eigene Komponente ausgelagert. Diese Komponente regelt die Datenbankverbindung und bietet Schnittstellen für die Frontends an. Über die Schnittstelle können interaktiv sowohl die Rohdaten, als auch Analyseergebnisse abgefragt werden.

Die Visualisierung der Daten und Analysen erfolgt in den Frontends. Hier wird zunächst ein Webfrontend vorgesehen. Dieses ermöglicht eine Visualisierung unabhängig vom genutzten Client.

## 4.3 Umsetzung

Vor der Implementierung von Analysen und der Visualisierung der Ergebnisse müssen die Wrapper zum Protokollieren der benötigten Daten vorhanden sein. Daher werden zunächst die Wrapper entwickelt.

### 4.3.1 Wrapper

Die benötigten Wrapper und alle zum Protokollieren in eine Datei benötigten Funktionen werden in einer Bibliothek zusammengefasst bereitgestellt. Diese Bibliothek heißt libiotrace. Sie wird in einer Variante für dynamisch gelinkte Programme und einer Variante zum statischen Linken bereitgestellt.

Die Umsetzung der libiotrace erfolgt aus Performancegründen in der Programmiersprache C. Dabei wird CMake als Buildtool genutzt. Somit kann die Entwicklung der Bibliothek

möglichst portabel gehalten werden. Für das Schreiben in die Protokolldatei müssen parallele Prozesse unterstützt werden. Dies macht eine threadsafe Implementierung der Bibliothek notwendig. Aus Performancegründen ist dabei eine Synchronisation über Locks unerwünscht. Stattdessen erfolgt die Umsetzung über hochperformante und möglichst lockfreie Datenstrukturen.

Die Entwicklung erfolgt in einer frei verfügbaren und kostenlosen IDE mit Unterstützung für die Sprache C. Neben Netbeans kommt somit auch Eclipse in Frage. Aufgrund des Einsatzes von CMake kann die IDE beliebig gewechselt werden. Sofern die IDE CMake-Projekte nicht direkt unterstützt erfolgt die Entwicklung zwar in der IDE, der Buildvorgang wird dann aber außerhalb der IDE durchgeführt.

Für die gemeinsame Entwicklung und die Verwaltung unterschiedlicher Entwicklungsstände wird ein Repository-System benötigt. Dieses muss wie die IDE frei verfügbar und kostenlos sein. Entsprechend der aktuellen Verbreitung und Popularität wurde hierfür GIT ausgewählt. Das Projektrepository ist unter `https://github.com/hpcraink/fsprj2` erreichbar.

Um Unit-Tests zu ermöglichen wird das Projekt CUnit genutzt [5].

# 5 Aktueller Stand

Alle nötigen Vorarbeiten für die Erstellung der Wrapper sind abgeschlossen (siehe Abschnitt 4.3.1, Seite 25). Die Entwicklungsumgebung und ein CMake-Projekt sind eingerichtet. Zudem sind die grundlegenden architektonischen Entscheidungen getroffen (siehe Abschnitt 4.2.2, Seite 18). Hierbei wurde auch die geplante Gesamtarchitektur für das ganze System inklusive Analyse und Visualisierung beachtet (siehe Abschnitt 4.2.3, Seite 24).

Durch erste kleine Testprogramme wurde die Funktionsweise der Wrapper erfolgreich getestet. Hierfür wurde ein Programm erstellt, welches in mehreren parallelen Threads die gleichen Funktionen aus der glibc aufruft (siehe Abschnitt A.1, Seite 30). Dieses Programm wurde dann mit einem dynamischne Wrapper (siehe Abschnitt A.2, Seite 30) und einem statischen Wrapper (siehe Abschnitt A.3, Seite 32) erfolgreich überwacht. Für den dynamischen Wrapper war hierbei ein Laden des Wrappers über die Umgebungsvariable „LD_PRELOAD“ (siehe Abschnitt 4.2.2, Seite 19) und für den statischen Wrapper ein Linken über Optionen des Linkers (siehe Abschnitt 4.2.2, Seite 20) notwendig. Ein Beispiel für das Linken ist im Anhang enthalten (siehe Abschnitt A.4, Seite 33).

Mit der Implementierung der ersten Wrapper wurde bereits begonnen. Diese wird jetzt fortgesetzt. Dabei liegt der Fokus zunächst auf POSIX-IO und MPI-IO.

# 6 Zusammenfassung und Ausblick

Nach einer kurzen Evaluation von verfügbaren Werkzeugen zur Analyse von File-IO wurden diese als unzureichend für den angestrebten Zweck beurteilt. Lediglich Darshan erfüllt die grundlegenden Anforderungen. Mit Darshan können POSIX-IO und MPI-IO in eine Log-Datei protokolliert werden. Diese Datei kann dann durch ein Analyseprogramm augewertet werden. Darshan unterstützt jedoch nur POSIX-IO und MPI-IO. Andere Arten von Dateizugriffen werden nicht berücksichtigt. Diese sind allerdings im HPC-Bereich von entscheidender Bedeutung. So erfordert eine hochparallelisierte Kommunikation zu Dateien zum Beispiel ein Dateisystem wie Lustre. Dieses bringt entsprechend eigene File-IO-Funktionen mit sich, welche über Darshan nicht protokolliert werden können. Zudem schreibt Darshan die Protokolldateien in einem Binärformat. Dieses ist entsprechend seiner Natur nicht leicht aus anderen Anwendungen heraus zu nutzen. Insbesondere bei der Verwendung von Programmiersprachen welche die binäre Kodierung vor dem Entwickler verbergen (wie zum Beispiel Java) stellt das Binärformat ein unerwünschtes Hindernis dar.

Nach der Evaluation wurde daher mit dem Entwurf einer eigenen Lösung begonnen. Dabei lag der Fokus im ersten Schritt auf den Wrappern. Deren Architektur und das Datenformat der Protokollierung wurden nach Gesichtspunkten der Performance und der Nutz- und Erweiterbarkeit gewählt. Dabei ergaben sich für das Abfangen der File-IO-Funktionsaufrufe ähnliche Ansätze wie in Darshan, während die Protokollierung andere Ansätze verfolgt. Neben der Architektur der Wrapper wurde für das gesamte System eine erste grobe Architektur festgelegt.

Welche Daten durch die ersten Wrapper für einfache Analysen gebraucht werden, wurde durch eine Betrachtung möglicher File-IO-Konstellatione ermittelt. Abhängig von den künftig im Projekt geplanten Analysen müssen jedoch weitere Daten protokolliert werden. Diese Erkenntnis wurde beim Entwurf der Wrapper und insbesondere bei der Wahl des Datenformats berücksichtigt.

Als Datenformat für die Protokollierung wurde JSON gewählt. Dieses Format ermöglicht eine Enfache Nutzung der protokollierten Daten mittels unterschiedlicher Programmiersprachen aus verschiedenen Plattformen heraus. Dabei hält sich der Overhead beim Protokollieren im Vergleich zu anderen Formaten in Grenze. JSON bietet zudem die Möglichkeit die Protokollierung einfach zu erweitern. Werden nachträglich durch Wrapper zusätzliche Daten benötigt, so können diese im JSON-Format hinzugefügt werden, ohne dass dies ein Auslesen der Datei beeinträchtigt.

Nach einem erfolgreichen Test des Konzepts der Wrapper wurde mittlerweile mit der Implementierung begonnen. Dabei liegt der Fokus zunächst auf POSIX-IO und MPI-IO. Im kommenden Semester ist geplant die Wrapper weitgehend fertig zu stellen. Nach erfolgreicher Implementierung der Wrapper wird mit der Umsetzung der restlichen Komponenten begonnen.

Da bei dem Entwurf einer eigenen Lösung für die Wrapper zum Protokollieren des File-IOs ähnliche Ansätze wie in Darshan gewählt wurden, ist eine weitergehende Analyse von Darshan bezüglich der Performance sinnvoll. Hier wird insbesondere ein Vergleich zwischen der im Rahmen dieses Projektes erstellten libiotrace und Darhans libdarshan.so angestrebt. Mit dieser Aufgabe kann begonnen werden, sobald die Wrapper einen mit Darshan vergleichbaren Funktionsumfang erreicht haben.

# A Anhang

## A.1 Testprogramm OpenMP

```c
#include <stdio.h>
#include <unistd.h>
#include "omp.h"


int main()
{
  #pragma omp parallel
  {
    printf("Hello Thread!\n");
    printf("Thread %d of %d\n", omp_get_thread_num(),
        omp_get_num_threads());
  }
  write(0, "Hello, Kernel!\n", 15);
  printf("Hello World!\n");

    return 0;
}
```

## A.2 dynamischer Wrapper mit zentraler Buffer

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdarg.h>
#include <pthread.h>

static void init() __attribute__((constructor));
static void cleanup() __attribute__((destructor));

/* Function pointers for glibc functions */
static ssize_t (*real_write)(int fd, const void *buf, size_t count) = NULL;
static int (*real_puts)(const char* str) = NULL;
static int (*real_printf)(const char *__restrict format, ...) = NULL;
static int (*real_vprintf)(const char *__restrict format, _G_va_list arg) =
    NULL;
```

```
18
19 /* Buffer */
20 #define BUFFER_SIZE 400
21 static char data_buffer[BUFFER_SIZE];
22 static char* endpos;
23 static char* pos;
24
25 /* Mutex */
26 static pthread_mutex_t lock;
27
28 void printData() {
29   real_printf(data_buffer);
30   pos = data_buffer;
31   *pos = '\0';
32 }
33
34 void writeData(char *data) {
35   int tmp_pos;
36   char print[100];
37   sprintf(print, "pid:%lu;pts:%lu;", getpid(), pthread_self());
38   strcat(print, data);
39   strcat(print, "\n");
40
41   /* write (synchronized) */
42   pthread_mutex_lock(&lock);
43   if (pos + strlen(print) > endpos) {
44     printData();
45   }
46   strcpy(pos, print);
47   pos += strlen(print);
48
49   pthread_mutex_unlock(&lock);
50 }
51
52 void cleanup() {
53   printData();
54
55   pthread_mutex_destroy(&lock);
56 }
57
58 static void init() {
59   real_write = dlsym(RTLD_NEXT, "write");
60   real_puts = dlsym(RTLD_NEXT, "puts");
61   real_printf = dlsym(RTLD_NEXT, "printf");
62   real_vprintf = dlsym(RTLD_NEXT, "vprintf");
63
64   endpos = data_buffer + BUFFER_SIZE - 1;
65   pos = data_buffer;
66
67   pthread_mutex_init(&lock, NULL);
68 }
69
```

```
70 ssize_t write(int fd, const void *buf, size_t count) {
71   char print[40];
72   sprintf(print, "write:chars#:%lu", count);
73   writeData(print);
74
75   return real_write(fd, buf, count);
76 }
77
78 int puts(const char* str) {
79   char print[40];
80   sprintf(print, "puts:chars#:%lu", strlen(str));
81   writeData(print);
82
83   return real_puts(str);
84 }
85
86 int printf(const char *__restrict format, ...) {
87   va_list args;
88   int retval;
89
90   char print[40];
91   sprintf(print, "printf:chars#:%lu", strlen(format));
92   writeData(print);
93
94   va_start(args, format);
95   /* use vprintf instead of printf because of the variable parameter-list */
96   retval = real_vprintf(format, args);
97   va_end(args);
98   return retval;
99 }
```

## A.3 statischer Wrapper

Die Funktino writeData ist im folgenden Beispiel nicht definiert. Sie kann analog zu einem dynamischen Wrapper implementiert werden (siehe Abschnitt A.2, Seite 30).

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  /* Function pointers for glibc functions */
6  ssize_t __real_write(int fd, const void *buf, size_t count);
7  int __real_puts(const char* str);
8
9  ssize_t __wrap_write (int fd, const void *buf, size_t count)
10 {
11   char print[40];
12   sprintf(print, "write:chars#:%lu", count);
13   writeData(print);
14
15     return __real_write(fd, buf, count);
```

```
16 }
17
18 int __wrap_puts (const char* str)
19 {
20   char print[40];
21   sprintf(print, "puts:chars#:%lu", strlen(str));
22   writeData(print);
23
24     return __real_puts(str);
25 }
```

## A.4 linken statischer Wrapper

Im folgenden Beispiel wurde wrap-link.o aus einem statischen Wrapper (siehe Abschnitt A.3, Seite 32) und test-link.o aus einem Testprogramm (siehe Abschnitt A.1, Seite 30) kompiliert.

```
1 gcc -Wl,-wrap=write -Wl,-wrap=puts bin/test-link.o bin/wrap-link.o -fopenmp
    -o bin/test-link-bin
```

# Literaturverzeichnis


[1] Darshan-util installation and usage, 19.01.2019.

[2] Vampirtrace 5.14.4: User manual, 2016.

[3] Darshan-runtime installation and usage, 22.01.2019.

[4] dlsym3. http://man7.org/linux/man-pages/man3/dlsym.3.html, o. J. [Online; Zugriff am 07.11.2018].

[5] gitlab. https://gitlab.com/cunity/cunit, o. J. [Online; Zugriff am 07.11.2018].

[6] intro2. http://man7.org/linux/man-pages/man2/intro.2.html, o. J. [Online; Zugriff am 07.11.2018].

[7] ld1. http://man7.org/linux/man-pages/man1/ld.1.html, o. J. [Online; Zugriff am 07.11.2018].

[8] syscalls2. http://man7.org/linux/man-pages/man2/syscalls.2.html, o. J. [Online; Zugriff am 07.11.2018].

[9] Holger Berger. Ludalo, 2014.

[10] John Cazes and Ritu Arora. Introduction to parallel i/o, 26.09.2013.

[11] Peter et.al. Corbett. Mpi-io: A parallel file i/o interface for mpi, 1995.

[12] Michael Kuhn. Mpi-io: Hochleistungs-ein-/ausgabe, 2016.

[13] Jeffrey B. Layton. Posix io must die!, 2010.

[14] Glenn Lockwood. What's so bad about posix i/o?, 2017.

[15] Sandra Mendez. Analyzing the high performance parallel i/o on lrz hpc systems, 23.06.2016.

[16] Robert Schanze. Mooresches gesetz: Defintion und ende von moore's law – einfach erklärt, 25.02.2016.

[17] Sameer Shende. Tau performance analysis, 2017.

[18] Sameer Shende, Allan D. Mallony, Wyatt Spear, and Karen Schuchardt. Characterizing i/o performance using the tau performance system, 2011.