Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Leistungsanalyse und -optimierung der Netzwerkkommunikation in dem HPC-Speichersystem JULEA unter Verwendung des OFI Frameworks

vorgelegt von

Arne Struck

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Bachelor of Science, Informatik
Matrikelnummer:	6326505
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Kira Duwe
Betreuer:	Dr. Michael Kuhn, Kira Duwe

Hamburg, 1. Dezember 2020

Abstract

Netzwerkkommunikation ist ein klassischer Flaschenhals der modernen Datenverarbeitung, im Speziellen im Bereich des High Performance Computing (HPC). In dieser Arbeit soll ein Beitrag zur Beantwortung der Frage geliefert werden, ob ein Performanzgewinn im Bereich der Netzwerkkommunikation durch direkte Integration einer spezialisierten Kommunikationslösung in ein bereits existierendes HPC-Programm möglich ist. Diese Frage soll auch unter dem Aspekt beleuchtet werden, dass keine speziellen Vorkenntnisse des untersuchten Frameworks beim Durchführenden vorliegen.
Als Referenzbeispiel dient das Open Fabrics Interface (OFI) als Vertreter der spezialisierten Netzwerkframeworks und JULEA exemplarisch als HPC-Anwendung. JULEA stellt ein flexibles Framework zur entfernten und lokalen Datenspeicherung dar und setzt für die Netzwerkkommunikation auf Berkley Sockets. OFI bietet eine Sammlung meist auf HPC spezialisierter Netzwerklösungen an. Libfabric ist hierbei eine Kernkomponente OFIs und stellt die API zur Zielanwendung dar.
Die Ergebnisse zeigen, dass eine direkte Integration libfabrics in JULEA unter Verwendung des libfabric sockets Providers in einer geringeren Datendurchsatzrate resultiert als die bisher verwendete direkte socket Implementation unter Verwendung von angebotenen Optimierungen. Besonders stark ist der Unterschied bei geringen Dateigrößen (im KB-Bereich) pro Übertragungsvorgang, während bei Dateigrößen im MB-Bereich sich die Durchsatzraten angleichen. Allerdings existiert auch im MB-Bereich immer noch ein Performanzvorsprung vor der bisherigen Variante.
Des weiteren wird demonstriert, dass auch eine direkte Integration bei komplexen Frameworks eine inhärente Komplexität und Fehleranfälligkeit mit sich bringt.
Es wird gezeigt, dass eine direkte Integration prinzipiell möglich ist, allerdings werden Performanzgewinne hierdurch nicht zwingend erreicht. Gründe hierfür können darin gefunden werden, dass Speziallösungen unter Umständen durch eine etablierte Lösung angebotene Optimierungen fehlen. Dies wiederum kann dazu führen, dass die Speziallösung perfomanztechnisch einer etablierten Lösung unterlegen sind. Des weiteren wird gezeigt, dass eine direkte Integration ohne Anpassung des Referenzprogrammes an die Vorgehensweisen des Netzwerkframeworks zu Performanzverlusten führen und somit leichte Anpassungen des Kommunikationsschemas notwendig sein kann.

Inhaltsverzeichnis

1. Einführung

Diese Arbeit beschäftigt sich mit dem Netzwerkaspekt des High Performance Computing (HPC) Bereiches, indem eine Standardlösung, durch Nutzung verschiedener dem Programmdesign entsprechende Angebote eines spezialisierten Netzwerkframeworks ersetzt werden.
Hierzu beschreibt die Arbeit die Nutzung und Integration des Open Fabric Interfaces (OFI) in das HPC-Speichersystem JULEA und die anschließende Messung und den Vergleich der Performance JULEAs anhand von Benchmarks. Somit soll gezeigt werden, ob ein Performanzgewinn durch das Austauschen von Netzwerktechnologien auf der Anwendungsschicht erreicht werden kann.

Da Netzwerkkommunikation sowohl aus dem HPC-Bereich als auch aus den restlichen heutigen Computersystemen nicht mehr wegzudenken ist, ist ein möglicher Performanzgewinn durch Nutzung neuerer Technologien von hoher Relevanz, auch wenn eine solche Integration nur einen kleinen Ausschnitt demonstrieren kann.
Die hohe Relevanz im HPC-Bereich rührt daher, dass es sich bei HPC-Systemen heutzutage in der Regel um Cluster-Systeme, also eine Gruppe gleichartiger über ein Netzwerk verbundener Rechner, handelt. Höhere Performanz eines Netzwerksystems resultiert in erhöhten Datendurchsatzraten, was wiederum genauere und daher datenintensivere Berechnungen ermöglicht. Insbesondere sind hier der direkte Zugriff und die Manipulation des Speichers eines Kommunikationspartners für größere Datenübertragungen interessant. Dies wird als Remote Memory Access (RMA) bezeichnet. Dabei wird eine spezielle Variante, bei welcher der Netzwerkhardware die Umgehung der eigenen CPU ermöglicht wird, als Remote Direct Memory Access (RDMA) bezeichnet.

Die Arbeit gliedert sich in drei Abschnitte. Zuerst wird in Kapitel 2 ein theoretischer Rahmen gesetzt, in dem die für die Zielanwendung relevanten Netzwerkebenen und Begriffe erläutert werden und sowohl eine Einführung in JULEA, als auch in das Open Fabrics Interface (OFI) geschieht, und die Beziehung zwischen OFI und seiner API libfabric erläutert wird. Im Mittelteil der Arbeit werden in Kapitel 3 die gewählten Integrationen der OFI Kommunikationsmethoden erläutert sowie in Kapitel 4 Benchmarkmethode und -sytem dargelegt und die Ergebnisse ebenjener Messungen präsentiert. Zum Ende geschieht in den Kapiteln 5 und 6 eine Aufbereitung der Ergebnisse, eine Erläuterung von aufgetretenen Problematiken sowie ein Ausblick auf mögliche aufbauende Arbeit.

Optimierung der Netzwerkkommunikation im HPC Bereich legt primär das Augenmerk auf die Topologie des Netzwerkes, da hier von einer Systemperspektive die größtmögliche Anzahl an Anwendungen optimiert werden kann. Ein Vergleich prominenter Topologien

ist beispielsweise in [Jai+16] zu finden.
Ansätze zur Performanzmessung einzelner Frameworks und deren Durchführung sind in deren Whitepaper wie [Gru+15] für OFI vorhanden.
Untersuchungen der einzelnen Frameworks an Referenzprogrammen sind eher selten, es existieren allerdings einige Paper, welche verschiedene libfabric Provider und UCX (Unified Communication X, [Sha+15]) in Verwendung für die Kommunikation der Kommunikationsmiddleware OpenSHMEM untersuchen. Bei OpenSHMEM handelt es sich um die Spezifikation einer API zur parallelen Programmierung im Partitioned Global Address Space, nebst Referenzinterpretation [PC20].
[Bha+19] vergleicht eine OpenSHMEM Variante, deren Kommunikationskanal auf libfabric unter Verwendung des sockets Providers beruht, mit einer alternativen OpenSHMEM Variante in Hinblick auf genutzte Bandbreite bei put und get Operationen. Im Vergleich zu einer Referenzimplementation des OpenSHMEM Netzwerkverkehrs konnte hier eine Steigerung der genutzten Bandbreite um bis zu 42% bei put und bis zu 11% bei get Operationen festgestellt werden.
In [Bak+16] wird eine Implementation von OpenSHMEM beschrieben, deren Kommunikation im Gegensatz zur Referenzimplementation auf UCX basiert und auf einem Cray XK System bezüglich Netzwerkperformanz getestet wurde. Im Vergleich zu der Referenzimplementation wird in den meisten Fällen eine Steigerung der Nachrichtenrate um bis zu 40% festgestellt.
[Gro+18] beschreibt eine OpenSHMEM Variante, welche den OpenSHMEM Kontext für die Kommunikation auf libfabric abbildet, allerdings den GNI Provider für Tests auf Aries Interconnect verwendet. Auch hier wird eine signifikante Verbesserung der Performanz in Relation zur Referenzvariante gemessen.
Anzumerken ist, dass es sich hierbei größtenteils um Neuimplementationen des Netzwerkverkehrs, also keine Integration in bestehende Programmteile, handelt.

2. Theoretische Einordnung

In diesem Kapitel wird ein theoretischer Überblick über die Theorie der Netzwerkkommunikation gegeben, erklärt worum es sich bei JULEA, OFI und libfabric handelt und eine kurze Einführung in den Aufbau von libfabric-Ressourcen gegeben.

2.1. Netzwerkkommunikation

Rechnernetze sind eine sowohl hardware-, als auch softwareseitige Verbindung mehrerer Recheneinheiten [TW12]. Recheneinheiten können in diesem Fall sowohl Computer jeglicher Art als auch ein untergeordnetes Rechnernetz sein. Eine solche Verbindung kann direkter oder indirekter Natur sein.
Rechnernetze ermöglichen den Austausch von Daten zwischen den einzelnen partizipierenden Rechnern und sind aus dem heutigen Leben nicht mehr weg zu denken. Dies kann man am einfachsten am größten und bekanntesten Rechnernetz sehen - dem Internet. Der heutige Alltag wäre ohne es ein anderer. Anwendungen in Rechnernetzen reichen von Kommunikation zwischen Anwendern bis hin zur Nutzung gemeinsamer Ressourcen. An der Basis solcher Rechnernetze stehen die Netzwerkkommunikation und die eigentliche Hardware als Voraussetzung.

2.1.1. Begriffe

Im Folgenden werden einige verwendete Computernetzwerke betreffende Begrifflichkeiten erläutert.

Client-Server-Modell Das verbreitetste Modell für die Kommunikation zwischen Rechnern ist das Client-Server-Modell. Ein oder mehrere Clients greifen hierbei sich auf dem Server befindenden Daten oder Dienste über ein Netzwerk zu. Somit können Clients mit den Daten oder Diensten, welche nicht auf ihrem System liegen, arbeiten.
Ein Server agiert hierbei als zentrale Anlaufstelle und kann Daten oder Dienste auf seinem System bereitstellen, welche das Client-System in ihrer Gesamtheit überlasten würden.
Der Kommunikationsablauf startet, wie in Abbildung 2.1 zu sehen, mit einer Anfrage des Client-Prozesses über das Netz an den Server-Prozess. Dieser nimmt die Anfrage entgegen, bearbeite sie und schickt dem Client-Prozess eine Antwort. Mit dem Erhalt der Antwort ist ein Kommunikationsvorgang abgeschlossen.

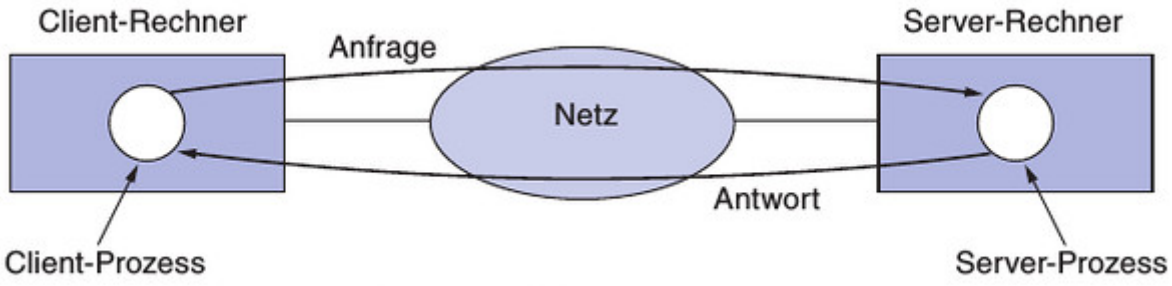

Abb. 2.1.: Client-Server-Modell [TW12, S.26]

Verbindungslos und verbindungsbasiert Protokolle zur Datenübertragung in der Transportschicht (siehe Unterabschnitt 2.1.2) lassen sich in zwei Kategorien aufteilen: Verbindungslos und verbindungsbasiert.
Bei der verbindungsbasierten Übertragung wird zuerst beidseitig eine Anlaufstelle geschaffen und deren Existenz dem jeweiligen Kommunikationspartner mitgeteilt, somit entsteht ein direkter Kanal, über welchen eine dauerhaft bidirektionale Datenübertragung in Form eines Streams geschehen kann. Sofern die Verbindung nicht mehr gewünscht ist, müssen beide Kommunikationsseiten ihren Teil des Kanals schließen.
Die verbindungslose Variante ist mit einem postalischen System vergleichbar. Eine zu übertragene Datenmenge wird in ein Paket eingebettet, welches außer den zu übertragenen Daten Quell- und Adressdaten enthält [TW12]. Sobald das Paket erstellt wurde, wird es in das Netzwerk gegeben.

Die am weitesten verbreiteten Protokolle sind die Internettransportprotokolle. Zwei von ihnen im speziellen finden häufige Verwendung und sollen genannt werden - TCP und UDP. TCP ermöglicht verbindungsbasierte Kommunikation und UDP ermöglicht verbindungslose Kommunikation [TW12, S.616].

RMA-basierte Kommunikation Im Unterschied zur normalen Netzwerkkommunikation, bei welcher die CPU des Quellsystems ein Datenpaket an die Netzwerkhardware (zumeist eine Netzwerkkarte) übergibt, ermöglicht remote memory access (RMA) den direkten Schreib- und Lesezugriff auf den Speicher des Kommunikationspartners oder ermöglicht dem Kommunikationspartner direkten Schreib- und Lesezugriff auf den Speicher des eigenen Systems.

Sockets Sockets stellen einen Kommunikationsendpunkt für verbindungsbasierte Kommunikation aus der Transportschicht dar und bieten eine Reihe von Primitiven für Netzwerkkommunikation. Sie wurden erstmals in der Berkeley-Unix-4.2BSD-Software eingeführt und finden auf vielen Betriebssystemen, insbesondere Unix-basierten, für Netzwerkkommunikation Verwendung [TW12, S.573].

Nagle Algorithmus In einer Kommunikation kann es je nach Anwendung dazu kommen, dass viele kleine Datenpakete auf das Netzwerk geschrieben werden. TCP organisiert zu schreibende Datenpakete in TCP Pakete mit eigenem Header. Sollte für jedes kleine Datenpaket ein eigenes TCP Paket erstellt werden, kann ein solches Programmverhalten zur suboptimalen Ausnutzung bis hin zur Überlastung des Netzwerks führen, da TCP Paketheader vergleichsweise groß sind. Eine Lösung hierfür stellt der Algorithmus von John Nagle dar [TW12, S. 643-644]. Er funktioniert in zwei Abschnitten:

- Sollte ein TCP Paket größentechnisch einen Schwellwert überschreiten oder eine Zeitschranke überschritten sein, sende es.
- Sollte der Gegenpart den Empfang aller vorherigen TCP Pakete bestätigt haben, sende aktuelles Paket größenunabhängig.

Dieses Verhalten ermöglicht es dem TCP-Stack des Betriebssystems durch die inhärent resultierende Verzögerung mehrere kleine zu versendende Datenblöcke in ein größeres TCP Paket zusammenzufassen, was die Netzwerkausnutzung zu Lasten der Latenz optimiert. Des weiteren führt es dazu, dass maximal ein kleines Paket zeitgleich auf einer Verbindung in eine Richtung gesendet wird.
Der Nagle Algorithmus ist standardmäßig in gängigen TCP-Stacks aktiv. Sollte dieses Verhalten nicht gewünscht sein, lässt sich Nagles Algorithmus durch das Setzen von `TCP_NODELAY` für den entsprechenden TCP socket umgehen [Lin20].

Cork `TCP_CORK` zu setzen ist im Linux TCP-Stack eine Möglichkeit die Datendurchsatzrate bei kleinen Datenmengen zu erhöhen [Lin20; Bau05]. Dies wird erreicht, indem kleinere Datenpakete zu größeren TCP-Paketen zusammengefasst werden, bis ein solches Paket den genutzten Schwellwert erreicht hat oder eine gewisse Zeit verstrichen ist. Im Falle des Linux Stacks sind dies 200 Millisekunden [Lin20]. Im Gegensatz zu Nagles Algorithmus setzt corking nicht auf Bestätigung eines Kommunikationspartners, sondern sammelt von diesem unabhängig Datenblöcke zur Kommunikation alleine in Abhängigkeit von Zeit und Paketgröße. Somit wird die Verbindung bei Nagles Algorithmus vor Überlastung geschützt, da großer Wert auf bestätigten Empfang gelegt wird, im Falle von corking aber performanter genutzt, da Netzwerkoverhead und Verzögerung durch Bestätigungsnachrichten entfallen.
`TCP_CORK` ist mit `TCP_NODELAY` kombinierbar, was bei vielen kleinen zu übertragenen Datenmengen unabhängig von einer Empfangsbestätigung des Kommunikationspartners zu insgesamt größeren Paketen führt.

2.1.2. Netzwerkarchitektur

Tannenbaum und Wertherall definieren ein hybrides Referenzmodell zwischen TCP/IP- und dem OSI-Referenzmodell bezüglich Netzwerkarchitektur, welches für diese Arbeit übernommen wird [TW12, S.74]. Wie in Abbildung 2.2 zu sehen, ist es in fünf Schichten unterteilt. Die einzelnen Schichten stellen hierbei Abstraktionsebenen für einen jeweiligen Kontext dar. Eine direkt untergeordnete Schicht stellt der momentan betrachteten

Abb. 2.2.: Referenzmodell Netzwerkarchitektur [TW12]

Schicht Dienste zur Datenübertragung in ihrem jeweiligen Kontext bereit.
Die unterste und erste Schicht ist die Bitübertragungsschicht. Auf ihr wird festgelegt, auf welche Weise elektrische oder analoge Signale ein Bit definieren und übertragen.
Die zweite Schicht ist die Sicherungsschicht. Auf dieser Ebene wird durch Protokolle die Integrität einer Datenübertragung zwischen zwei verbundenen Recheneinheiten gesichert. Es ist möglich, dass es innerhalb der Netzwerkhardware zu Spannungsschwankungen und unregelmäßigen Zeitabständen zwischen zwei Bits kommt und somit unzuverlässige Daten das Resultat einer Übertragung sind. Dies wird in der Sicherungsschicht softwaretechnisch ausgeglichen.
Die dritte Schicht stellt die Vermittlungsschicht dar. Auf ihr soll garantiert werden, dass Nachrichten über ein Netzwerk an ihr Ziel gelangen. Ab einer gewissen Größe des Netzwerkes ist es physisch unmöglich, dass jeder Rechner des Netzes direkt mit jedem anderen physisch verbunden wird. Somit muss die Nachricht über dritte Rechner im Netzwerk weitergeleitet werden, hier greifen die Funktionalitäten der Vermittlungsschicht.
Die vierte Schicht stellt die Transportschicht dar. Sie bietet zuverlässige Ende-zu-Ende Protokolle für die Versendung von Daten und eine Abstraktion von der Vermittlungsschicht [TW12, S.687].
An fünfter Stelle und zuletzt steht die Anwendungsschicht. Sie umfasst alle Anwendungen, welche über das Netzwerk kommunizieren, beispielsweise über Nutzung von Schnittstellen zur Transportschicht.
Die vorliegende Arbeit befasst sich mit Modifikationen der Nutzung der Transportschicht und daraus resultierenden Änderungen der Anwendungsschicht.

2.1.3. Netzwerkkommunikation im HPC-Bereich

Der Bereich High Performance Computing (HPC) beschäftigt sich mit der Verarbeitung großer Datenmengen und Berechnungen, welche einen, relativ zu der momentan zur Verfügung stehenden Rechenkapazität, hohen Rechenaufwand besitzen.
Um solche Aufgaben zu bewältigen, werden leistungsstarke Recheneinheiten benötigt und versucht, die Performanz dieser an den Engpässen der Systeme (Flaschenhals) zu

optimieren. Moderne HPC-Systeme bestehen, wie an den Top 500 [Pro20] zu sehen, aus Netzen vieler mit Hinblick auf Performanz entwickelter einzelner Rechner, so genannten Rechenknoten. Somit handelt es sich auch bei Clustern um Rechennetze.
Das Netzwerk stellt im HPC-Bereich einen klassischen Flaschenhals bezüglich der Datenverarbeitung dar, sofern eine Aufgabe über mehrere Knoten verteilt gelöst werden soll. Dass in diesem Bereich im Netzwerk ein Flaschenhals gesehen wird, kann an den Bemühungen im HPC-Bereich veranschaulicht werden, immer neue Arten der Verbindung von Knoten (Netzwerktopologie) im Hinblick auf mögliche Performanzgewinne zu entwickeln. Aktuelle Beispiele wären hier:

- Tofu Interconnect D ist eine Variante der Tofu Topologie, die im Fugaku Supercomputer umgesetzt wurde [Aji+18]. Es handelt sich um ein sechsdimensionales Netzwerk mit den Achsen X, Y, Z, A, B und C, welche verschiedene Elemente der Knoten miteinander verbinden [Aji+12].

- Cray Cascade ist eine Variante der Dragonfly Topologie, welche beispielsweise im Piz Daint Supercomputer eingesetzt wird [Faa+12]. Hierbei werden einzelne Netzwerkbereiche in Gruppen zusammengefasst und innerhalb der Gruppen eine Vollvermaschung hergestellt. Die einzelnen Mitglieder einer Gruppe erhalten jeweils Verbindungen zu Mitgliedern anderer Gruppen.

- Bei FatTrees handelt es sich um eine Netzwerktopologie welche an Binärbäume angelehnt ist. Je näher an der Wurzel sich die Leitungen befinden, desto höher ist die maximale Leistung des Netzwerkabschnitts. Näheres über FatTrees lässt sich in [Liu+15] erfahren. FatTrees werden unter anderem im Sierra Supercomputer und Summit Supercomputer verwendet.

Auch wenn sich die Illustration des Flaschenhalses auf die Hardwareebene und unteren Schichten des Referenzmodells bezieht, sind alle an einer Netzwerkkommunikation beteiligten Ebenen für eine Betrachtung im Hinblick auf Performanzgewinne interessant für den HPC-Bereich.

2.2. JULEA

2.2.1. Beschreibung von JULEA

JULEA ist ein Framework, welches für das flexible Speichern von Daten im Hinblick auf Forschung und Entwicklung im HPC-Bereich entwickelt wurde und das parallele verteilte Dateisystem ersetzt. Der Nutzer ist hierbei nicht gezwungen, sich mit multiplen Speziallösungen auszukennen. Dies soll den Entwicklungsprozess vereinfachen [Kuh17]. Hierzu bietet es eine Schnittstelle zwischen den eigentlichen Daten- und den Metadatenstoragebackends und den Nutzerprogrammen an, so dass die Nutzer allein durch die JULEA-Schnittstelle mit den gespeicherten Daten interagieren können.
Die JULEA-Schnittstelle für Backends orientiert sich in ihrem Design an bisherigen Speichersystemen. Einige verbreitete Datenbanksysteme wie POSIX [Wal95], MongoDB

[Mon20] oder LevelDB [DG11] werden bereits von JULEA nativ als Backends für Daten- oder Metadatenspeichersysteme unterstützt. Das Framework bietet aber auch die Möglichkeit Schnittstellen für weitere Speichersysteme einzupflegen [Kuh17].
JULEA befindet sich in fortlaufender Entwicklung und bietet diverse Konfigurationsmöglichkeiten, beispielsweise durch die Wahl verschiedener Backends, um sich an die Anforderungen des Nutzers anzupassen.

2.2.2. Netzwerkkommunikation in JULEA

JULEA basiert auf einem Client-Server-Modell (siehe Abbildung 2.3). JULEAs Server sind in Daten- und Metadatenserver unterteilt. Datenserver sind dafür designt, die Nutzung größerer Datenmengen zu unterstützen, während Metadatenserver schnellen Zugriff auf Metadaten ermöglichen [Kuh17]. Sowohl für Daten- als auch Metadatenserver stehen mehrere Backends zur Verfügung.
JULEA ist flexibel einsetzbar. Zwei verschiedene Konfigurationsmöglichkeiten werden in Abbildung 2.4 dargestellt. Beide Beispielanwendungen nutzen den item-Client, während die linke LevelDB als Backend für Metadaten und POSIX als Datenbackend nutzt, benutzt die rechte lokal MongoDB als Metadatenbackend und ebenso POSIX als Datenbackend [Kuh17]. Die Netzwerkkommunikation in JULEA ist bisher über die vom Gnome Projekt entwickelte GSocketConnection gelöst, welche auf GSockets aus der GLib Interfaces and Objects (GIO) Bibliothek basiert [Kuh+20, Dateien: `j_connection_pool.c`, `server.c`]. GSockets orientieren sich an der BSD Socket API und unterstützen somit sowohl Unix als auch Windows Socket Implementationen [RP07, Abschnitt GSockets]. Demzufolge basiert die Kommunikation in JULEA auf der etablierten, oben in Unterabschnitt 2.1.1 beschriebenen Socket-Kommunikation.

JULEA-intern werden die einzelnen Daten in einer Struktur namens JMessages zusammengefasst, bevor diese über das Netzwerk verschickt werden. Eine JMessage besteht aus zwei Haupt- und einer optionalen Komponente. Die erste Hauptkomponente ist ein Header von fester Größe, welcher Felder für Informationen wie Handlungsanweisungen

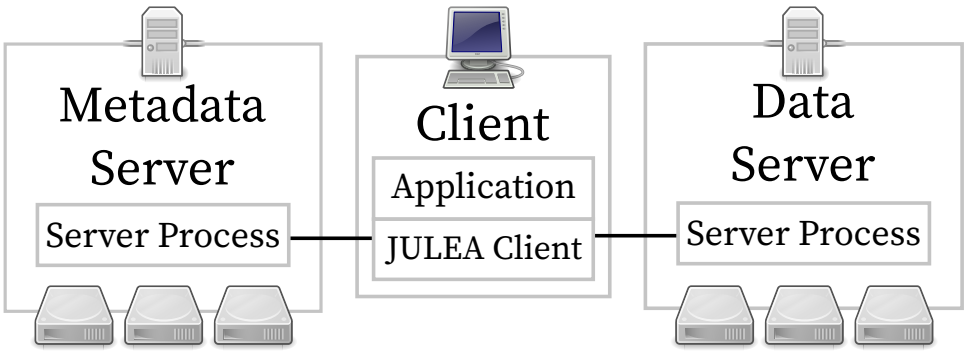

Abb. 2.3.: JULEA Kommunikationsmodell [Kuh17]

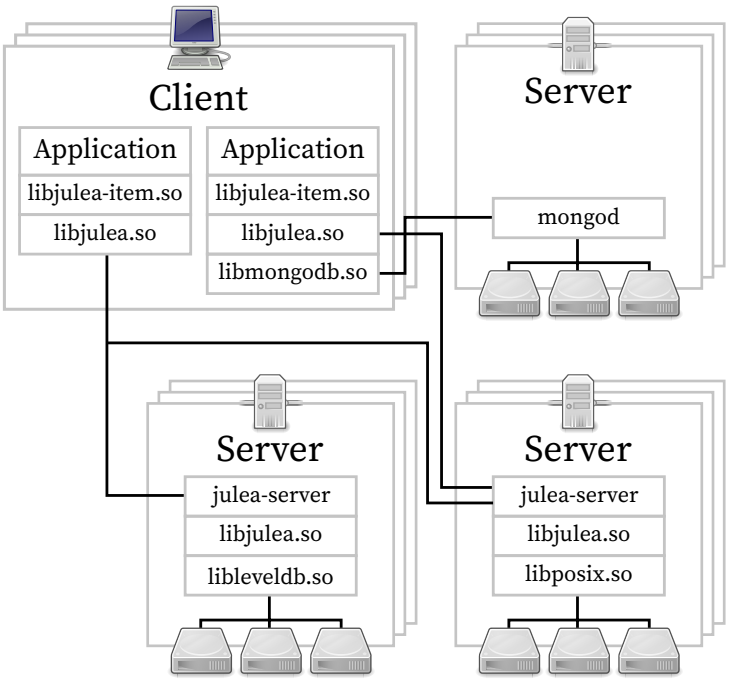

Abb. 2.4.: JULEA-Server bedient zwei Clients verschiedener Konfigurationen [Kuh17]

sowie welche für diverse Metadaten über die gesamte JMessage enthält, was dem Kommunikationspartner die Verarbeitung der gesamten JMessage ermöglicht.
Die zweite ist der Körper der JMessage, er ist von variabler Größe und enthält primär einen Datenblock, welcher die zu verarbeitenden Daten für den Kommunikationspartner enthält. Der Körper kann optional eine Referenz auf eine Liste von zusätzlich zu übertragenden Datenblöcken enthalten.
Sowohl die Header als auch die Körper werden jeweils voneinander separiert über das Netzwerk übertragen, wobei die Header den Kommunikationspartner über die Menge und Größe der zu erwartenden Daten informieren. Sollten Referenzen auf weitere Datenblöcke im Körper oder den Elementen der Liste enthalten sein, werden auch diese voneinander separat über das Netzwerk übertragen.

2.3. OFI und libfabric

Das Open Fabrics Interface (kurz: OFI) bietet Anwendungen direkten Zugang zu den Kommunikationsdiensten eines Computernetzwerkes (*engl. Fabric Communication Services*). Es besteht aus mehreren Komponenten, unter anderem Provider Bibliotheken, Anwendungs-API und Kerneldiensten.
Bei libfabric handelt es sich um eine Bibliothek, welche die Nutzer-API von OFI darstellt [Gru+15].
Aus der Nutzersicht sind vor allem zwei der OFI-Komponenten relevant: Libfabric als Anwendungsschnittstelle sowie die verschiedenen OFI-Provider, durch welche OFI die

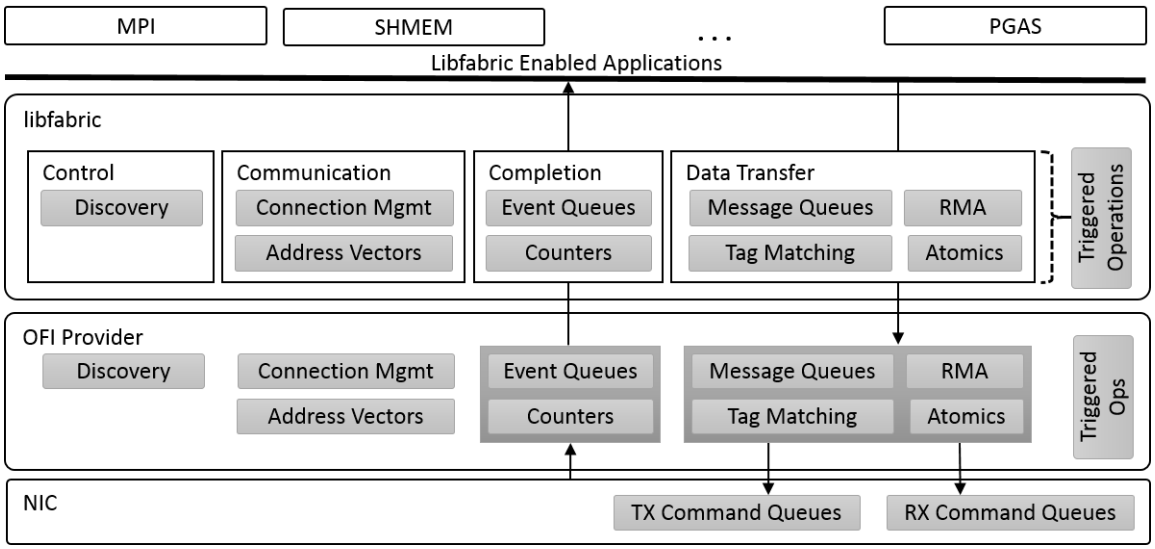

Abb. 2.5.: Libfabric Architektur, High-Level Darstellung [Gru+15]

Netzwerkhardware ansteuert und die Art der Netzwerkkommunikation bestimmt.
Genauer kann dies in Abbildung 2.5 gesehen werden. Libfabric bietet die Schnittstelle zu den eigentlichen Anwendungen und hält Mechanismen zum Aufbau und Verwalten einer Kommunikation (Control und Communication), zur Strukturierung des Anwendungsablaufs basierend auf dem Kommunikationsstatus (Completion) und zur Übertragung der eigentlichen Daten (Data Transfer) bereit. Außerdem werden durch die Konfiguration der benötigten libfabric-Ressourcen die Provider im Hintergrund eingebunden, sofern diese auf dem System vorhanden sind.
Die Provider werden von libfabric durch die Mechanismen zum Datentransfer angesprochen. Daraufhin steuern sie die Netzwerkhardware (NIC) an, um die zu übertragenen Daten dem Netzwerk zuzuführen. Eine Rückmeldung an das System erfolgt über die Event Queues oder Counters.

2.3.1. OFI Kommunikationsmodelle

OFI bietet drei Kommunikationsmodelle für Netzwerkkommunikation an: verlässlich verbindungsbasiert (*engl.: reliable connected*), unzuverlässiges Datagramm (*eng.: unreliable datagram*) und verlässlich verbindungslos (*engl.: relieable connected*) [Gru+15]. Die Verlässlichkeit einer Verbindungsart resultiert aus dem genutzten Protokoll zum Datentransfer [Hef17].
Bei der verbindungsbasierten Variante wird zwischen den libfabric-Ressourcen zweier Systeme ein konstant bereitstehender Datenkanal erzeugt. Die verbindungslose Kommunikation speichert Netzwerkadressen möglicher Kommunikationspartner in Adressvektoren und sendet Daten an die gespeicherten Adressen anstatt über einen Kanal. Der Aufbau einer verbindungsbasierten Kommunikation wird in Unterabschnitt 2.3.4 genauer erläutert.

2.3.2. Provider

Die unterschiedlichen OFI-Provider erfüllen jeweils ein Subset der durch OFI definierten möglichen Fähigkeiten eines Providers. Eine solche Fähigkeit ist die Unterstützung mindestens einer der in Unterabschnitt 2.3.1 beschriebenen Kommunikationsmodelle. Die Anwendung bestimmt später durch die Spezifikation der benötigten Eigenschaften ein Profil für den zu verwendenden Provider. Libfabric schlägt daraufhin alle Provider, welche die Anforderungen erfüllen, beginnend mit dem performantesten vor.
Die meisten Provider unterstützen nur einen speziellen Hardwaretyp, bieten dafür aber auf diesem bessere Performanz als die breitflächig verfügbaren Provider [Hef17]. Eine Ausnahme hierzu stellen der udp und der sockets Provider dar, wobei vor allem der sockets Provider versucht, möglichst viele Fähigkeiten zu implementieren und anzubieten. Beide bieten zum einen eine generelle Testumgebung als auch einen Rückfallmechanismus an, falls kein anderer Provider zu finden sein sollte.

Genutzte Provider

sockets OFI bietet mit libfabric eine weit gefasste Schnittstelle für Nutzeranwendungen an. Der sockets Provider versucht einer Anwendung alle Funktionalitäten anzubieten [Ope20b, Eintrag `fi_sockets(7)`]. Er ist über TCP-sockets implementiert, um die Entwicklung auf möglichst vielen Systemen anzubieten. Hierdurch ist er im Vergleich zu den anderen Providern im Hinblick auf Performanz eingeschränkt.

verbs Der verbs-Provider ist ein auf die Infiniband-Hardware spezialisierter Provider, welcher verbindungsbasierte Kommunikation sowohl RMA- als auch nachrichtenbasiert anbietet. Er verwendet für die Kommunikation die Linux Verbs API [Ope20b, Eintrag `fi_verbs(7)`].

2.3.3. libfabric-Ressourcen, Ausschnitt

Die Informationen dieses Abschnitts stammen aus dem OFI Developer-Guide [Hef17] und dem libfabric Manual [Ope20b].

fi_info

Bei einem `fi_info` (siehe Listing A.1) handelt es sich um eine Struktur, welche die benötigten Parameter für den Aufbau der meisten libfabric-Strukturen bezüglich des dazugehörigen Providers hält. Des weiteren beinhaltet es eine einfach verkettete Liste über alle gefundenen Provider und Informationen über Quell- und Zieladressen einer Verbindung oder Datenübertragung. Das erste Element der Liste enthält die potentiell performanteste Konfiguration. Außerdem hält eine `fi_info` Struktur Capabilities und Mode Felder.
Über das Capability-Feld kann eine Anwendung die benötigten Verhaltensweisen eines Providers definieren und somit bei der Providerwahl ungeeignete Provider herausfiltern,

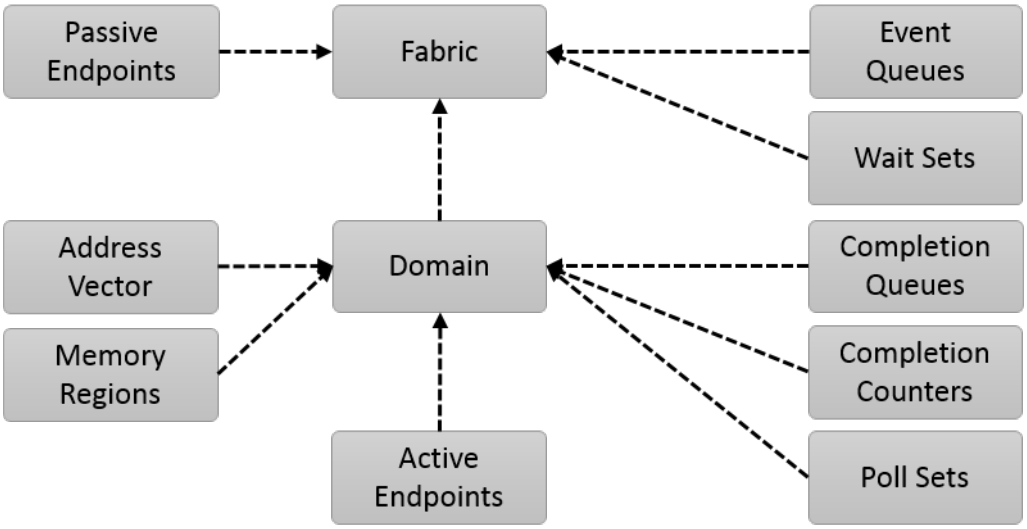

Abb. 2.6.: Libfabric Objektmodell [Gru+15]

während über das Mode-Feld ein Provider der Anwendung wiederum seine Einschränkungen mitteilt.
Die Erzeugung einer `fi_info`-Struktur ist im Regelfall der erste Teil des Aufbaus der Ressourcen für eine libfabric Kommunikation. Hierfür wird eine häufig `hints` genannte, durch `fi_allocinfo` erzeugte leere `fi_info`-Struktur erstellt, in welcher die von der Anwendung benötigten Anforderungen an den darunterliegenden Provider durch die Anwendung gesetzt werden (siehe Listing 2.1 Zeile 1-5).
Nun wird die eigentliche im weiteren verwendete `fi_info`-Struktur erzeugt. Dies geschieht über den Aufruf der `fi_getinfo`-Funktion (siehe Listing 2.1 Zeile 7), welcher auch die `hints`-Struktur übergeben wird. Das zurückgegebene `fi_info` enthält nun eine Liste der Provider, welche auf dem Host-System verfügbar waren und den Capabilities entsprechen. Sobald `fi_getinfo` eine `fi_info` zurückgegeben hat, können mit diesem die lokalen libfabric-Ressourcen angelegt werden.
Eine `fi_info`-Struktur kann über `fi_freeinfo` geschlossen werden (siehe Listing 2.1 Zeile 13).

Fabric

Eine Fabric stellt das Top-Level Objekt der libfabric-Kommunikation dar (siehe Abbildung 2.6). Aus der Anwendungssicht stellt es die zentrale Verbindungsstelle aller libfabric-Ressourcen dar. Eine Fabric bietet einer Anwendung außerdem die Möglichkeit den verwendeten Provider namentlich zu identifizieren.
Im Hintergrund handelt es sich um eine Verbindung aus netzwerkbezogenen Hardware- und Softwareressourcen. Libfabric ermöglicht die Nutzung mehrerer Fabrics zur gleichen Zeit auf dem selben System. Sie werden über die im `fi_info` enthaltende Parameter-Struktur `fi_fabric_attr` (Listing A.2) bestimmt. Durch den Aufruf der `fi_fabric-`

Funktion wird eine Fabric erstellt (siehe Listing 2.1 Zeile 15).
Wie alle anderen libfabric-Ressourcen enthält jede Fabric einen `fid`, einen Fabric Identifier, welcher über die Natur der Ressource Auskunft gibt.
Wie alle folgenden Objekte wird eine Fabric geschlossen, indem der libfabric-weit einsetzbaren Funktion `fi_close` das `fid` des jeweiligen Objektes übergeben wird.

Domain

Mit Domain ist im libfabric-Kontext ein Objekt gemeint, welches zwei Hauptfunktionen erfüllt. Zum einen dient eine Domain im Normalfall als Repräsentation einer bestimmten Netzwerkkarte oder Hardwareports innerhalb einer Fabric. Zum anderen bestimmt sie auch die Beziehung zwischen mit ihr assoziierten libfabric-Ressourcen untereinander (siehe Abbildung 2.6). Aus Fehlermitigationsgründen kann eine Domain allerdings auch mehrere Hardwarekomponenten eines Netzwerkes repräsentieren.
Eine Domain wird mit dem Aufruf der `fi_domain`-Funktion erzeugt, erhält ihre zur Erzeugung nötigen Informationen aus der Übergabe einer `fi_info`-Struktur und der übergeordneten Fabric (siehe Listing 2.1 Zeile 17). Die bestimmenden Parameter finden sich in der `fi_domain_attr`-Struktur des `fi_infos`, zu sehen in Listing A.3. Domains können mit Event Queues durch den Aufruf der `fi_domain_bind`-Funktion assoziiert werden (siehe Listing 2.1 Zeile 19).

Endpoint

Endpoints stellen die Schnittstelle zwischen der Anwendung und dem Netzwerk bezüglich der Kommunikation dar. Es existieren zwei Arten von Endpoints in libfabric, aktive und passive.

Aktive Endpoints können verbindungsorientiert oder verbindungslos sein. Über sie läuft die Datenübertragung ab. Während ihrer Erzeugung werden sie durch die angegebene `fi_info` konfiguriert und an die übergeordnete Domain gebunden (siehe Abbildung 2.6). Zum Zeitpunkt ihrer Erzeugung befinden sie sich in einem inaktiven Status. Um einen aktiven Endpoint in einen aktiven Status zu überführen, muss er zuerst mit einer libfabric-Ressource assoziiert werden. Im Normalfall ist ein aktiver Endpoint mit einer Event Queue und mindestens einem Completion Mechanismus, beispielsweise einem Completion Counter oder einer Completion Queue, assoziiert (siehe Abbildung 2.7).
Primär werden als Completion Mechanismus Completion Queues eingesetzt. Diese können auch durch das Setzen von flags beim Aufruf von `fi_ep_bind` (siehe Listing 2.1 Zeile 22-27) zur Assoziation nur auf Empfang oder Senden Ereignisse spezialisiert werden. Der jeweils andere Part muss dann durch eine weitere Completion Queue aufgefangen werden. Wenn alle benötigten Ressourcen an einen Endpoint gebunden sind, kann per `fi_connect` [Ope20b, Eintrag `fi_cm(3)`] ein Endpoint mit einem Kommunikationspartner verbunden werden. Näheres zum Aufbau einer Verbindung ist in Abschnitt 3.1 zu finden. Alternativ kann ein aktiver Endpoint auch verbindungslos bleiben und über einen Addressvektor Daten an einen Kommunikationspartner versenden.

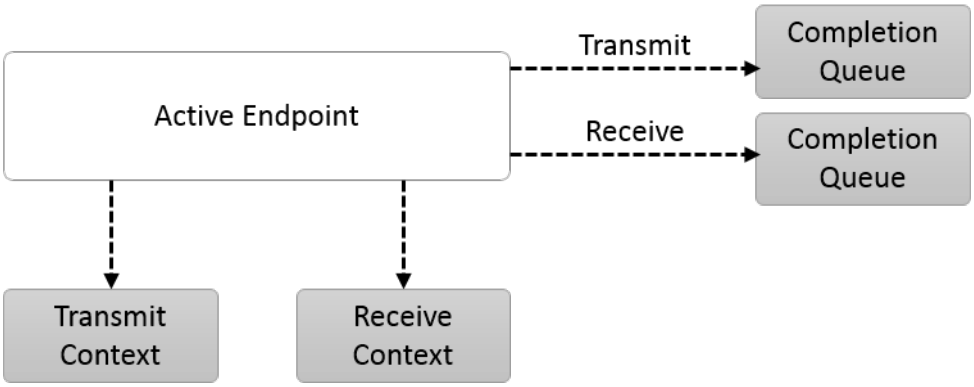

Abb. 2.7.: Libfabric aktiver Endpoint mit Connection Queue verbunden [Gru+15]

Sobald ein aktiver Endpoint durch `fi_connect` (Verbindungsanfrage), `fi_accept` (Annahme einer Verbindungsanfrage) oder `fi_enable` aktiviert wird, kann er zu jeglicher Datenübertragung, die der jeweilige Provider anbietet, genutzt werden.
Sollte der aktive Endpoint zum Zeitpunkt seiner Beendigung verbunden sein, kann die Verbindung über die `fi_shutdown`-Funktion aus [Ope20b, Eintrag `fi_cm(3)`] geschlossen werden.

Außerdem besitzen aktive Endpoints getrennte Transmit und Receive Contexts, welche für das Senden und Empfang von Daten verwendet und normalerweise in den Adressraum des Hostprozesses abgebildet werden.

Passive Endpoints ermöglichen keine Datenübertragung sondern sollen Verbindungsanfragen entgegennehmen und einer Anwendung so ermöglichen, den Gegenpart des anfragenden aktiven Endpoints zu erstellen. Somit sind passive Endpoints ausschließlich verbindungsorientiert. Sie werden durch den Aufruf `fi_passive_ep` mit einer `fi_info`-Struktur erzeugt und an eine übergeordnete Fabric gebunden.
Bevor passive Endpoints per `fi_listen`-Funktion aus [Ope20b, Eintrag `fi_cm(3)`] zum Empfang von potentiellen Verbindungsanfragen gebracht werden können, müssen sie durch `fi_pep_bind` mit einer Event Queue assoziiert werden (siehe Listing 2.2 Zeile 17-21). Weiteres zur Rolle passiver Endpoints beim Verbindungsaufbau findet sich in Abschnitt 3.1.

Event Queues

Eine Event Queue stellt einen Mechnismus zur Kommunikation von durch libfabric aufgefangene Ereignisse an die Anwendung dar. Die Ereignisse werden in einer Warteschleife gesichert, bis sie durch die Anwendung abgeholt werden. Die Art dieser Ereignisse ist stark durch die Natur der jeweiligen Ressource beeinflusst. Es kann sich hierbei beispielsweise um Fehler, Verbindungsevents oder Kontrolloperationen handeln.
Eine Event Queue wird durch die in `fi_info` enthaltene Parameter-Struktur `fi_eq_-`

`attr` (siehe Listing A.6) und der übergeordneten Fabric erzeugt (siehe Abbildung 2.6). Event Queues können durch die jeweiligen `fi_x_bind`-Funktionen der entsprechenden libfabric-Ressourcen mit ihnen assoziiert werden (siehe Listing 2.1 Zeilen 19 und 25). Durch die Assoziierung können Ereignisse der jeweiligen Ressource auf die Event Queue gegeben werden.
Für die Abholung der Ereignisse stehen blockierende oder nicht blockierende Funktionen in libfabric zur Verfügung.

Completion Queues

Completion Queues dienen als Mechanismus, um einer Anwendung Informationen über Ereignisse bezüglich Datentransfers zugänglich zu machen. Hierbei handelt es sich primär um Ereignisse bezüglich des Abschlusses von Übertragungen.
Im Gegensatz zu den anderen bisher erwähnten Ressourcen benötigt eine Completion Queue keine Informationen aus einer `fi_info`, sondern bedarf einer `fi_cq_attr`-Struktur, über welche das Verhalten der zu erstellenden Completion Queue bestimmt werden kann. Completion Queues werden mit einer übergeordneten Domain erstellt (siehe Listing 2.1 Zeile 23-24 und Abbildung 2.6).
Damit die Anwendung Ereignisse von der Completion Queue lesen kann, stehen eine Standardfunktion und eine erweiterte Funktion eines Lesebefehls zur Wahl, sowohl in blockierender, als auch in nicht-blockierender Variante. Für die blockierenden Varianten muss im `fi_cq_attr` ein `wait_object` definiert werden, welches zum Blockieren verwendet wird.
Sollte für die Beendigung einer Completion Queue das blockierende Verhalten aufgelöst werden müssen, kann die Funktion `fi_cq_signal` aus einem anderen Thread die übergebene Completion Queue aufwecken.

Memory Registration

Bei Memory Registration handelt es sich um eine Methode Speicherbuffer mit libfabric-Ressourcen zu verknüpfen. Die Buffer werden hierdurch mit Zugriffsfunktionalitäten für den Zugriff anderer libfabric-Ressourcen, beispielsweise einem Zugriffsschlüssel und einem Identifikationsmechanismus für die Buffer, ausgestattet. Die Zugriffsschlüssel werden für jegliche RMA-Kommunikation benötigt. Allerdings benötigen einige Provider auch für andere Kommunikationsmethoden registrierten Speicher, welche dies über die mode-Bits kommunizieren.
Ein Speicherbereich wird über die Funktion `fi_mr_reg` registriert.

Codebeispiel libfabric Ressourcenaufbau:

Im Folgenden wird der Aufbau von libfabric-Ressourcen, auf welche die vorherigen Abschnitte Bezug nehmen, client- und serverseitig dargestellt. Die Definitionen verwendeter Variablen finden sich in Listing A.7. Auf der Clientseite bis zur Verbindungsanfrage, auf der Serverseite bis zum Listening. Der hier nicht repräsentierte Aufbau des aktiven

Endpoints auf der Serverseite geschieht analog zur Clientseite mit der der Verbindungsanfrage zu entnehmenden, `fi_info`-Struktur.
Das Beispiel ist so konfiguriert, dass der Server anhand seiner IPV4-Adresse, repräsentiert in einer `sockaddr_in`, adressiert wird und die angelegten Ressourcen für eine verbindungsbasierte Kommunikation basierend auf dem sockets Provider geeignet sind. Das Error-Handling wird nur einmal exemplarisch an dem `fi_getinfo` demonstriert (Listing 2.1 Zeile 7 bis 11). Error-Handling für andere libfabric-Funktionen ist analog durchzuführen.

```
1  hints = fi_allocinfo();
2
3  hints->add_format = FI_SOCKADDR_IN;
4  hints->fabric_attr->prov_name = strdup("sockets");
5  hints->ep_attr->type = FI_EP_MSG;
6
7  error = fi_getinfo(FI_VERSION(1, 8),
       ↪inet_ntoa(node->sin_addr), service, 0, hints, &info);
8  if (error < 0)
9  {
10    printf("Error with fi_getinfo, Error Message: %s",
         ↪fi_strerror(abs(error)));
11 }
12
13 fi_freeinfo(hints);
14
15 fi_fabric(info->fabric_attr, &fabric, NULL);
16
17 fi_domain(fabric, info, &domain, NULL);
18 fi_eq_open(fabric, eq_attr, &domain_eq, NULL);
19 fi_domain_bind(domain, domain_eq->fid, 0);
20
21 fi_endpoint(domain, info, &active_ep, NULL);
22 fi_eq_open(fabric, eq_attr, &event_queue, NULL);
23 fi_cq_open(domain, cq_attr, &cq_transmit, NULL);
24 fi_cq_open(domain, cq_attr, &cq_receive, NULL);
25 fi_ep_bind(active_ep, event_queue->fid, 0);
26 fi_ep_bind(active_ep, cq_transmit->fid, FI_TRANSMIT);
27 fi_ep_bind(active_ep, cq_receive->fid, FI_RECV);
28
29 fi_connect(active_ep, node, NULL, 0);
```

Listing 2.1: Libfabric Ressourcenaufbau Client

```
1  hints = fi_allocinfo();
2
3  hints->add_format = FI_SOCKADDR_IN;
4  hints->fabric_attr->prov_name = strdup("sockets");
5  hints->ep_attr->type = FI_EP_MSG;
6
7  error = fi_getinfo(FI_VERSION(1, 8),
       ↪inet_ntoa(node->sin_addr), service, 0, hints, &info);
8  if (error < 0)
9  {
10    printf("Error with fi_getinfo, Error Message: %s",
        ↪fi_strerror(abs(error)));
11 }
12
13 fi_freeinfo(hints);
14
15 fi_fabric(info->fabric_attr, &fabric, NULL);
16
17 fi_passive_ep(fabric, info, &passive_ep, NULL);
18 fi_eq_open(fabric, eq_attr, &event_queue, NULL);
19 fi_pep_bind(passive_ep, event_queue->fid, 0);
20
21 fi_listen(passive_ep);
```

Listing 2.2: Libfabric Ressourcenaufbau Server

2.3.4. Kommunikation in libfabric

Die Informationen dieses Abschnitts stammen aus dem OFI Developer-Guide [Hef17] und dem libfabric Manual [Ope20b].

Verbindungsbasierte Kommunikation

Zum Aufbau einer verbindungsbasierten Kommunikation über libfabric (siehe Abbildung 2.8) müssen sowohl client- als auch serverseitig die in Unterabschnitt 2.3.3 beschriebenen Ressourcen angelegt werden. Sobald die durch `fi_connect` gestartete Verbindungsanfrage den Server und den passiven Endpoint erreicht, wird auf dessen Event Queue ein `FI_CONNREQ`-Event generiert.
Nun muss der Server, analog zu Listing 2.1, einen aktiven Endpoint mit dem im `FI_CONNREQ`-Event enthaltenen `fi_info` aufbauen. Als letzter Schritt wird die Verbindung über einen serverseitigen Aufruf von `fi_accept` mit dem erstellten aktiven Endpoint bestätigt. Auf den Event Queues der aktiven Endpoints, sowohl des Servers als auch des Clients, wird ein `FI_CONNECTED`-Event generiert (siehe Abbildung 2.8).
Eine Verbindung ist nun aktiv und kann für Datentransfers genutzt werden.

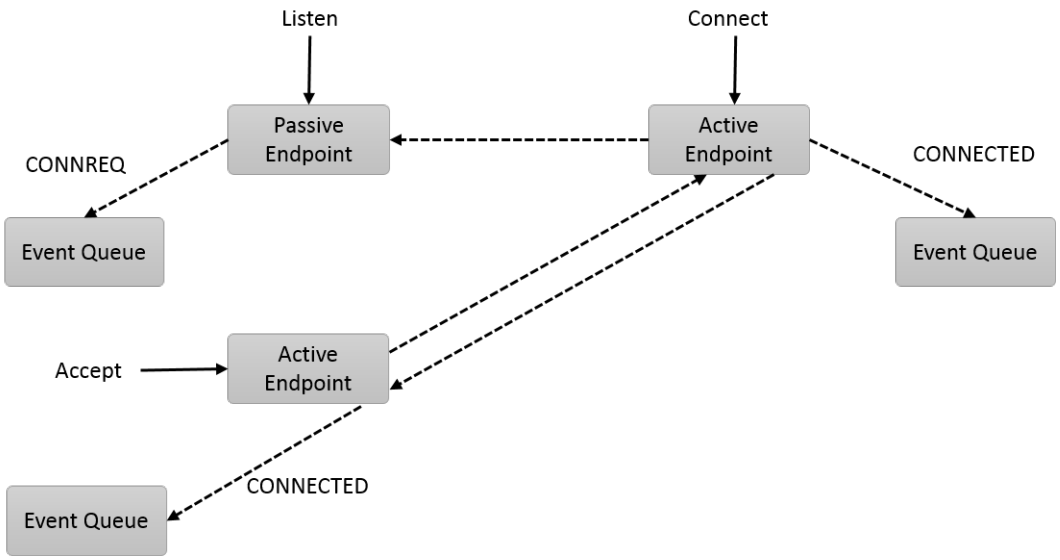

Abb. 2.8.: Libfabric verbindungsbasierte Kommunikation [Gru+15]

```
1  event_entry = malloc(event_entry_size)
2
3  fi_eq_sread(event_queue, &event, event_entry,
       ↪event_entry_size, -1, 0);
4
5  if(event = FI_CONNREQ)
6  {
7      info = event_entry->info;
8      free (event_entry);
9      // Baue Domain und Endpoint auf
10     fi_accept(active_ep);
11 }
```

Listing 2.3: Libfabric Verbindungsaufbau serverseitig

Verbindungslose Kommunikation

Für den Aufbau einer verbindungslosen Kommunikation muss ein aktiver Endpoint mit einem Addressvektor assoziiert werden (siehe Abbildung 2.9). In den Addressvektoren findet zum Zeitpunkt der Eingabe eine Abbildung von Adressinformationen höherer Schichten, beispielsweise einer IP, auf Adressinformationen niedrigerer Schichten, beispielsweise MAC-Adressen, statt. Somit kann der aktive Endpoint beim Aufruf einer Datenübertragung die Zeit für die Abbildung einsparen.
Addressvektoren liefern den Erfolg einer Abbildung über die verbundene Event Queue zurück.

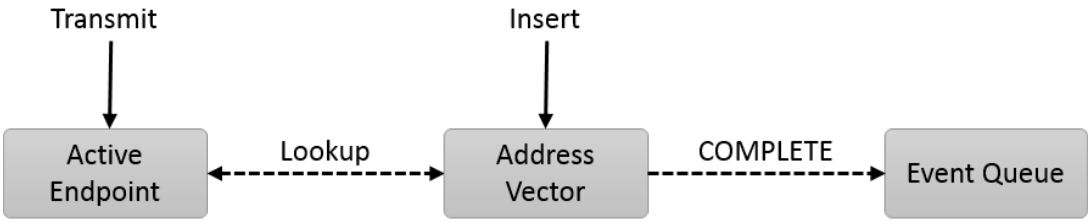

Abb. 2.9.: Libfabric verbindungslose Kommunikation [Gru+15]

Da es sich um verbindungslose Kommunikation handelt, muss die Anwendung sicherstellen, dass auf der Seite des Kommunikationspartners ein aktiver Endpoint vorhanden ist, der die Nachricht entgegen nehmen kann.

3. Design

In diesem Kapitel wird das Design der libfabric-Integration in JULEA beschrieben.

Es wurden im Laufe dieser Arbeit zwei verschiedene durch das OFI angebotene Kommunikationsmodelle für die Datenübertragungen in JULEA integriert. JULEAs Basiskommunikationsdesign ist auf eine konstant gehaltene Verbindung zwischen Server- und Clientsystem ausgerichtet.
Demzufolge sind beide Kommunikationsmodelle verbindungsbasiert. Zum einen wurde ein rein nachrichtenbasiertes Modell genutzt. Zum anderen wurde ein hybrides Modell, das für die Kommunikation von Initialdaten (Metadaten und kleinere Pakete) eine nachrichtenbasierte Kommunikation verwendet. Für die an eine JMessage angehängten Datenblöcke, wird im hybriden Modell ein entfernter Speicherzugriff verwendet (RMA), da sie die größten Datenmengen enthalten.
Erstere Integration wurde im Vorlauf dieser Arbeit begonnen und in ihrem Lauf fertig gestellt, letztere stellt ein Produkt dieser Arbeit dar. Beide orientieren sich in ihrem Ablauf an dem bisherigen Kommunikationsmodell.

3.1. Verbindungsverwaltung

Zum Aufbau und Verwaltung einer Verbindung werden in beiden Varianten Teile der im libfabric Manual [Ope20b] beschriebenen Funktionalitäten verwendet.

3.1.1. Client

Damit für die Nutzung aktive Verbindungen bereit stehen, werden aktive, ungenutzte Verbindungen in einem Verbindungspool (`j_connectio_pool`) abgelegt und nur neue Verbindungen aufgebaut, sofern keine ungenutzten Verbindungen im Pool vorhanden sind. Bei jeder Verbindung wird, sobald sie vom Pool angefordert wird, zur Sicherheit geprüft, ob das Gegenüber die Verbindung geschlossen hat. Sollte es nicht genügend aktive Verbindungen im Pool geben, wird eine neue aufgebaut (siehe hierzu Listing 2.1).
Dieses Design gilt auch für den Fall der hybriden Variante. Für die Nutzung eines Providers, welcher nur RMA-Funktionalitäten anbietet, oder für die Verwendung unterschiedlicher Provider müssten hier zwei Infrastrukturen aufgebaut werden. Ein solches Design wurde verworfen, da Interferenzen existierten und in den meisten Kommunikationsfällen das Bereithalten einer doppelten Infrastruktur irrelevant ist.
Nach dem Ressourcenaufbau wird eine Verbindungsanfrage an den passiven Endpoint des Servers gesendet und auf die Antwort des Servers gewartet. Sollte der Server die

Gegenseite aufbauen und die Verbindung bestätigen, wird die Verbindung dem Clienten zur Verfügung gestellt und nach Gebrauch in den Pool übergeben.

Sobald der Client beendet wird, werden alle im Pool befindlichen Endpoints sowie ihre Ressourcen abgebaut und zuvor eine Nachricht an den Server geschickt, welche das Beenden der Verbindung signalisiert.

3.1.2. Server

Der Server besteht aus der Sicht des Verbindungsaufbaus aus zwei Bestandteilen: Einem Hauptprozess, welcher auf hereinkommende Verbindungsanfragen wartet und diese bearbeitet, und den Verbindungen, die jeweils in einem eigenen Thread arbeiten, in welchem die Datenverarbeitung geschieht.
Zuerst muss der Hauptprozess erstellt werden, welcher eine listening-Schleife als Hauptbestandteil enthält. Hierzu wird pro verfügbarem und auf die gesetzten Voraussetzungen passendem Netzwerkinterface ein passiver Endpoint angelegt, welcher die Funktion eines Listeners übernimmt und auf eingehende Verbindungsanfragen wartet. Die einzelnen passiven Endpoints teilen sich eine Event Queue. Diese wird auf eingegangene Events bezüglich Verbindungsanfragen auf einem der passiven Endpoints überprüft (siehe Listing 3.1 Zeile 11).
Außerdem stellt der Hauptprozess bei jedem Schleifendurchlauf sicher, ob der Server per Systemsignal beendet werden soll und ob sich die einzelnen angelegten Threads schon beendet haben (siehe Listing 3.1 Zeile 1 und 36).
Sofern eine Verbindungsanfrage festgestellt wird (Listing 3.1 Zeile 11 und 13), wird eine Struktur mit Informationen für das Erstellen der Ressourcen in den einzelnen Threads erstellt (Listing 3.1 Zeile 15 und 17) und einem neuen Thread übergeben (Listing 3.1 Zeile 19).

```
1  do
2  {
3      ssize_t con_req_size = 0;
4      uint32_t event = 0;
5      struct fi_eq_cm_entry* event_entry =
           ↪malloc(event_entry_size);
6      struct fi_eq_err_entry event_queue_err_entry;
7      JConData* con_data;
8
9      thread_data = NULL;
10
11     con_req_size = fi_eq_sread(passive_ep_event_queue, &event,
           ↪event_entry, event_entry_size, 1000, 0);
12
13     if (event == FI_CONNREQ && j_server_running &&
           ↪j_thread_running)
```

```
14      {
15          thread_data = j_thread_data_new(jd_configuration,
                ↪jfabric, domain_manager,
                ↪j_con_data_get_uuid(con_data), thread_cq_array,
                ↪thread_cq_array_mutex, j_server_running,
                ↪j_thread_running, thread_count, jd_statistics,
                ↪jd_statistics_mutex[0]);
16
17          j_thread_data_set_msg_event(thread_data, event_entry);
18
19          g_thread_new(NULL, *j_thread_function,
                ↪(gpointer)thread_data);
20      }
21      else if (event == FI_CONNREQ)
22          // Um Suche nach dem entsprechenden passiven Endpoint
                ↪gekürzt
23          fi_reject(passive_ep, event_entry->fid, NULL, 0);
24          free(event_entry);
25      else
26          free(event_entry);
27  } while (j_server_running == TRUE);
```

Listing 3.1: Server Hauptschleife (gekürzt), RMA-Variante

Sollte eine Verbindungsanfrage festgestellt werden, der Server aber ein Signal zum Beenden erhalten hat, werden die Verbindungsanfragen abgelehnt (Listing 3.1 Zeile 21 bis 24). Wenn keiner dieser Fälle eintritt, wird angenommen, dass keine Verbindungsanfrage in diesem Schleifendurchlauf geschehen ist. Allerdings wird, um im Sinne einer defensiven Programmierung potentielle Konflikte mit dem zuvor angelegten `event_entry` zu vermeiden, `free(event_entry)` trotzdem aufgerufen (Listing 3.1 Zeile 25 und 26).
Die Verbindungsthreads legen, sobald sie aufgerufen wurden, die libfabric-Ressourcen für eine aktive Verbindung an, bestätigen die Verbindungsanfrage und erzeugen so auf der Event Queue des Gegenübers sowie auf der eigenen ein `FI_CONNECTED` Event, welches eine neue, aktive Verbindung anzeigt. Hiernach gehen die Threads dazu über, Daten zu empfangen.
Sobald der Server per Systemsignal beendet wird, werden die Threads, welche sich normalerweise in einem Zustand des Listenings befinden, aufgeweckt. Daraufhin wird die Verbindung geschlossen, dann die Ressourcen und schließlich die einzelnen Threads beendet.

3.2. Verhalten Client und Server

3.2.1. Client

Der Client stellt den aktiven Part einer JULEA-Datenübertragung dar. Er initialisiert eine Datenübertragung durch die `j_message_write_x`-Funktionen. Sofern die Datenübertragung eine Antwort verlangt, wartet der Server durch den Aufruf der `j_message_read`-Funktion auf diese.
Wenn die Antwort die Übertragung von zusätzlichen Datenblöcken erfordert, was bei bestimmten Operationstypen möglich ist, werden diese extern durch direkten Aufruf der benötigten libfabric-Funktionen empfangen. Sobald die Antwort erfolgreich übertragen wurde, kann der Client eine neue Datenübertragung in die Wege leiten.

3.2.2. Server

Der Server stellt den passiven Part einer JULEA-Datenübertragung dar. Ein Thread pro aktiver Verbindung wartet jeweils auf eine beginnende Datenübertragung. Präziser die `j_message_read` Funktion wartet darauf, dass ein Completion Event für den Eingang der Übertragung eines Headers einer JMessage generiert wird, und der Serverteil der Verbindung verharrt in der `j_message_read`-Funktion.
Sollte ein solches Event eintreten, werden zuerst durch `j_message_read` die Übertragung der ersten beiden oben erwähnten Schritte abgearbeitet. Sofern durch den Operationstyp definiert, empfängt der Server noch zusätzliche Datenblöcke über die jeweilige im Header angefragte Kommunikationsstrecke. Daraufhin bearbeitet der Server, je nach spezifiziertem Operationstyp, die empfangenen Daten, generiert eine Antwortnachricht und verschickt diese durch eine der beiden `j_message_write_x`-Funktionen an den Clienten. Schlussendlich werden je nach Kommunikationsmodell verschiedene Aufräumarbeiten vorgenommen.

3.3. Nachrichtenbasierte Datenübertragung

Die nachrichtenbasierte Datenübertragung verwendet die im libfabric Manual [Ope20b] beschriebenen Funktionalitäten zur Datenübertragung und deren Steuerung. Versenden und Empfangen einer JMessage geschieht bei einer nachrichtenbasierten Datenübertragung in drei Schritten:

- Übertragung des JMessage-Headers, welcher Metadaten über die JMessage und die Übertragung enthält und eine feste Größe besitzt.
- Übertragung des JMessage-Körpers von variabler Größe, welcher primär einige Daten für kleinere Übertragungen sowie einen Verweis auf zusätzliche potentielle Daten enthält.
- Übertragung von zusätzlichen Daten von variabler Größe. Im Normalfall fallen hier die größten Datenmengen an.

Diese drei Schritte sind für das Senden in eine Funktion (`j_message_write_msg`) gekapselt. Aufgrund der Unterschiede in den Programmverläufen zwischen Client und Server sind für das Empfangen einer JMessage nur die ersten beiden Schritte in eine Funktion (`j_message_read`) gekapselt und der letzte Schritt mit seinen jeweiligen Spezifikationen in Client und Server direkt implementiert.

3.3.1. j_message_write_msg

Im Falle von nachrichtenbasierter Datenübertragung besteht diese Funktion aus der Umsetzung der drei oben genannten Schritte. Als erstes wird der Header der JMessage gesendet (Listing 3.2 Zeile 1 und 2), danach der Körper übertragen (Listing 3.2 Zeile 4 bis 5) und im Anschluss, sofern vorhanden, eine variable Menge an weiteren Datenblöcken übertragen (Listing 3.2 Zeile 19 bis 34).

```
1  fi_send(j_endpoint_get_endpoint(jendpoint),
       ↪(void*)&(message->header), sizeof(JMessageHeader), NULL,
       ↪0, NULL);
2  j_endpoint_read_completion_queue(
       ↪j_endpoint_get_completion_queue(jendpoint, FI_TRANSMIT),
       ↪-1, "TRANSMIT", "JMessage Header");
3
4  fi_send(j_endpoint_get_endpoint(jendpoint), message->data,
       ↪(size_t)j_message_length(message), NULL, 0, NULL);
5  j_endpoint_read_completion_queue(
       ↪j_endpoint_get_completion_queue(jendpoint, FI_TRANSMIT),
       ↪-1, "Transmit", "JMessage");
6
7  if (message->send_list != NULL)
8  {
9      iterator = j_list_iterator_new(message->send_list);
10     while (j_list_iterator_next(iterator))
11     {
12         JMessageData* message_data =
               ↪j_list_iterator_get(iterator);
13         fi_send(j_endpoint_get_endpoint(jendpoint),
               ↪message_data->data, message_data->length, NULL, 0,
               ↪NULL);
14         j_endpoint_read_completion_queue(
               ↪j_endpoint_get_completion_queue(jendpoint,
               ↪FI_TRANSMIT), -1, "Transmit", "JMessage list");
15     }
16 }
```

Listing 3.2: j_message_write_msg, gekürzt

Jeder dieser drei Schritte wartet darauf, dass libfabric durch eine Completion Queue das erfolgreiche Verschicken, nicht aber das Empfangen, einer Nachricht verifiziert. Sollten mehr als ein Datenblock in Schritt 3 verschickt werden, wird auf das erfolgreiche Versenden pro Datenblock gewartet (Listing 3.2 Zeile 9 bis 16).

3.3.2. j_message_read

Im Falle von nachrichtenbasierter Datenübertragung besteht diese Funktion aus der Umsetzung der ersten beiden oben genannten Schritte.
Als erstes wird indefinit gewartet, bis auf der jeweiligen Completion Queue des aufrufenden Endpoints ein Completion Event für den Empfang eines Headers einer JMessage generiert wird (Listing 3.3 Zeile 1 und 2). Hierfür die benötigten Buffer bereitzustellen ist möglich, da der Header von fester Größe ist. Daraufhin werden, ermöglicht durch die Informationen aus dem Header, die benötigten Ressourcen für den Empfang des Körpers angelegt, ein Lesezugriff auf das Netzwerk bezüglich des Körpers gestartet und gewartet, bis dieser Zugriff abgeschlossen ist (Listing 3.3 Zeile 13 und 14).
Schritt 3 ist aufgrund der unterschiedlichen Behandlung von zusätzlichen Datenblöcken auf Client- beziehungsweise Serverseite nicht in `j_message_read` generalisiert, sondern wird auf beiden Seiten jeweils separat umgesetzt.

```
1  fi_recv(j_endpoint_get_endpoint(jendpoint),
       ↪(void*)&(message->header),
       ↪(size_t)sizeof(JMessageHeader), NULL, 0, NULL);
2  j_endpoint_read_completion_queue(
       ↪j_endpoint_get_completion_queue(jendpoint, FI_RECV), -1,
       ↪"RECEIVING", "JMessage Header");
3  j_message_ensure_size(message, j_message_length(message));
4
5  /**
6  * Schlüssel und Addressaustausch für RMA hier.
7  * Listing 3.5
8  */
9
10 fi_recv(j_endpoint_get_endpoint(jendpoint), message->data,
       ↪j_message_length(message), NULL, 0, NULL);
11 j_endpoint_read_completion_queue(
       ↪j_endpoint_get_completion_queue(jendpoint, FI_RECV), -1,
       ↪"Receiving", "JMessage");
12
13 message->current = message->data;
```

Listing 3.3: j_message_read, gekürzt

3.4. Hybride Datenübertragung aus RMA und Nachrichten

Zusätzlich zu den in Abschnitt 3.3 genutzten Funktionalitäten werden für die hybride Datenübertragung Funktionalitäten für RMA-Kommunikation verwendet. Die in Abschnitt 3.3 angeführten 3 Schritte müssen erweitert werden, um die Voraussetzungen für eine RMA-Datenübertragung zu schaffen:

- Registrierung der Speicherbereiche, aus denen Datenblöcke übertragen werden sollen, inklusive Generierung von Zugriffsschlüsseln, sofern zusätzliche Datenblöcke vorhanden sind. Dies ist nur auf der Quellseite notwendig.
- Übertragung des JMessage-Headers, welcher Metadaten über die JMessage und die Übertragung enthält und eine feste Größe besitzt.
- Übertragung des JMessage-Körpers von variabler Größe, welcher primär einige Daten für kleinere Übertragungen sowie einen Verweis auf zusätzliche potentielle Daten enthält.
- Übertragung der Zugriffsschlüssel an den Kommunikationspartner, sofern zusätzliche Datenblöcke vorhanden sind.
- Direkte Übertragung der Datenblöcke an den Kommunikationspartner, sofern zusätzliche Datenblöcke vorhanden sind.

Diese fünf Schritte sind für den Senden-Teil der Datenübertragung in eine Funktion (`j_message_write_rma`) gekapselt.
Aufgrund der Unterschiede in den Programmabläufen zwischen Client und Server werden im Empfangen-Teil nur die Schritte zwei, drei und vier in eine Funktion (`j_message_read`) übernommen. Der erste Schritt entfällt, da beim Empfangen keine Speicherregistrierung notwendig ist. Der letzte Schritt mit seinen jeweiligen Spezifikationen entfällt beim Empfang ebenfalls, da er in Client und Server direkt implementiert wird.

3.4.1. j_message_write_rma

Im Falle der hybriden Datenübertragung besteht diese Funktion aus der Umsetzung der 5 oben genannten Schritte. Zuerst wird überprüft, ob die zu übertragende JMessage auf zusätzliche Datenblöcke verweist (Listing 3.4 Zeile 6). Sollte das der Fall sein, wird für jeden Datenblock sequentiell ein 64-Bit Schlüssel generiert, der Datenblock in der libfabric-Domäne zur Lesefreigabe im Netzwerk registriert und anschließend die Anzahl der Schlüssel im Header der JMessage festgehalten (Listing 3.4 Zeile 6 bis 28).
Daraufhin wird der JMessage-Header und der JMessage-Körper, wie im nachrichtenbasierten Verfahren geschildert, übertragen (Listing 3.4 Zeile 30 und 31). Sofern Datenblöcke vorhanden sind, folgt die Übertragung der Struktur, welche die Schlüssel für den Speicherzugriff enthält. Da es sich auch hierbei um eine nachrichtenbasierte Übertragung handelt, wird gewartet, bis ein entsprechendes Completion Event generiert wurde.

Durch die Reihenfolge ihrer Ablage wird eine eindeutige Zuordnung der Speicherbereiche garantiert.
Um die Menge an Kopiervorgängen im Speicher des Gegenparts zu reduzieren, startet dieser mit den übertragenen Schlüsseln einen RMA-Lesezugriff auf die Speicherbereiche, welche die Datenblöcke enthalten. Der aufrufende Part wartet nun auf eine Nachricht des Gegenparts in der Kommunikation, dass der Lesevorgang abgeschlossen wurde (Listing 3.4 Zeile 42 bis 46). Sobald diese erfolgt ist, ist die Datenübertragung erfolgreich abgeschlossen. Die nun nicht mehr benötigten Schlüssel werden gelöscht und die freigegebenen Speicherbereiche dem Netzwerk entzogen.

```
1  GSList* mr_list = NULL;
2  GRand* key_generator = g_rand_new();
3  g_autoptr(JListIterator) iterator = NULL;
4
5  if (j_list_length(message->send_list) != 0)
6  {
7     guint counter = 0;
8     iterator = j_list_iterator_new(message->send_list);
9     message->header.key_number =
         ↪j_list_length(message->send_list);
10    message->header.data_chunks = TRUE;
11
12    message->key_buf = malloc(message->header.key_number *
         ↪sizeof(uint64_t));
13
14    while (j_list_iterator_next(iterator))
15    {
16       struct fid_mr* memory_region;
17       uint64_t key;
18       JMessageData* message_data =
            ↪j_list_iterator_get(iterator);
19       key = ((uint64_t)g_rand_int(key_generator)) << 32 |
            ↪(uint64_t)g_rand_int(key_generator);
20
21       fi_mr_reg(j_endpoint_get_domain(jendpoint),
            ↪message_data->data, message_data->length,
            ↪j_configuration_fi_get_mr_access(j_configuration()),
            ↪0,   key,
            ↪j_configuration_fi_get_mr_flags(j_configuration()),
            ↪&memory_region, NULL);
22
23       message->key_buf[counter] = fi_mr_key(memory_region);
24
```

```
25          mr_list = g_slist_prepend(mr_list,
                ↪(gpointer)memory_region);
26          counter++;
27      }
28  }
29
30  fi_send(j_endpoint_get_endpoint(jendpoint),
        ↪(void*)&(message->header), sizeof(JMessageHeader), NULL,
        ↪0, NULL);
31  j_endpoint_read_completion_queue(
        ↪j_endpoint_get_completion_queue(jendpoint, FI_TRANSMIT),
        ↪-1, "Transmit", "JMessage Header");
32
33  if (message->header.data_chunks)
34  {
35      fi_send(j_endpoint_get_endpoint(jendpoint),
            ↪(void*)message->key_buf, message->header.key_number *
            ↪sizeof(uint64_t), NULL, 0, NULL);
36      j_endpoint_read_completion_queue(
            ↪j_endpoint_get_completion_queue( jendpoint,
            ↪FI_TRANSMIT), -1, "Transmit", "key transmition");
37  }
38
39  fi_send(j_endpoint_get_endpoint(jendpoint), message->data,
        ↪(size_t)j_message_length(message), NULL, 0, NULL);
40  j_endpoint_read_completion_queue(
        ↪j_endpoint_get_completion_queue(jendpoint, FI_TRANSMIT),
        ↪-1, "Transmit", "JMessage");
41
42  if (message->header.data_chunks)
43  {
44      fi_recv(j_endpoint_get_endpoint(jendpoint),
            ↪(void*)wakeup_buf, sizeof(int), NULL, 0, NULL);
45      j_endpoint_read_completion_queue(
            ↪j_endpoint_get_completion_queue(jendpoint, FI_RECV),
            ↪-1, "Transmit", "JMessage list data completion
            ↪message");
46  }
```

Listing 3.4: j_message_write_rma, gekürzt

3.4.2. j_message_read

`j_message_read` des hybriden Modells basiert auf der nachrichtenbasierten Variante. Es wird lediglich nach dem Empfang des Headers der in Listing 3.5 beschriebene Empfang der für den RMA-Zugriff benötigten Schlüssel eingefügt (Listing 3.3 Zeile 6). Der erste Schritt entfällt, denn der empfangende Part muss keine eigene Schlüsselverwaltung aufbauen, sondern nur die des sendenden Parts entgegennehmen. Auch hier entfallen die letzten Schritte des reads auf diverse funktionelle Teile JULEAs und sind nicht generalisiert.

```
1 if (message->header.data_chunks)
2 {
3     message->key_buf = malloc(message->header.key_number *
        ↪sizeof(uint64_t));
4     fi_recv(j_endpoint_get_endpoint(jendpoint),
        ↪(void*)message->key_buf, message->header.key_number *
        ↪sizeof(uint64_t), NULL, 0, NULL);
5     j_endpoint_read_completion_queue(
        ↪j_endpoint_get_completion_queue(jendpoint, FI_RECV),
        ↪-1, "Receiving", "key_buf");
6 }
```

Listing 3.5: j_message_read, gekürzt

3.5. Anpassungen an den verbs Provider

Libfabric definiert ein breites Spektrum an Funktionalitäten zur Unterstützung verschiedener Kommunikationstypen. Der sockets Provider deckt den gesamten Bereich der libfabric-API ab, während andere Provider nur ein Subset dieser API benötigen und dementsprechend eigene Voraussetzungen an die Anwendung stellen. Daher müssen bei einem Providerwechsel einige Programmteile an die durch die Bitmaske, welche das mode-Feld in der zurückgegebenen `fi_info` enthält, kommunizierten Voraussetzungen der einzelnen Provider angepasst werden. Das mode-Feld des verbs Provider setzt vier Bits: `FI_MR_VIRT_ADDR`, `FI_MR_ALLOCATED`, `FI_MR_PROV_KEY` und `FI_MR_LOCAL`. Die benötigten Anpassungen an den verbs Provider werden im Anschluss umrissen.

FI_MR_VIRT_ADDR

`FI_MR_VIRT_ADDR` bedeutet, dass Aufrufe, welche eine entfernte Speicherstelle adressieren (bspw. RMA), diese durch ihre virtuelle Adresse und nicht durch einen 0 basierten Offset ansprechen. Hierfür müssen bei der hybriden Variante neben dem Schlüssel die Adressen an den Kommunikationspartner übertragen werden. Da mit der Schlüsselübertragung schon ein entsprechender Mechanismus vorhanden ist, wird dieser für eine solche Übertragung genutzt. Jeder zweite Eintrag in dem vormaligen Schlüsselarray wird nun

für eine Adresse genutzt, welcher mit seinem Vorläufer gepaart die Informationen für die aktuelle Übertragung bereithält (siehe Listing 3.6 Zeile 10 und 11).

```
1  message->key_addr_buf =
       ↪malloc(message->header.key_addr_number *
       ↪sizeof(uint64_t) * 2);
2
3  while (j_list_iterator_next(iterator))
4  {
5      struct fid_mr* memory_region;
6      JMessageData* message_data = j_list_iterator_get(iterator);
7
8      fi_mr_reg(j_endpoint_get_domain(jendpoint),
           ↪message_data->data, message_data->length,
           ↪j_configuration_fi_get_mr_access(j_configuration()),
           ↪0, 0,
           ↪j_configuration_fi_get_mr_flags(j_configuration()),
           ↪&memory_region, NULL);
9
10     message->key_addr_buf[counter * 2] =
           ↪fi_mr_key(memory_region);
11     message->key_addr_buf[counter * 2 + 1] =
           ↪&message_data->data;
12
13     mr_list = g_slist_prepend(mr_list,
           ↪(gpointer)memory_region);
14     counter++;
15 }
```

Listing 3.6: j_message_write_rma verbs Schlüssel- und Addressübertragung, gekürzt

FI_MR_ALLOCATED

`FI_MR_ALLOCATED` bedeutet, dass alle registrierten Speicherbereiche durch zuvor allozierten Speicher gestützt werden müssen. Dies ist zuvor schon der Fall gewesen, daher sind hier keine Anpassungen notwendig.

FI_MR_PROV_KEY

`FI_MR_PROV_KEY` bedeutet, dass der Provider den Schlüssel für Zugriffe auf registrierte Speicherregionen selber erzeugt und durch die Anwendung gesetzte Schlüssel werden ignoriert. Schlüssel werden durch `fi_mr_key` geholt. Da dies bereits der Fall ist (Listing 3.4 Zeile 23), muss nur die Schlüsselgenerierung aus Listing 3.4 entfernt werden.

FI_MR_LOCAL

`FI_MR_LOCAL` bedeutet, dass alle Buffer, welche von Datentransferfunktionen genutzt werden, registriert werden müssen. Hierzu muss der entsprechenden Funktion der zugehörige Memory Descriptor, welcher durch `fi_mr_desc` erlangt werden kann, übergeben werden (siehe Listing 3.7 Zeile 3).

```
1 fi_mr_reg(j_endpoint_get_domain(jendpoint),
    ↪message_data->header, (size_t)sizeof(JMessageHeader),
    ↪j_configuration_fi_get_mr_access(j_configuration()), 0,
    ↪0, j_configuration_fi_get_mr_flags(j_configuration()),
    ↪&memory_region, NULL);
2
3 fi_recv(j_endpoint_get_endpoint(jendpoint),
    ↪(void*)&(message->header),
    ↪(size_t)sizeof(JMessageHeader),
    ↪fi_mr_desc(memory_region), 0, NULL);
4 j_endpoint_read_completion_queue(
    ↪j_endpoint_get_completion_queue(jendpoint, FI_RECV), -1,
    ↪"RECEIVING", "JMessage Header");
```

Listing 3.7: Exemplarischer fi_recv Aufruf für einen JMesssage Header mit memory descriptor

4. Analyse und Ergebnisse

In diesem Kapitel wird sowohl das Analysevorgehen und die -umgebung beschrieben, als auch die Ergebnisse der Analyse aufbereitet präsentiert.

4.1. Analysevorgehen

JULEA bietet ein eigenes Shellskript namens `benchmark.sh` [Kuh+20, Datei `benchmark.sh`], welches für gewählte Funktionalitäten einen Benchmark durchführt. Die Laufzeit der gemessenen Funktionalitäten kann festgelegt werden und wird für die Messung auf 120 Sekunden gesetzt. Das Benchmarkskript misst, wie viel Zeit in einem Benchmark verbracht wird, wie viel Zeit insgesamt für einen einzelnen Benchmark benötigt wurde (inklusive Setup), wie viele Operationen pro Sekunde im Schnitt durchgeführt wurden und gegebenenfalls die Anzahl der übertragenen Bytes pro Sekunde im Schnitt. Über die Anzahl der Operationen pro Sekunde und die Anzahl der übertragenen Bytes pro Sekunde lässt sich demzufolge ein Vergleich in Bezug auf die Performanz anstellen.
Für die Datenerhebung wird dieses Skript verwendet. Daran schließt sich ein Vergleich der Leistungsdaten der Funktionalitäten, welche das Netzwerk verwenden, an.

Als Basislinie wird die Leistung der bisherigen Implementation auf der Testumgebung gemessen. Daraufhin werden die Benchmarks für den sockets, und den verbs Provider durchgeführt, sowohl in der nachrichtenbasierten als auch in der hybriden Variante. Die Benchmarks werden in der beschriebenen Konfiguration für die datenübertragenden Funktionalitäten des item-, des object- und des HDF5-Clienten jeweils zehnfach durchgeführt. Anschließend wird der Mittelwert zwischen den einzelnen Messungen gebildet und eine Standardabweichung von diesem berechnet.
Als Backends für den object Server wird POSIX, für den key value Server LevelDB und für den db Server SQLite verwendet.

4.1.1. Benchmark-Umgebung

Die Testumgebung ist der Cluster des Arbeitsbereichs Wissenschaftliches Rechnen der Universität Hamburg [Arb]. Die Benchmarks werden auf den abu-Knoten durchgeführt, da sie nicht nur über Gigabit-Ethernet verbunden sind, sondern auch über ein 40 Gigabit-Infiniband als Netzwerkhardware verfügen.
Die Spezifikationen des Clusters sind in Tabelle 4.1 zu finden.

Partitionsname	Knoten	Kerne/ Knoten	verfügbare Netzwerke	RAM in GB
abu	5	48	1 Gigabit Ethernet 40 Gigabit Infiniband	256[1] 128[2-5]
amd	3	24	1 Gigabit Ethernet	12
magny	1	48	1 Gigabit Ethernet	128
nehalem	1	8	1 Gigabit Ethernet	12
west	10	24	1 Gigabit Ethernet	12

Tabelle 4.1.: Cluster Universität Hamburg Arbeitsbereich wissenschaftliches Rechnen Spezifikationen

4.1.2. Benchmark-Funktionalitäten

In den Benchmarks werden die Funktionalitäten von drei JULEA-Clients im Bezug auf Netzwerkkommunikation gemessen. Diese sind der item-Client (Abbildung 4.1), der object-Client (Abbildung 4.2) und der HDF5-Client (Abbildung 4.3 und 4.4). Sowohl der item-, als auch der object-Client messen das Schreiben (write) und Lesen (read) von Daten vom jeweiligen Server. Dies gilt sowohl für die relativ kleinen Datenmengen pro Übertragung als auch für die wesentlich größeren batches. Der object-Client bietet die Möglichkeit die einzelnen Objekte zentral oder in einzelne Teile zerlegt (distributed object) zu speichern, dementsprechend werden beide Optionen gemessen.
Die Benchmarks durchlaufen eine Schleife, bis die Zeitbeschränkung erreicht wird, und benutzen eine Datenblockgröße von 4 KB pro Übertragung für die item und object Benchmarks. Im Fall der read und write Benchmarks werden die jeweiligen items oder objects nach dem Anlegen der Datenblöcke in ihnen direkt für das Messen der Netzwerkkommunikation genutzt. Es werden 1.000 Operationen pro Benchmarkiteration durchlaufen.
Im Falle der batch Messungen werden 10.000 Operationen für eine Übertragung in ein zu übertragende Datengruppierung (batch) zusammengefasst und im Anschluss für die Messungen der Netzwerkkommunikation genutzt. Dies resultiert in einer Gesamtgröße von 40 MB pro batch.
Der HDF5-Client wird für Übertragungsgrößen von sowohl 4 Kibibyte (dataset4K) als auch von 4 Mebibyte (dataset4M) in Bezug auf das Schreiben und Lesen bemessen. Des weiteren wird das Schreiben und Lesen der HDF5 attributes gemessen. Die Datenmenge hierbei liegt im unteren bis mittleren Kilobytebereich.

4.2. Ergebnisse

Da es sich um Netzwerkanalyse handelt, soll das primäre Augenmerk auf den übertragenen Bytes pro Sekunde liegen. Durch die Länge der Messung können Aussagen über die mittlere Übertragungsperformanz gemacht werden.

4.2.1. GIO und libfabric sockets

Die Abbildungen 4.1, 4.2, 4.3 und 4.4 zeigen, dass die bisherige GIO sockstream Variante im Vergleich zu der libfabric-sockets Variante bei geringeren Datenmengen pro Übertragung deutlich mehr Bytes pro Sekunde über das Netzwerk überträgt. Am deutlichsten ist dieses Verhältnis bei den Funktionalitäten mit geringer Größe der einzelnen Datenübertragungen, einfaches Lesen und Schreiben. Im Vergleich hierzu nähern sich die Verhältnisse zwischen GIO und den libfabric-sockets an, sobald die versendete Datenmenge pro Kommunikation größer wird. Ähnliches kann bei den Ergebnissen der HDF5 Benchmarks beobachtet werden.

item-Client Benchmarks

Abbildung 4.1 stellt den Durchschnitt der Ergebnisse der Benchmarks des item-Clienten im Bezug auf die Datendurchsatzrate der GIO und beider libfabric Varianten unter Nutzung des sockets Providers dar. Eine auf KB/s gerundete Version der gemessenen Werte findet sich in Tabelle B.1.
Es ist zu sehen, dass die Ergebnisse der jeweiligen read und write Benchmarks sich gleichen. Dies ist der Fall für die GIO und der nachrichtenbasierten sockets Variante sowie für die entsprechenden batch Benchmarks. Die hybride Variante der write-batch Messungen fällt im Vergleich zu ihren read-batch Messungen ein wenig ab im Bezug auf die gemessene Datendurchsatzrate.
Werden die Ergebnisse eines einzelnen Benchmarks verglichen, fällt auf, dass die GIO Variante wesentlich höhere Datendurchsatzraten besitzt, als die beiden libfabric Varianten, welche sich in ihren Messungsergebnissen gleichen. Letzteres gilt abgesehen von den bereits erwähnten write-batch Benchmarks, hier bleibt die hybride Variante hinter der nachrichtenbasierten zurück.

Die write und read Benchmarks der GIO Variante liegen bei ca. 21 MB/s und ihre Standardabweichung beträgt 2,2 MB/s, sie liegt also bei 10,5%. Die entsprechenden libfabric Messungen ergaben Ergebnisse zwischen ca. 2,2 MB/s und 2,35 MB/s, während die zugehörige Standardabweichung bei maximal 3,1% liegt. Dies ergibt ein Verhältnis von ca 9:1 der Datendurchsatzraten der GIO Variante zu den libfabric Varianten.

Die write-batch und read-batch Benchmarks der GIO Variante liegen bei ca 110 MB/s und die Standardabweichung beträgt um 0,1%. Die Ergebnisse der nachrichtenbasierten Variante betragen gerundet 15,3 MB/s mit einer Standardabweichung von 5,6% für die read-batch Benchmarks und ca. 16,2 MB/s mit einer Standardabweichung von 2,5% für die write-batch Benchmarks. In der hybriden Variante resultierten die read-batch Benchmarks in einer Übertragungsrate von gerundet 15 MB/s mit einer Standardabweichung von 4,3% und die write-batch Benchmarks von 12,6 MB/s mit einer Standardabweichung von 2,7%. Dies resultiert in Verhältnissen von 8,7:1 bis 6,8:1 von den Ergebnissen der GIO Variante zu den Ergebnissen der libfabric Varianten.

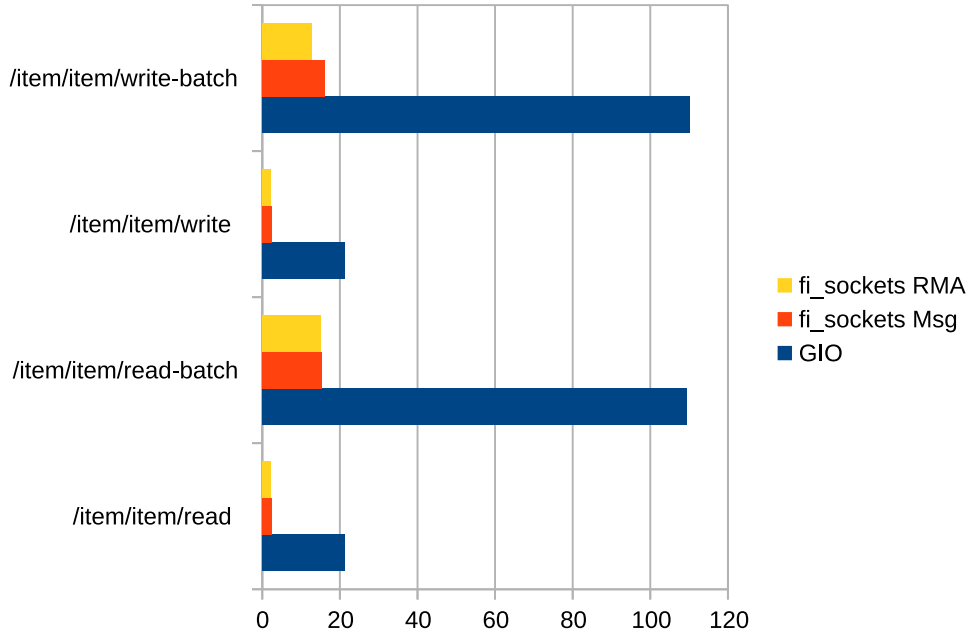

Abb. 4.1.: Durchschnittliche Ergebnisse der item-Benchmarks im Vergleich der übertragenen MB/s

object-Client Benchmarks

Abbildung 4.2 zeigt die Ergebnisse der Benchmarks des object-Clients im Bezug auf die Datendurchsatzrate der GIO und beider libfabric Varianten unter Nutzung des socket Providers. Eine auf KB/s gerundete Version der gemessenen Werte findet sich in Tabelle B.3.
Die generellen Beobachtungen gleichen jenen über die Ergebnisse des item-Clients. Zwischen distributed- und non-distributed-object existieren keine relevanten Unterschiede bezüglich der Ergebnisse der jeweiligen Benchmarkdurchschnitte.

Die read und write Benchmarks der GIO Variante liegen bei rund 21 MB/s mit einer Standardabweichung von 10,6%. Die entsprechenden Messungen der libfabric Varianten unter Verwendung des socket Providers liegen zwischen 2,36 MB/s (nachrichtenbasierte Variante) und 2,21 MB/s (hybride Variante) mit einer Standardabweichung von maximal 3,5%. Dies resultiert in einem Verhältnis von 8,9:1 bis 9,5:1 von den Ergebnissen der GIO Variante zu den Ergebnissen der libfabric Varianten.

Die Ergebnisse der read-batch und write-batch Benchmarks der GIO Variante liegen bei ungefähr 110 MB/s mit einer Standardabweichung von bis zu 0,15%. Die entsprechenden Messungen der nachrichtenbasierten libfabric Variante betragen 15,3 MB/s für die read-batch Benchmarks mit einer Standardabweichung von 4,9% und 16,2 MB/s für die write-batch Benchmarks mit einer Standardabweichung von 2,1%. In der hybriden

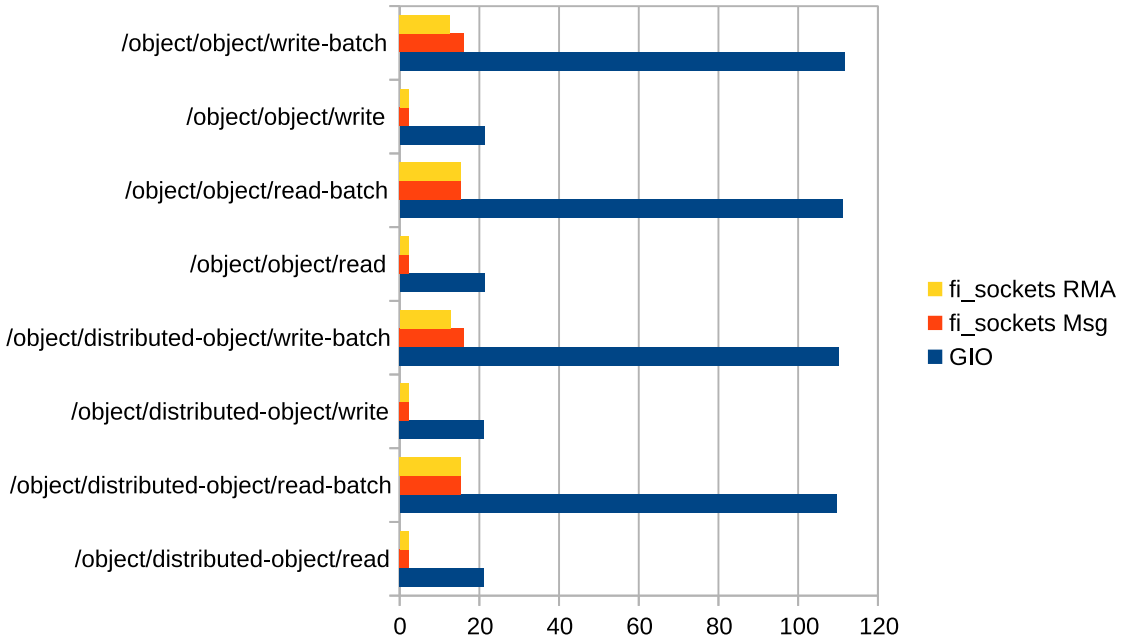

Abb. 4.2.: Durchschnittliche Ergebnisse der object und distributed-object Benchmarks im Vergleich der übertragenen MB/s

Variante betragen die Ergebnisse der read-batch Benchmarks ungefähr 15,3 MB/s mit einer Standardabweichung von 5,2% und diejenigen der write-batch Benchmarks 12,5 MB/s mit einer Standardabweichung von 3%. Dies resultiert in einem Verhältnis der GIO Variante zu den libfabric Varianten unter Verwendung des socket Providers zwischen 8,8:1 und 6,8:1. Wie bei den Ergebnissen des item-Clients fällt auf, dass in den write-batch Benchmarks die hybride Variante der nachrichtenbasierten überlegen ist.

HDF5-Client Benchmarks

In den Abbildungen 4.3 und 4.4 werden die durchschnittlichen Ergebnisse der HDF5-Benchmarks der GIO Variante und der beiden libfabric Varianten unter Verwendung des socket Providers dargestellt. Eine auf KB/s gerundete Version der gemessenen Werte findet sich in Tabelle B.4.
Abbildung 4.3 zeigt die Ergebnisse bezüglich kleinerer Datenmengen pro Übertragungsvorgang. Auch hier ist zu beobachten, dass die GIO Variante wesentlich größere Datendurchsatzraten aufweist als die beiden libfabric Varianten. Bei der GIO Variante sind die dataset4K-read Benchmarks die durchsatzstärksten, die attribute-write Benchmarks die durchsatzschwächsten, weisen allerdings eine deutlich höhere mittlere Durchsatzrate auf als die libfabric Varianten. Die Ergebnisse der libfabric Varianten gleichen sich sehr mit Ausnahme derjenigen der attribute write Benchmarks, welche ein wenig größere Datendurchsatzraten aufweisen.

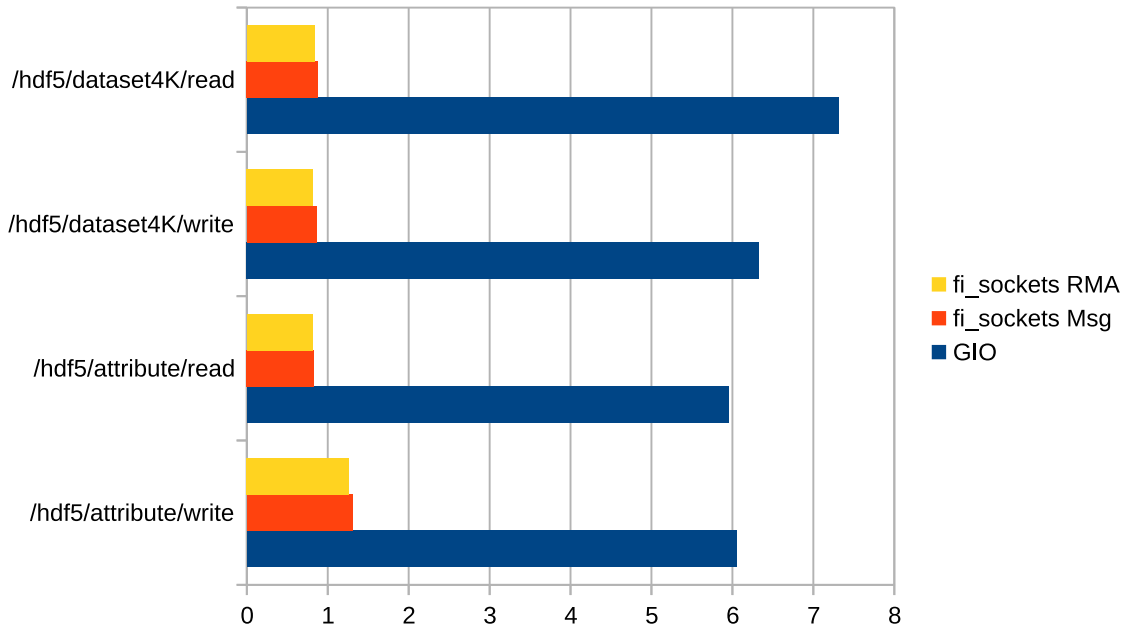

Abb. 4.3.: Durchschnittliche Ergebnisse der HDF5 dataset4K und attribute Benchmarks im Vergleich bezüglich der übertragenem MB/s

Die attribute read und write Benchmarks der GIO Variante weisen Messungen von etwa 6 MB/s mit einer Standardabweichung von 23,33% im write Fall und 11,67% im read Fall auf. Die Messungen der libfabric Varianten zeigen Datendurchsatzraten von 1,3 MB/s mit einer Standardabweichung von maximal 1,1% im Falle der write Benchmarks und 0,8 MB/s mit einer Standardabweichung von maximal 0,6% im Falle der read Benchmarks. Dies führt zu Verhältnissen der GIO Variante zu den libfabric Varianten von 4,6:1 im Falle der write Benchmarks und 7,5:1 im Falle der read Benchmarks.
Die dataset4K read Benchmarks der GIO Variante weisen eine durchschnittliche Durchsatzrate von 7,3 MB/s mit einer Standardabweichung von 8,9% auf, während diejenigen der write Benchmarks eine Durchsatzrate von 6,3 MB/s mit einer Standardabweichung von 10,8% aufweisen. Die Messungen der dataset4K Benchmarks der libfabric Varianten gleichen sich sehr und weisen eine Durchsatzrate von etwa 0,85 MB/s mit einer Standardabweichung von maximal 2,3% auf. Dies führt zu Verhältnissen der GIO Variante zu den libfabric Varianten von 8,6:1 im Falle der read Benchmarks und 7,4:1 im Falle der read Benchmarks.

In Abbildung 4.4 werden die Ergebnisse bezüglich größerer Datenmengen pro Übertragungsvorgang gezeigt. Die GIO Variante besitzt auch hier eine größere Datendurchsatzrate als die beiden libfabric Varianten. Allerdings ist der relative Unterschied zwischen der GIO Variante und den libfabric Varianten wesentlich kleiner als bei den anderen Benchmarks. Die Fälle read und write gleichen sich in der GIO Variante, während die Benchmarks des write Falls in den libfabric Varianten eine etwas kleinere Durchsatzrate

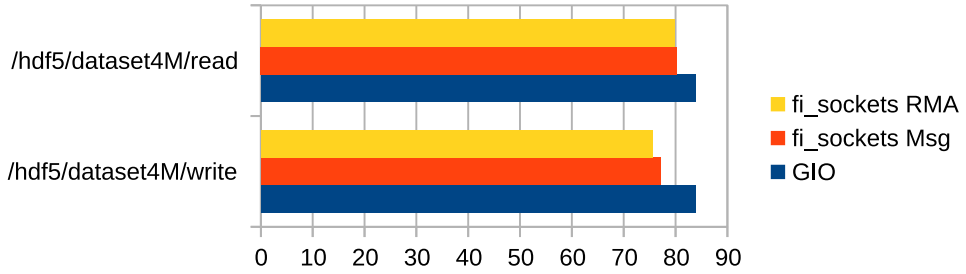

Abb. 4.4.: Durchschnittliche Ergebnisse der hdf5/dataset4M Benchmarks im Vergleich bezüglich der übertragenem MB/s

besitzen, als die Benchmarks des read Falles.

Die read und write Benchmarks der GIO Variante haben eine Durchsatzrate von 84 MB/s mit einer Standardabweichung von 3% im Fall der read Benchmarks und von 2,3% im Fall der write Benchmarks. Die read Benchmarks der libfabric Varianten ähneln sich und besitzen eine Durchsatzrate von ungefähr 80 MB/s mit einer Standardabweichung von 2,4%. Die write Benchmarks der nachrichtenbasierten libfabric Variante besitzen eine Durchsatzrate von 77 MB/s mit einer Standardabweichung von 1,3% während diejenigen der hybriden Variante eine Durchsatzrate von 75 MB/s mit einer Standardabweichung von 1,5% besitzen. Dies führt zu Verhältnissen von GIO zu libfabric Varianten von 1,05:1 bis 1,12:1.

Unoptimierte GIO Variante

In den Abbildungen 4.5, 4.6 und 4.7 sieht man exemplarisch Ergebnisse aus den Benchmarks mit und ohne Versenden von batches und HDF5, wenn die in Abschnitt 2.1.1 beschriebenen Optimierungen (Nagle Algorithmus und Cork) für die durch die GIO Variante genutzten sockets nicht genutzt werden.

In Abbildung 4.5 werden die Ergebnisse der Messungen der unoptimierten GIO Variante bei kleinen Datenübertragungen, wie den normalen JULEA reads und writes vertreten durch den Item-Clienten, dargestellt. Diese liegen hier im Bereich von 93 KB/s bei einer Standardabweichung von 0,001%. Um diesen Wert liegen auch die Ergebnisse des object-Clienten, sowohl für die distributed-object als auch dessen normale Variante. Eine auf KB/s gerundete Version der gemessenen Werte findet sich in Tabelle B.5. Wie zuvor erwähnt, befinden sich die Werte der nachrichtenbasierten Variante um 2,35 MB/s und für die hybride Variante bei 2,2 MB/s. Ähnliche Ergebnisse lassen sich für die ebenfalls relativ kleinen Datenmengen pro Übertragung von den HDF5 dataset4k und attribute Benchmarks feststellen.
Die Verhältnisse bei den HDF5 Benchmarks von geringerer Datenmenge pro Übertragung von GIO unoptimiert zu der nachrichtenbasierter sockets Variante schwankt zwischen 1:18 und 1:50.

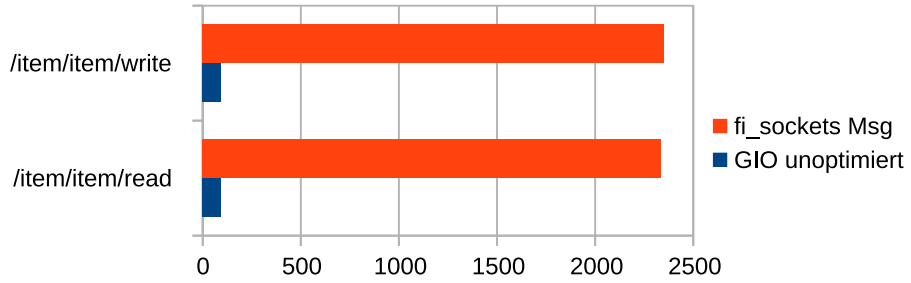

Abb. 4.5.: Unoptimierte GIO Variante der item read und write Benchmarks repräsentativ für Benchmarks mit kleinen Datenmengen pro Datenübertragung gegen entsprechende nachrichtenbasierte libfabric Benchmarks in KB/s

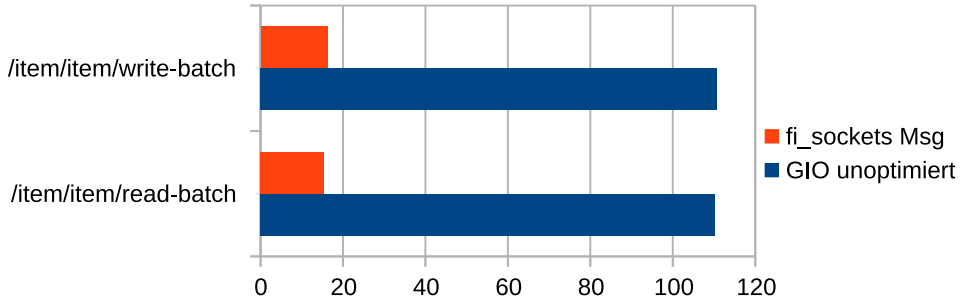

Abb. 4.6.: Unoptimierte GIO Variante der item read-batch und write-batch Benchmarks repräsentativ für Benchmarks mit größeren batches zur Datenübertragung gegen entsprechende nachrichtenbasierte libfabric Benchmarks in MB/s

Währenddessen weisen die Ergebnisse der übrigen reads und writes ein Verhältnis von ca. 1:25 auf.

In Abbildung 4.6 werden die Ergebnisse der Messungen der unoptimierten GIO Variante bei großen Datenübertragungen dargestellt, welche in batches organisiert sind, repräsentiert durch die read-batch und write-batch Benchmarks des item-Clienten. Diese liegen im Bereich von gerundet 110 MB/s, dem Bereich der unoptimierten Variante. Hier beträgt die Standardabweichung 0,09% im Falle der write Benchmarks und 0,05% im Falle der read Benchmarks. Die Ergebnisse der beiden Varianten des object-Clienten liegen ebenfalls in diesem Bereich, während die Messungen der entsprechenden narichtenbasierten libfabric Variante zwischen 3,6 und 4 MB/s liegen. Dies führt zu Verhältnissen der unoptimierten GIO Variante zu der nachrichtenbasierten libfabric Variante von 27:1 bis 31:1.

In Abbildung 4.7 werden die Ergebnisse der Messungen der unoptimierten GIO Variante bei den 4 Megabyte großen Datenübertragungen dargestellt. Diese liegen bei 30,7 MB/s mit einer Standardabweichung von 1,2% für writes und bei 19,5 MB/s mit einer

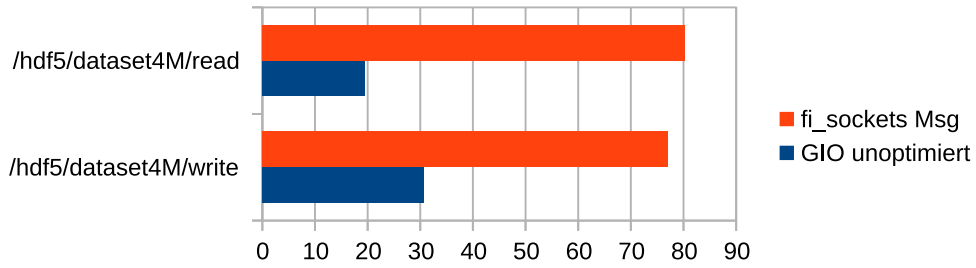

Abb. 4.7.: Unoptimierte GIO Variante hdf5/dataset4M read und write Benchmarks gegen entsprechende nachrichtenbasierte libfabric Benchmarks in MB/s

Standardabweichung von 0,9% für reads. Die Ergebnisse der entsprechenden nachrichtenbasierten libfabric Variante liegen für writes bei 77 MB/s und für reads bei 80 MB/s. Daraus resultieren Verhältnisse von GIO unoptimierte Variante zur nachrichtenbasierten libfabric Variante von 1:2,5 für die writes, während das Verhältnis bei den reads bei 1:4 liegt.

4.2.2. libfabric verbs

Bei der Umstellung auf den libfabric verbs Provider ist nach den in Abschnitt 3.5 geschilderten Anpassungen ein bisher ungelöstes Problem aufgetreten. Auf der Event Queue des Clients liegt nach dem Verbindungsversuch auf dem Infiniband Netzwerk ein Error Event, welches einen Fehler mit der Bezeichnung `Unknown Error -8` enthält.
Dieser Fehler wird durch das `fi_strerror` Kommandozeilentool libfabrics als „Exec format error“ identifiziert, welcher aber im eigentlichen libfabric Code [Ope20a, Datei `fi_errno.h`] als Definition auskommentiert wurde. Mit dem Server wird nicht interagiert, er registriert auch keinen fehlerhaften Versuch eines Verbindungsaufbaus.
Bis kurz vor Ende der Arbeit konnte dieses Problem nicht behoben werden und somit konnten keine Messungen vorgenommen werden.

5. Diskussion

5.1. Interpretation

Die Messungen bestätigen für den sockets Provider die in Abschnitt 2.3.2 Segment sockets erwähnte geringere Leistung des sockets Provider im Vergleich zu einer direkten socket-Implementation. Das Ausmaß ist vor allem bei dem Versenden kleinerer Datenmengen pro Übertragung bemerkenswert. Die Datendurchsatzraten gleichen sich bei der Versendung größerer Datenpakete an (siehe Abbildung 4.4), was auf einen größeren Overhead in der libfabric Kommunikation zurückzuführen ist. Dies ist allerdings nur der Fall, sofern Optimierungen wie Cork in Kombination mit der Ausschaltung des Nagle Algorithmus wie in Abschnitt 2.1.1 erläutert für die direkte socket-Implementation genutzt wurden. Die schlechte Performanz der libfabric Versionen im Falle der eigentlich größere Datenmengen enthaltenden batches stellen eine Überraschung dar, da ein Overhead bei größeren Datenmengen weniger stark ins Gewicht fallen sollte. Eine wahrscheinliche Ursache hierfür stellt die direkte Integration dar und wird in Abschnitt 5.2.2 beleuchtet.
Aufgrund der in Unterabschnitt 4.2.2 erwähnten, bisher ungeklärten Probleme konnten keine Messungen bezüglich des verbs Providers durchgeführt werden, was auf zusätzliche Probleme bei einem weiteren gewünschten Providerwechsel schließen lässt. Die geweckte Erwartungshaltung eines relativ einfachen Providerwechsels, sofern die kommunizierten Voraussetzungen erfüllt werden, wird von libfabric nicht vollständig erfüllt.

5.2. Mögliche Ursachen

Allgemein sind für die Performanzunterschiede drei Hauptfaktoren zu identifizieren. Zum einen ist zu nennen, dass die direkte Integration der libfabric Varianten in das JULEA Kommunikationsmodell ohne dieses auch an libfabrics Design anzupassen dazu führt, dass auf der libfabric Seite für die Performanz nachteilige Kompromisse eingegangen werden mussten. Außerdem konnte an einigen Stellen mangels Erfahrung mit dem Umgang mit libfabric das Potential des Frameworks nicht ausgeschöpft werden. Eine weitere mögliche Ursache ist, dass die GIO Variante durch ihre Interaktion mit einer direkten socket-Implementation durch diese bereitgestellte Optimierungen (wie beispielsweise Corking mit deaktiviertem Nagle Algorithmus) nutzen kann. Solche Optimierungen müssten bei der Nutzung von libfabric die Anwendungen selber durchführen.

Im Folgenden wird auf die größten Auffälligkeiten in den Ergebnissen im Einzelnen eingegangen.

5.2.1. GIO Versionen

Der Unterschied zwischen der unoptimierten und der optimierten Version von GIO ist der Funktionsweise des Nagle Algorithmus und Corking geschuldet. Beide Ansätze beschäftigen sich mit kleineren Datenpaketen, was die ähnlichen Ergebnisse bei den batch Benchmarks erklärt. Bei kleineren Datenmengen verlangt der Nagle Algorithmus, wie in Abschnitt 2.1.1 erläutert, Bestätigung vom Kommunikationspartner, was beim Corking nicht der Fall ist. Außerdem ist das Corking aggressiver beim eventuellen Zusammenführen größerer Datenpakete, was zu einer höheren Netzwerkauslastung führt.

5.2.2. sockets Provider

Geringe Performanz bei geringen Datenmengen

Die wahrscheinlichste Ursache für geringe Performanz bei geringen Datenmengen pro Übertragung, wie sie bei den read und write Benchmarks oder den hdf5 attribute Benchmarks zu finden ist, ist der Overhead, welchen libfabric Übertragungen im Vergleich zu direkten socket-Implementationen (siehe [Ope20b, Eintrag `fi_sockets(7)`]) mit sich bringen.
Da das Verhältnis von Kommunikationsoverhead zu Datenmenge hier besonders gering ist, schlägt der Overhead sich stark im Ergebnis nieder.

Angleichende Performanz bei größeren Datenblöcken

In den Ergebnissen der hdf5/dataset4M read und write Benchmarks (siehe Abbildung 4.4) ist zu sehen, dass sich bei zunehmender Größe der zu versendenden Datenblöcke die Performanz zwischen GIO und den libfabric Varianten angleicht. Dies ist dadurch zu erklären, dass der zuvor erwähnte Overhead bei größeren zu übertragenen Datenblöcken weniger ins Gewicht fällt.

Geringe Performanz bei batches

Da die Gesamtdatenmenge eines batches größer ist als die Datenmenge der HDF5 Blöcke in dataset4M wäre zu erwarten, dass sich die Performanz hier weiterhin angleicht. Das Ausbleiben einer solchen Angleichung bei den eigentlich größeren batches ist durch eine Schwachstelle bei der direkten Integration der libfabric Kommunikation in JULEA begründet. Die Größe der batches resultiert nicht aus größeren Datenblöcken, sondern aus der Anzahl der an die jeweilige Nachricht angehängten Datenblöcken. Diese stellen je eine Übertragung in libfabric dar. In Listing 3.2 ist außerdem zu sehen, dass das Senden jedes einzelnen dieser Blöcke miteinander synchronisiert wird, da der nächstfolgende Block erst mit dem Senden beginnen darf, sobald der letzte bestätigt, vollkommen auf das Netzwerk geschrieben worden zu sein. Ein ähnlicher Mechanismus findet sich bei dem Lesen der per RMA zu übertragenen Speicherbereiche im Kommunikationspartner. Dies resultiert in vielen kleinen Verzögerungen im Programmablauf.
Hierfür bieten sich zwei Lösungsansätze an. Zum einen könnte man ähnlich dem Corking

kleinere Nachrichten zu einem größeren Nachrichtenblock zusammenfassen und diesen verschicken, zum anderen wäre es möglich, eine gewisse Asynchronität bei der Übertagung der Daten zuzulassen und die Event Queue des sendenden Endpoints erst im Nachhinein zu lesen und auf Fehler zu prüfen.

Hybride Version gegen nachrichtenbasierte Version

In den einzelnen Ergebnissen fällt auf, dass sich die read Benchmarks der beiden libfabric Varianten im Ergebnis sehr gleichen, während bei den write Varianten der nachrichtenbasierte Ansatz einen Performanzvorsprung aufweist (siehe Abbildungen 4.1, 4.2, 4.3, 4.4). Eine Erklärung hierfür liegt in der identischen Funktionsweise der beiden Varianten und der gleichen Sequenzialität beim Versenden der Datenblöcke in Kombination damit, dass die RMA Zugriffe durch den sockets Provider emuliert werden [Ope20b, Eintrag `fi_sockets(7)`].
Für den Unterschied bei den write Benchmarks ist die Kombination aus zwei Punkten am wahrscheinlichsten. Zum einen fällt ins Gewicht, dass die RMA Zugriffe durch den sockets Provider emuliert werden und somit keinen Performanzgewinn darstellen. Zum anderen muss durch die benötigte Freigabe der zu lesenden Datenblöcke in der hybriden Variante auf den Abschluss des Kommunikationspartners gewartet werden, während die nachrichtenbasierte Variante nach dem Versenden des letzten Blocks direkt in den nächsten Benchmarkdurchlauf gehen kann. Außerdem ist zu nennen, dass mögliche Performanzvorteile durch den zuvor erwähnten Verzögerungsmechanismus überdeckt werden.

Unterschiedliche Standardabweichungen

Wenn man sich die Standardabweichungen der einzelnen Ergebnisse ansieht, fällt auf, dass die meisten Standardabweichungen der GIO Variante bei kleinen Datenmengen um die 10% betragen, während sie bei großen Datenmengen weniger als 1% betragen. Die entsprechenden Standardabweichungen der libfabric Varianten bleiben hingegen konstant zwischen etwa 2 und 5%.
Eine gewisse Varianz ist bei Netzwerkverkehr aufgrund der Problematiken auf den unteren Ebenen, welche in Unterabschnitt 2.1.2 beschrieben sind, zu erwarten. Eine mögliche Erklärung für diese Werte besteht in dem Abhängigkeitsgrad der einzelnen Vorkommnisse von der durch die Anwendung genutzten CPU und damit deren Scheduling. Libfabric als komplettes Framework bringt einen gewissen CPU Overhead mit sich, was somit deren zwischen den einzelnen Benchmarks relativ konstante Varianz erklärt. Außerdem sind die einzelnen Schritte stark sequenzialisiert.
Die Unterschiede bei der GIO Variante lassen sich über das zusätzliche Spiel, welches bei kleinen Datenmengen durch die Funktionsweise des Corkings hinzukommt, erklären.

5.3. Limitationen

Während der Arbeit sind diverse Limitationen in Erscheinung getreten.

Primär ist hier der Fehler bei dem Verbindungsaufbau des verbs Providers unbekannter Ursache, welcher in Unterabschnitt 4.2.2 geschildert wurde, zu nennen.

Eine andere Limitation stellt die Übernahme des Kommunikationsmodells von JULEA dar, welches das Versenden vieler kleiner Nachrichten beinhaltet. Dies wird durch die Optionskombination `TCP_CORK` und `TCP_NODELAY` (siehe Abschnitt 2.1.1 Cork und Nagles Algorithmus) der direkten socket-Implementation, auf welche GIO zugreift, optimiert, während libfabric solche Optionen nicht bietet und solche Funktionalitäten durch die Anwendung durchführen lassen würde. Allerdings würden mehrere vollständige Neuentwürfe des Kommunikationsmodelles und deren Implementation den Rahmen einer solchen Arbeit übersteigen.

Eine weitere Limitation stellt die Tatsache dar, dass libfabric zwar eine größere Zahl an Providern besitzt, allerdings viele davon spezialisiert sind und kein Zugriff auf entsprechende Testhardware möglich war.

Auch ist zu nennen, dass es sich bei libfabric um ein komplexes Framework handelt, dessen Dokumentation nicht überall dem aktuellen Stand angepasst wurde.

5.4. Weiterführende Arbeit

Um ein gesamtes Bild zu liefern, wäre die Integration von verschiedenen Netzwerkframeworks wie UCX [Sha+15], CCI [Atc+11] oder Portal 4 [San20] in Referenzprogramme notwendig.
Des weiteren sollte OFIs gesamtes Potential diesbezüglich weiter erforscht werden. Dies schließt sowohl die Anpassung des Kommunikationsdesigns des Referenzprogramms an den jeweiligen Provider als auch einen möglichen Umstieg auf nicht nachrichtenbasierte Provider mit ein. Beispielsweise wäre ein Provider, welcher Endpoints des Typs `FI_EP_DGRAM` anbietet, denkbar.

Außerdem muss das Problem mit der Verbindungsanfrage des verbs Providers identifiziert und gelöst werden. Sobald dieses Problem gelöst ist, sollten ähnliche Messungen wie die vorliegenden durchgeführt werden.
Des weiteren könnten leichte Veränderungen am Kommunikationsdesign JULEAs vorgenommen werden, um besser auf libfabrics Bedürfnisse einzugehen. Ein erster Ansatz hierzu wäre die Umstellung auf partielle Asynchronität beim Versenden der Datenblöcke (siehe Abschnitt 5.2.2). Ein Vergleich zwischen dieser Version und der modifizierten Variante würde sich anschließen.

6. Fazit

Ziel dieser Arbeit war, anhand der Integration von libfabric in JULEA einen Beitrag zu der Frage zu leisten, ob eine Integration von spezialisierten Netzwerklösungen in existierende HPC-Programme eine valide Möglichkeit zur Erhöhung der Performanz darstellt, insbesondere wenn der Programmierer keine vorangehende Erfahrung mit der jeweiligen Speziallösung besitzt.
Nach Durchführung der Integration und der Messung der Benchmarks ist die Antwort bezüglich des sockets Providers negativ, während sie unter der Nutzung des verbs Providers durch das aufgetretene, in Unterabschnitt 4.2.2 erläuterte Problem offen bleiben muss. Der zweite Frageteil muss klar negativ beantwortet werden.
Im Bezug auf die Performanz ist daher nur eine eingeschränkte Aussage möglich. Die Ergebnisse zeigen, dass libfabric unter Verwendung des sockets Providers bei Datenübertragungen von relativ kleinen Datenmengen pro libfabric Übertragungsvorgang deutlich langsamer als die Kommunikation über einen nativ implementierten TCP socket ist. Verhältnisse von der libfabric Variante zur GIO Variante bezüglich der übertragenen Datenmengen pro Sekunde lagen hierbei um 1:9. Sobald die zu übertragende Datenmenge allerdings größer wird, fangen die Varianten an sich anzugleichen. Dies ist unter der Voraussetzung der Fall, dass der native TCP socket angebotene Optimierungsmöglichkeiten verwendet. In einem Vergleich einer unoptimierten GIO Variante zu libfabric sockets ist die unoptimierte GIO Variante schwächer bezüglich des Datendurchsatzes bei kleinen zu übertragenen Datenmengen. Letzteres überrascht, da der sockets Provider primär für die Entwicklung gedacht ist und in Erwartung stellt, dass er in geringerer Performanz resultiert [Ope20b, Eintrag `fi_sockets(7)`]. Genaueres hierzu ist in Abschnitt 4.2.1 beschrieben.
Der Umstieg auf einen anderen Provider brachte trotz der benötigten Anpassungen (siehe Abschnitt 3.5) bisher ungelöste Probleme mit sich (siehe Unterabschnitt 4.2.2).

Um die Frage in ihrer Gesamtheit und allgemein zu beantworten bedarf es der Durchführung ähnlicher Experimente, sowohl mit weiteren Speziallösungen und anderen Referenzprogrammen, als auch eine Erweiterung der JULEA-Integration auf andere Provider, wie in Abschnitt 5.4 erläutert wurde.
Außerdem sind Anpassungen der Übertragung JULEAs an das libfabric-Vorgehen möglich und nötig, um so potentiell die Performanz bei zu übertragenden batches zu erhöhen.

Quellenverzeichnis

[Aji+12] Y. Ajima, T. Inoue, S. Hiramoto und T. Shimizu. "Tofu: Interconnect for the K computer". In: *Fujitsu Scientific and Technical Journal* 48 (Juli 2012).

[Aji+18] Y. Ajima, T. Kawashima, T. Okamoto u. a. "The Tofu Interconnect D". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, S. 646–654.

[Arb] Arbeitsbereich Wissenschaftliches Rechnen. *Cluster Arbeitsbereich Wissenschaftliches Rechnen*. Online. URL: `https://wr.informatik.uni-hamburg.de/teaching/ressourcen/start#cluster`. [Online; zugegriffen 06-Oktober-2020].

[Atc+11] S. Atchley, D. Dillow, G. Shipman u. a. "The Common Communication Interface (CCI)". In: *2011 IEEE 19th Annual Symposium on High Performance Interconnects*. 2011, S. 51–60.

[Bak+16] M. Baker, F. Aderholdt, M. G. Venkata und P. Shamis. "OpenSHMEM-UCX: Evaluation of UCX for Implementing OpenSHMEM Programming Model". In: *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*. Hrsg. von M. Gorentla Venkata, N. Imam, S. Pophale und T. M. Mintz. Cham: Springer International Publishing, 2016, S. 114–130. ISBN: 978-3-319-50995-2.

[Bau05] C. Baus. *TCP_CORK: More than you ever wanted to know*. Online. 2005. URL: `https://baus.net/on-tcp_cork/`. [Online; zugegriffen 22-11-2020].

[Bha+19] S. Bhattacharya, S. Salman, M. Gorentla Venkata u. a. "An Initial Implementation of Libfabric Conduit for OpenSHMEM-X". In: *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*. Hrsg. von S. Pophale, N. Imam, F. Aderholdt und M. Gorentla Venkata. Cham: Springer International Publishing, 2019, S. 56–69. ISBN: 978-3-030-04918-8.

[DG11] J. Dean und S. Ghemawat. *LevelDB: A Fast Persistent Key-Value Store*. Online. 2011. URL: `https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html`. [Online; zugegriffen 25-09-2020].

[Faa+12] G. Faanes, A. Bataineh, D. Roweth u. a. "Cray Cascade: A scalable HPC system based on a Dragonfly network". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, S. 1–9.

[Gro+18] M. Grossman, J. Doyle, J. Dinan u. a. “Implementation and Evaluation of OpenSHMEM Contexts Using OFI Libfabric”. In: *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*. Hrsg. von M. Gorentla Venkata, N. Imam und S. Pophale. Cham: Springer International Publishing, 2018, S. 19–34. ISBN: 978-3-319-73814-7.

[Gru+15] P. Grun, S. Hefty, S. Sur u. a. “A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, Aug. 2015. URL: https://openfabrics.org/downloads/ofiwg/Industry_presentations/2015_HotI23/paper.pdf.

[Hef17] S. Hefty. *OFI Developer Guide*. Online. 2016 – 2017. URL: https://github.com/ofiwg/ofi-guide/blob/master/OFIGuide.md. [Online; zugegriffen 3-July-2020].

[Jai+16] N. Jain, A. Bhatele, S. White, T. Gamblin und L. V. Kale. “Evaluating HPC Networks via Simulation of Parallel Workloads”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, S. 154–165.

[Kuh+20] M. Kuhn, B. Warnke, M. Strassberger u. a. *Julea Repository*. Online. 2010 – 2020. URL: https://github.com/julea-io/julea. [Online; zugegriffen 10-9-2020].

[Kuh17] M. Kuhn. “JULEA: A Flexible Storage Framework for HPC”. In: *Lecture Notes in Computer Science*. Hrsg. von J. Kunkel, R. Yokota, M. Taufer und J. Shalf. Bd. 10524. ISC High Performance. Springer International Publishing, Okt. 2017, S. 712–723. URL: https://link.springer.com/chapter/10.1007/978-3-319-67630-2_51.

[Lin20] Linux man-pages project. *Linux manual TCP*. Online. 2020. URL: https://man7.org/linux/man-pages/man7/tcp.7.html. [Online; zugegriffen 13-11-2020].

[Liu+15] N. Liu, A. Haider, X.-H. Sun und D. Jin. “FatTreeSim: Modeling Large-Scale Fat-Tree Networks for HPC Systems and Data Centers Using Parallel and Discrete Event Simulation”. In: *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM PADS ’15. London, United Kingdom: Association for Computing Machinery, 2015, S. 199–210. ISBN: 9781450335836. URL: https://doi.org/10.1145/2769458.2769474.

[Mon20] MongoDB Inc. *MongoDB Manual*. Online. 2008 – 2020. URL: https://docs.mongodb.com/manual/introduction/. [Online; zugegriffen 25-09-2020].

[Ope20a] OpenFabrics Interfaces Working Group. *Libfabric Code*. Online. 2020. URL: https://github.com/ofiwg/libfabric/. [Online; zugegriffen 18-11-2020].

[Ope20b] OpenFabrics Interfaces Working Group. *Libfabric Manual*. Online. 2020. URL: https://ofiwg.github.io/libfabric/v1.11.1/man/. [Online; zugegriffen 19-10-2020].

[PC20] S. Poole und T. Curtis. *OpenSHMEM Project about*. Online. 2010 – 2020. URL: http://openshmem.org/site/About. [Online; zugegriffen 27-11-2020].

[Pro20] Prometeus GmbH. *Top 500*. Online. 1993 – 2020. URL: https://www.top500.org/. [Online; zugegriffen 02-10-2020].

[RP07] G. Rakic und F. Peters. *Gnome Project Docomentation*. Online. 2006 – 2007. URL: library.gnome.org. [Online; zugegriffen 3-July-2020].

[San20] Sandia National Laboratories. *Portal 4 about*. Online. 2020. URL: https://github.com/Portals4/portals4. [Online; zugegriffen 18-11-2020].

[Sha+15] P. Shamis, M. G. Venkata, M. G. Lopez u. a. "UCX: an open source framework for HPC network APIs and beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE. 2015, S. 40–43.

[TW12] A. S. Tanenbaum und D. J. Wetherall. *Computernetzwerke*. Pearson Studium, 1. Aug. 2012. 1032 S. ISBN: 3868941371. URL: https://www.ebook.de/de/product/19285180/andrew_s_tanenbaum_david_j_wetherall_computernetzwerke.html.

[Wal95] S. R. Walli. "The POSIX Family of Standards". In: *StandardView* 3.1 (März 1995), S. 11–17. ISSN: 1067-9936. URL: https://doi.org/10.1145/210308.210315.

Nachwort

Danksagung

An dieser Stelle möchte ich einigen Personen und Personengruppen Dank aussprechen, welche es mir ermöglicht haben, diese Arbeit zu schreiben.

Zuerst gilt mein besonderer Dank meinen beiden Betreuern Dr. Michael Kuhn und Kira Duwe, welche stets mit Rat und Verbesserungsvorschlägen die Arbeit begleitet haben.

Außerdem möchte ich Prof. Dr. Thomas Ludwig und Dr. Michael Kuhn für die Möglichkeit danken nach einer längeren, krankheitsbedingten Studienpause am Arbeitsbereich wissenschaftliches Rechnen ein Projekt durchführen zu können, um mich wieder einzuarbeiten. Aus diesem Projekt ging die vorliegende Arbeit hervor.

Der libfabric-user Mailingliste, insbesondere Sean Hefty, möchte ich dafür danken, mir mit meinen streckenweise naiven Fragen und Verständnisproblemen geholfen zu haben.

Des Weiteren danke ich Dr. Martin Kneip, welcher maßgeblich an meiner Gesundung beteiligt war und diese begleitet.

Auch möchte ich mich bei meinen Freunden und der Rollenspielrunde Farpoint dafür bedanken, dass sie meinem unruhigen Geist den einen oder anderen Abend Ruhe ermöglicht haben.

Zuletzt möchte ich meinen Eltern Susanne und Eberhard Struck meinen Dank für ihre große Geduld und Fürsorge während Studium und Krankheit, sowie dafür, mich auf meinem Weg bis an diesen Punkt begleitet zu haben, aussprechen.

Appendices

A. libfabric Code

```
1  struct fi_info {
2     struct fi_info        *next;
3     uint64_t              caps;
4     uint64_t              mode;
5     uint32_t              addr_format;
6     size_t                src_addrlen;
7     size_t                dest_addrlen;
8     void                  *src_addr;
9     void                  *dest_addr;
10    fid_t                 handle;
11    struct fi_tx_attr     *tx_attr;
12    struct fi_rx_attr     *rx_attr;
13    struct fi_ep_attr     *ep_attr;
14    struct fi_domain_attr *domain_attr;
15    struct fi_fabric_attr *fabric_attr;
16    struct fid_nic        *nic;
17 };
```

Listing A.1: `fi_info`-Struktur

```
1  struct fi_fabric_attr {
2     struct fid_fabric *fabric;
3     char              *name;
4     char              *prov_name;
5     uint32_t          prov_version;
6     uint32_t          api_version;
7  };
```

Listing A.2: `fi_fabric_attr`-Struktur

```
1  struct fi_domain_attr {
2     struct fid_domain     *domain;
3     char                  *name;
4     enum fi_threading     threading;
5     enum fi_progress      control_progress;
6     enum fi_progress      data_progress;
```

```
    enum fi_resource_mgmt resource_mgmt;
    enum fi_av_type       av_type;
    int                   mr_mode;
    size_t                mr_key_size;
    size_t                cq_data_size;
    size_t                cq_cnt;
    size_t                ep_cnt;
    size_t                tx_ctx_cnt;
    size_t                rx_ctx_cnt;
    size_t                max_ep_tx_ctx;
    size_t                max_ep_rx_ctx;
    size_t                max_ep_stx_ctx;
    size_t                max_ep_srx_ctx;
    size_t                cntr_cnt;
    size_t                mr_iov_limit;
    uint64_t              caps;
    uint64_t              mode;
    uint8_t               *auth_key;
    size_t                auth_key_size;
    size_t                max_err_data;
    size_t                mr_cnt;
};
```

Listing A.3: `fi_domain_attr`-Struktur

```
struct fi_ep_attr {
    enum fi_ep_type type;
    uint32_t        protocol;
    uint32_t        protocol_version;
    size_t          max_msg_size;
    size_t          msg_prefix_size;
    size_t          max_order_raw_size;
    size_t          max_order_war_size;
    size_t          max_order_waw_size;
    uint64_t        mem_tag_format;
    size_t          tx_ctx_cnt;
    size_t          rx_ctx_cnt;
    size_t          auth_key_size;
    uint8_t         *auth_key;
};
```

Listing A.4: `fi_ep_attr`-Struktur

```
struct fi_cq_attr {
    size_t               size;
    uint64_t             flags;
    enum fi_cq_format    format;
    enum fi_wait_obj     wait_obj;
    int                  signaling_vector;
    enum fi_cq_wait_cond wait_cond;
    struct fid_wait     *wait_set;
};
```

Listing A.5: `fi_cq_attr`-Struktur

```
struct fi_eq_attr {
    size_t               size;
    uint64_t             flags;
    enum fi_wait_obj     wait_obj;
    int                  signaling_vector;
    struct fid_wait     *wait_set;
};
```

Listing A.6: `fi_eq_attr`-Struktur

```
struct fi_info* hints;
struct fi_info* info;
struct fid_fabric* fabric;
struct fi_eq_attr eq_attr;     // Event Queue Attribute
struct fid_cq_attr cq_attr;    // Completion Queue Attribute

struct fid_domain* domain;
struct fid_eq* domain_eq;      // Event Queue

struct fid_ep* active_ep;      // Endpoint
struct fid_eq* event_queue;
struct fid_cq* cq_transmit;    // Completion Queue
struct fid_cq* cq_receive;

struct fid_pep* passive_ep;

char* service;                 // Normalerweise als Port
    ↪genutzt
struct sockaddr_in* node;      // Zieladresse
int error;

uint32_t event;
```

```
22 uint32_t event_entry_size;
23 struct fi_eq_cm_entry* event_entry;
24 struct fi_eq_err_entry event_queue_err_entry;
25
26
27 event_entry_size = sizeof(struct fi_eq_cm_entry*) + 128;
28
29 eq_attr.size = eq_size;
30 eq_attr.flags = 0;
31 eq_attr.wait_obj = FI_WAIT_MUTEX_COND;
32 eq_attr.signaling_vector = 0;
33 eq_attr.wait_set = NULL;
34
35 cq_attr.size = cq_size;
36 cq_attr.flags = 0;
37 cq_attr.format = FI_CQ_FORMAT_CONTEXT;
38 cq_attr.wait_obj = FI_WAIT_MUTEX_COND;
39 cq_attr.signaling_vector = 0;
40 cq_attr.wait_cond = FI_CQ_COND_NONE;
41 cq_attr.wait_set = NULL;
```

Listing A.7: Definition in Beispielen genutzter Variablen, Initialisierung soweit im Libfabrickontext nötig

B. Ergebnis-Tabellen

Benchmarkname (Variante)	KB/s	StAb (KB/s)	Op/s	StAb (Op/s)
write (gio)	21.204,95	2.223,60	5.176,99	542,87
write (msg sockets)	2.350,16	28,89	573,77	7,05
write (rma sockets)	2.248,63	71,64	548,98	17,49
read (gio)	21.155,08	2.219,88	5.164,81	541,96
read (msg sockets)	2.336,42	51,59	570,41	12,59
read (rma sockets)	2.226,42	47,92	543,56	11,70
write-batch (gio)	110.107,99	128,28	26.881,83	31,32
write-batch (msg sockets)	16.232,03	408,19	3.962,90	99,66
write-batch (rma sockets)	12.646,06	338,70	3.087,42	82,69
read-batch (gio)	109.529,89	115,45	26.740,70	28,19
read-batch (msg sockets)	15.368,44	856,73	3.752,06	209,16
read-batch (rma sockets)	15.081,31	654,00	3.681,96	159,67

Tabelle B.1.: Durchschnittliche über das Netzwerk übertragene KB/s und Operationen/s (Op/s) mit jeweiliger Standardabweichung (StAb) der Messungen des item-Clienten.

Benchmarkname (Variante)	KB/s	StAb (KB/s)	Op/s	StAb (Op/s)
write (gio)	21.207,47	2.228,73	5.177,61	544,12
write (msg sockets)	2.359,93	34,50	576,15	8,42
write (rma sockets)	2.254,31	79,23	550,37	19,34
read (gio)	21.174,39	2.240,92	5.169,53	547,10
read (msg sockets)	2.348,62	39,96	573,39	9,76
read (rma sockets)	2.210,30	44,50	539,62	10,86
write-batch (gio)	110.219,64	160,82	26.909,10	39,26
write-batch (msg sockets)	16.195,15	348,79	3.953,89	85,15
write-batch (rma sockets)	12.745,88	387,47	3.111,79	94,60
read-batch (gio)	109.648,94	104,47	26.769,76	25,50
read-batch (msg sockets)	15.311,90	752,01	3.738,26	183,60
read-batch (rma sockets)	15.335,27	798,17	3.743,96	194,87

Tabelle B.2.: Durchschnittliche über das Netzwerk übertragene KB/s und Operationen/s (Op/s) mit jeweiliger Standardabweichung (StAb) der Messungen des object-Clienten für verteilte Objekte.

Benchmarkname (Variante)	KB/s	StAb (KB/s)	Op/s	StAb (Op/s)
write (gio)	21.315,72	2.228,53	5.204,03	544,08
write (msg sockets)	2.365,69	31,24	577,56	7,63
write (rma sockets)	2.251,06	70,46	549,58	17,20
read (gio)	21.265,19	2.241,42	5.191,70	547,22
read (msg sockets)	2.348,35	51,08	573,33	12,47
read (rma sockets)	2.234,13	57,57	545,44	14,06
write-batch (gio)	111.640,13	133,81	27.255,89	32,67
write-batch (msg sockets)	16.193,09	375,38	3.953,39	91,65
write-batch (rma sockets)	12.561,55	376,01	3.066,78	91,80
read-batch (gio)	111.129,97	113,83	27.131,34	27,79
read-batch (msg sockets)	15.328,36	749,97	3.742,28	183,10
read-batch (rma sockets)	15.341,50	802,63	3.745,48	195,96

Tabelle B.3.: Durchschnittliche über das Netzwerk übertragene KB/s und Operationen/s (Op/s) mit jeweiliger Standardabweichung (StAb) der Messungen des object-Clienten.

Benchmarkname (Variante)	KB/s	StAb (KB/s)	Op/s	StAb (Op/s)
4M write (gio)	83.887,50	1.912,66	20,00	0,46
4M write (msg sockets)	77.095,10	1.012,78	18,38	0,24
4M write (rma sockets)	75.590,32	1.091,86	18,02	0,26
4M read (gio)	83.849,35	2.650,01	19,99	0,63
4M read (msg sockets)	80.281,02	1.964,41	19,14	0,47
4M read (rma sockets)	79.870,49	1.918,00	19,04	0,46
4K write (gio)	6.329,34	680,94	1.545,25	166,25
4K write (msg sockets)	860,90	13,69	210,18	3,34
4K write (rma sockets)	818,40	19,29	199,80	4,71
4K read (gio)	7.312,82	654,40	1.785,36	159,77
4K read (msg sockets)	879,92	17,02	214,82	4,15
4K read (rma sockets)	839,57	4,58	204,97	1,12
attr read (gio)	5.955,98	697,98	1.454,10	170,40
attr read (msg sockets)	830,31	5,42	202,71	1,32
attr read (rma sockets)	809,97	3,69	197,75	0,90
attr write (gio)	6.047,98	1.437,58	1.476,56	350,97
attr write (msg sockets)	1.304,55	14,81	318,49	3,62
attr write (rma sockets)	1.262,10	12,89	308,13	3,15

Tabelle B.4.: Durchschnittliche über das Netzwerk übertragene KB/s und Operationen/s (Op/s) und die jeweilige Standardabweichung (StAb) der HDF5 Benchmarks für read und write Operationen auf einer Datenmenge von 4 KB und 4 MB sowie die attribute Benchmarks

Benchmarkname	KB/s	StAb (KB/s)	Op/s	StAb (Op/s)
object distr obj read	93,09	0,003	22,73	0,0007
object distr obj read-batch	110.217,98	73,40	26.908,69	17,92
object distr obj write	93,09	0,003	22,73	0,0006
object distr obj write-batch	110.749,78	125,67	27.038,52	30,68
object object read	93,09	0,003	22,73	0,0007
object object read-batch	111.554,37	50,42	27.234,95	12,31
object object write	93,09	0,002	22,73	0,0005
object object write-batch	111.937,58	414,37	27.328,51	101,17
item item read	93,09	0,001	22,73	0,0003
item item read-batch	110.183,13	56,31	26.900,18	13,75
item item write	93,12	0,001	22,74	0,0003
item item write-batch	110.672,54	94,47	27.019,66	23,06
hdf5 attribute write	32,00	0,002	7,81	0,0005
hdf5 attribute read	23,81	0,001	5,81	0,0003
hdf5 dataset4M write	30.744,71	369,21	7,33	0,09
hdf5 dataset4M read	19.480,28	172,54	4,64	0,04
hdf5 dataset4K write	46,48	0,16	11,35	0,04
hdf5 dataset4K read	23,76	0,0007	5,80	0,0002

Tabelle B.5.: Durchschnittliche über das Netzwerk übertragene KB/s und Operationen/s (Op/s) mit jeweiliger Standardabweichung (StAb) der Messungen der unoptimierten GIO Variante.

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang BSc. 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.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift