UNIKASSEL
VERSITÄT

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Master Informatik

Fachgebiet Programmiersprachen / Methodik
Prof. Dr. Claudia Fohry

Entwicklung einer Netzwerkschicht für ein Java-basiertes Programmiersystem aus dem Bereich des Hochleistungsrechnens

Masterarbeit
vorgelegt von

Steve Hildebrandt

13. Dezember 2020

Erstgutachter:	Prof. Dr. Claudia Fohry
Zweitgutachter:	Prof. Dr. Albert Zündorf
Betreuer:	Prof. Dr. Claudia Fohry

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

APGAS	Asynchronous Partitioned Global Address Space.
BC	Betweenness Centrality.
BTL	Byte Transfer Layer.
GASNet	Global-Address Space Networking.
GCC	GNU Compiler Collection.
GLB	Global Load Balancing.
HPC	High Performance Computing.
IMDG	In-Memory Data Grid.
IPoIB	IP over Infiniband.
JDK	Java Development Kit.
JNI	Java Native Interface.
JOPI	Java Object-Passing Interface.
JVM	Java Virtual Machine.
MCA	Modular Component Architecture.
MPI	Message Passing Interface.
MTU	maximale Übertragungseinheit.
PGAS	Partitioned Global Address Space.
PML	Point-To-Point Management Layer.
RMA	Remote Memory Access.
RMI	Remote Method Invocation.
ULFM	User Level Fault Mitigation.
UTS	Unbalanced Tree Search.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Masterarbeit mit dem Titel "Entwicklung einer Netzwerkschicht für ein Java-basiertes Programmiersystem aus dem Bereich des Hochleistungsrechnens" selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, habe ich als solche gekennzeichnet. Diese Masterarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Kassel, den 13. Dezember 2020

1. Einleitung

Im Hochleistungsrechnen werden üblicherweise vernetzte Multicore-Rechner (Cluster) zur Lösung von Problemen eingesetzt. Um die Ressourcen eines solchen Systems nutzen zu können, wird ein Problem in Teilprobleme zerlegt. Diese werden parallel von verschiedenen Recheneinheiten (CPU-Kernen, Prozessoren oder auch Rechnern) bearbeitet.

Um kooperativ eine Lösung aus den Teilproblemen zu ermitteln, ist Kommunikation zwischen den Recheneinheiten notwendig. Um die Rechner effizient zu nutzen gilt es diese zusätzlichen Kommunikationskosten zu minimieren.

Über die Jahre wurden viele Programmiersysteme für den High Performance Computing (HPC) Bereich entwickelt. Sie unterscheiden sich im Umfang der Steuerungsmöglichkeiten des Programmierers über den Ablauf der Berechnung.

Eines der meist genutzten Programmiersysteme ist das Message Passing Interface (MPI). MPI bietet eine große Anzahl von spezialisierten Konstrukten zur Kommunikation zwischen Prozessen. Hintergrund für die große Anzahl an Konstrukten ist die effiziente Unterstützung unterschiedlicher Kommunikationsmuster und Hardwaretechnologien bei gleichzeitiger Abwärtskompatibilität zwischen API-Versionen. Die meisten der Konstrukte zur Kommunikation bringen eine nicht unerhebliche Menge an Anforderungen mit sich. So ist zum Beispiel für die grundlegenden Konstrukte zur Peer-To-Peer Kommunikation genau geregelt wann Speicher nach Aufruf eines Konstrukts wieder genutzt werden kann, ob Kopien von Daten erzeugt werden und in welcher Reihenfolge Konstrukte aufgerufen werden müssen. Hinzu kommt, dass es von den meisten Konstrukten Varianten mit abgeändertem Verhalten gibt. Die Diversität bei Verhalten und Anforderungen der Konstrukte sorgt in Kombination mit der großen Auswahl an Konstrukten für eine hohe Komplexität bei Entwicklung, Wartung und Optimierung von MPI-Programmen. Des Weiteren führt das Verletzen von Anforderungen nicht automatisch zu einem Fehler oder falschen Ergebnis. Dieser Umstand erschwert das Entwickeln von korrekten MPI-Programmen.

Neben dem klassischen MPI wird in dieser Arbeit das moderne Programmiersystem APGAS betrachtet. APGAS ist eine Java-Implementierung des gleichnamigen Programmiermodells [1], wobei APGAS für Asynchronous Partitioned Global Address

Space steht. Das APGAS Modell ist eine Variante des Partitioned Global Address Space (PGAS) Modells. Das PGAS Modell betrachtet den gesamten Speicher eines parallelen Systems als global adressierbar. Speicherpartitionen zusammen mit Prozessoren werden als Place bezeichnet. Zugriffe auf jede Speicherpartition sind von jedem Place aus möglich, jedoch sind Zugriffe auf lokale Speicher schneller. Programme, die dem ursprünglichen PGAS Modell entsprechen, folgen dem Prinzip SPMD, sprich auf allen beteiligten Places wird dasselbe Programm ausgeführt. Im APGAS Modell hingegen wird ein Programm zunächst auf einem Place ausgeführt. Weitere Places können durch das dynamische Zuweisen von Teilberechnungen ausgenutzt werden. Diese Teilberechnungen werden im Modell als Aktivitäten bezeichnet und können sowohl in synchroner als auch asynchroner Form verarbeitet werden. Ziel des APGAS Modells ist eine Reduktion der Komplexität, die Systeme wie MPI für den Anwendungsprogrammierer mit sich bringen.

APGAS verfolgt den Ansatz möglichst viel Komplexität von Programmiererseite in das Laufzeitsystem zu verlagern. Aufgaben wie das Starten von Prozessen, das Verwalten und Ausführen von Tasks, Serialisierung und Interprozesskommunikation werden durch das Laufzeitsystem geleistet. Obwohl Funktionalität teilweise in Komponenten separiert ist, ist der Kern des Laufzeitsystems monolithisch strukturiert. So ist der Netzwerkcode zu Teilen in Komponenten realisiert, während im Kern des Laufzeitsystems ebenfalls zugehörige Funktionalität implementiert wurde. Implementiert wurde der Netzwerkcode in APGAS auf Basis einer weiteren Java Bibliothek, Hazelcast [2].

Betrachtet man den aktuellen Netzwerkcode sowie die Features von Hazelcast, so ist Potential für bessere Performance gegeben. Mögliche Ansätze, um eine performantere Kommunikation in APGAS zu realisieren, sind unter anderem die direkte Unterstützung von Spezialhardware wie InfiniBand-Karten, die Implementierung von neuen Konstrukten optimiert für spezifische Kommunikationsmuster oder auch Optionen, um die Netzwerkschicht durch Tuning an spezifische Probleme anzupassen. Neben diesen möglichen Verbesserungen ist die aktuelle Struktur des APGAS Laufzeitsystems für die Analyse der Kommunikationsperformance eher hinderlich. Fehlende Isolation der Hazelcast Bibliothek sowie des darauf aufbauenden Netzwerkcodes stehen im Weg alternativer Implementierungen der Kommunikation in APGAS. Eine alternative Implementierung, die dynamisch mit bestehendem Netzwerkcode austauschbar ist erlaubt, im Gegensatz zu den bereits möglichen Vergleichen zwischen gesamten Programmiersystemen, APGAS intern Vergleiche des Netzwerkcodes. Die Performance von bestehendem Netzwerkcode ließe sich im Rahmen eines solchen Vergleiches isoliert vom Rest des Laufzeitsystem betrachten und somit evaluieren.

Diese Arbeit setzt sich zum Ziel mithilfe einer alternativen Implementierung des Netzwerkcodes in APGAS die Performance des Systems zu verbessern sowie die be-

stehende Implementierung zu bewerten. Neben Leistungsmetriken wie Speicherbedarf, CPU-Auslastung oder auch CPU-Zeit ist auch die Benutzerfreundlichkeit für Anwendungsprogrammierer relevant bei der Entwicklung. Unter Benutzerfreundlichkeit fällt zum Beispiel der zu betreibende Aufwand zur Installation oder notwendige Konfiguration um eine APGAS Anwendung auszuführen.

Zur Erfüllung dieser Ziele wurde zunächst das APGAS Laufzeitsystem restrukturiert. Diese Anpassung war wie bereits angedeutet notwendig, um den bestehenden Netzwerkcode klar innerhalb des eher monolithischen Laufzeitsystems zu isolieren. Als Ergebnis der Restrukturierung findet sich die ursprüngliche Funktionalität in mehreren Komponenten mit klar definierten Schnittstellen wieder. Diese Komponenten, beziehungsweise deren Funktionalität, sind im Folgenden als APGAS Netzwerkschicht zu verstehen. Die so geschaffenen Schnittstellen sind Grundlage für die Implementierung alternativer Netzwerkschichten in APGAS.

Als nächster Schritt wurde eine neue Netzwerkschicht basierend auf MPI entwickelt. Hierbei ist festzuhalten, dass die neue Netzwerkschicht nicht den Anspruch erhebt die Möglichkeiten von MPI im Bezug auf Performanceverbesserungen auszuschöpfen. Verwendet wurde die MPI Implementierung OpenMPI. Zur Einbindung in Java beziehungsweise APGAS wurden die von Vega-Gisbert et al. [3] entwickelten JNI-Bindings für OpenMPI genutzt. Die Entscheidung für MPI zum Aufbau einer neuen Netzwerkschicht liegt vor allen darin begründet, dass die zuvor angesprochenen Ansätze zur Verbesserung der Performance mittels MPI umsetzbar sind. In gängigen MPI-Implementierungen wird Spezialhardware wie InfiniBand-Karten direkt unterstützt, viele Kommunikationsmuster haben ein passendes Konstrukt in der MPI-API und Tuning-Möglichkeiten sind üblicherweise als Teil von MPIs Konfiguration verfügbar. Eine Schwierigkeit bei der Entwicklung der Netzwerkschicht war die Sprachbarriere zwischen der C-Bibliothek OpenMPI und dem in Java implementierten APGAS. Verschiedene Ansätze zur Verwendung von MPI bezüglich vergleichbarer Funktionalität in Java werden im weiteren Verlauf dieser Arbeit diskutiert.

Abschließend wurden die beiden Netzwerkschichten im Rahmen mehrerer Experimente analysiert. Für die Performanceanalyse der Netzwerkschichten wurden die von Posner [4] implementierten Benchmarks Betweenness Centrality (BC) und Unbalanced Tree Search (UTS) verwendet. Hierbei wurden die Experimente jeweils einmal mit Infiniband und mit Ethernet durchgeführt. Beide Benchmarks liefern separate Laufzeiten für die verteilte Berechnung von Zwischenergebnissen und dem Zusammenführen selbiger. Betrachtete man die gesamte Laufzeit der Benchmarks, so dominiert klar der Berechnungsteil. Auffällig ist, während die MPI Netzwerkschicht tendenziell bessere Laufzeiten für den Berechnungsteil der Experimente erreicht, variieren die Zeiten bei isolierter Betrachtung der Ergebniszusammenführung stark. Diese Laufzeiten lassen vermuten, dass die Performance der Netz-

werkschichten stark im Zusammenhang mit der Nachrichtengröße steht. Dieser Vermutung folgend wurde experimentell Bandbreite und Latenz in Abhängigkeit von Nachrichtengröße bestimmt.

Ergebnis der Experimente war, dass sich mit der MPI Netzwerkschicht im Vergleich zur Hazelcast Netzwerkschicht teilweise kürzere, andernfalls vergleichbare Laufzeiten erreichen lassen. Weiterhin unterstützt die MPI Netzwerkschicht eine größere Anzahl Places. Hazelcast benötigt eine größere Anzahl limitierter Systemressourcen, die für Kommunikationsverbindungen zwischen Places eingesetzt werden, da Verbindungen immer von jedem Place zu jedem anderen Place aufgebaut werden. Im Fall der MPI Netzwerkschicht werden hingegen Verbindungen zwischen Places nur nach Bedarf etabliert und so weniger der limitierten Ressourcen verbraucht. Im Zuge der Experimente hat sich ebenfalls gezeigt, dass die MPI Netzwerkschicht zuverlässig größere Kommunikationsvolumen verarbeiten kann. Im Rahmen des Experiments zur Ermittlung der Bandbreite lief Hazelcast systematisch in einen internen Fehler, der weder weiteren Programmfortschritt noch eine saubere Terminierung zuließ. Jedoch erfordert das Einrichten von APGAS mit OpenMPI einen nicht trivialen Bau- und Konfigurationsprozess. Da OpenMPI mit Java-Bindings nicht als gebautes Paket verfügbar ist, ist dieser Prozess auf jeder eingesetzten Plattform notwendig. Die Komplexität dieses Prozesses liegt vor allem in der großen Anzahl an Features, die in OpenMPI durch optionale Abhängigkeiten realisiert werden und der Fülle von gebotenen Konfigurationsmöglichkeiten begründet. Tuning-Möglichkeiten konnten durch die neue Netzwerkschicht und die Verwendung von OpenMPI geschaffen werden. Jedoch ist das Tuning von OpenMPI komplex. So haben unter anderem verwendete Hardware und Protokolle (z.B. IP) direkten Einfluss auf verfügbare Optionen.

Problematisch für die Entwicklung der Netzwerkschicht waren Inkompatibilitäten zwischen OpenMPI und Bibliotheken, die zur Verwendung von Infiniband empfohlen werden. In Konsequenz konnte der Bauprozess mit bestimmten Versionskombinationen nicht abgeschlossen werden oder Laufzeitfehler wurden verursacht. Ebenfalls konnte verfügbare Infiniband-Hardware auf dem für Experimente verwendeten Cluster nur über IP over Infiniband (IPoIB) und nicht direkt verwendet werden. Darüber hinaus wurden zunächst innerhalb der MPI Netzwerkschicht blockierende APIs von OpenMPI verwendet. Diese stellten sich als problematisch heraus, da aktives Warten innerhalb von OpenMPI große Anteile an CPU-Zeit in Anspruch nahm. Optionen, die den Einsatz von aktivem Warten steuern sollen, waren ohne Effekt. Um diese Problematik zu umgehen, wurden nicht-blockierende APIs verwendet und das Warten auf Fertigstellung innerhalb von APGAS realisiert. Diese Änderungen sorgten für eine ungefähre Halbierung der Laufzeiten für betroffene Experimente.

Im Folgenden gibt Abschnitt 2 einen Überblick über die für diese Masterarbeit

relevanten Bibliotheken und Frameworks. Anschließend wird in Abschnitt 3 das Konzept hinter der neu entwickelten Netzwerkschicht vorgestellt. In diesem Zuge werden Anforderungen an die Netzwerkschicht und der Entscheidungsprozess MPI mittels Java Native Interface (JNI) Bindings zu verwenden dargelegt. Die Implementierung dieses Konzepts wird in Abschnitt 4 vorgestellt. Daraufhin werden in Abschnitt 5 die durchgeführten Experimente und ermittelten Ergebnisse beschrieben und diskutiert. Die Arbeit schließt in Abschnitt 6 mit einer Zusammenfassung und einem Ausblick auf zukünftige Arbeiten ab.

2. Grundlagen

2.1. Die APGAS Bibliothek

Die APGAS Java-Bibliothek ist entwickelt worden, um Funktionalität der Programmiersprache X10 [5] durch die Portierung nach Java einer breiteren Anzahl von Nutzern zugänglich zu machen [6]. Laut Tardieu [7] dient die Bibliothek der Entwicklung paralleler, verteilter, resilienter, elastischer Programme für den Einsatz in Cluster Umgebungen. Die ursprüngliche APGAS Version 1.0 wurde auf Basis von Java 8 umgesetzt, da Lambda-Ausdrücke zentral für das API-Design sind. Der in dieser Arbeit verwendete APGAS-Fork [8] beinhaltet zusätzliche Bugfixes, Erweiterungen in den Bereichen Fehlertoleranz und Lastenbalancierung [9, 10] sowie Unterstützung für lokalitätsflexible Tasks [11, 12]. Die im Fork enthaltenen Anpassungen erfordern Java 12.

Wie bereits beschrieben, ist die Bibliothek eine Implementierung des APGAS Programmiermodells. Ein Place, auf Modellebene Speicher und Recheneinheiten, ist in der Java Implementierung einem JVM Prozess gleichzusetzen. Neben dem Begriff Place ist das Konzept einer Aktivität relevant. Eine Aktivität kann als leichtgewichtiger Thread, der einen spezifischen Code-Abschnitt ausführt, verstanden werden. Aktivitäten können sowohl auf dem aktuellen, als auch einem entfernten Place erzeugt werden. Die Ausführung einer Aktivität kann sowohl synchron, als auch asynchron geschehen. Soll heißen, wenn durch ein Konstrukt eine Aktivität synchron ausgeführt wird, so wird der Programmfluss erst dann fortgeführt, wenn die Aktivität geendet hat. Analog gilt, dass der Programmfluss bei einer asynchron ausgeführten Aktivität vor deren Ende fortgeführt werden kann. Das Konzept Aktivität entspricht einem Task wie er von Java `ForkJoinPools` [13] verarbeitet wird. In der APGAS Bibliothek wird pro Place ein `ForkJoinPool` zur Verwaltung und Verarbeitung der Tasks instantiiert.

Konzeptuell beginnt die Ausführung einer APGAS Anwendung mit einer Aktivität auf dem ursprünglichen Place 0. Das Ende des Programms ist mit dem Ende dieser Aktivität erreicht. In der Praxis bedeutet dies, dass mit der Ausführung des eigentlichen Programms begonnen werden kann, wenn alle Places verbunden sind. Die notwendigen Prozesse können extern oder über einen so genannten `Launcher`

gestartet werden. Welchen Launcher das APGAS-Laufzeitsystem verwendet, wird zu Programmstart festgelegt. Zum Beispiel startet der `SshLauncher` alle weiteren benötigten Prozesse durch SSH Verbindungen von der Place 0 JVM aus. Welche Prozesse zu starten sind, wird durch eine Konfigurationsdatei gesteuert.

```
1  import apgas.Constructs;
2  import apgas.Place;
3
4  public class HelloWorld {
5    public static void main(String[] args) {
6      Constructs.finish(() -> {
7        for (Place place : Constructs.places()) {
8          Constructs.asyncAt(place, () -> {
9            System.out.println("message from " + place);
10         });
11       }
12     });
13   }
14 }
```

Listing 2.1.: Ein einfaches APGAS Programm

Wie Listing 2.1 zeigt, basiert die Struktur von APGAS Programmen auf Varianten der `async`-/`finish`-Konstrukte. Das in Listing 2.1 verwendete Konstrukt `places` liefert eine Liste aller Places zurück. Das Konstrukt `asyncAt` erzeugt einen neuen Task beziehungsweise eine Aktivität zur asynchronen Verarbeitung auf dem beim Aufruf festgelegten Place. Das `finish` Konstrukt dient dem Warten auf alle in einem Block erzeugten Tasks. Entsprechend erzeugt das Programm aus Listing 2.1 auf jedem Place einen Task und terminiert wenn diese ausgeführt wurden. Die Ausgabe ist eine Nachrichtenzeile für jeden Place mit dem das APGAS Programm gestartet wurde.

Neben dem `asyncAt` Konstrukt gibt es noch die Varianten `async` und `at`. Beide erzeugen jeweils einen neuen Task. Tasks, die mit `async` erzeugt wurden, werden asynchron auf dem aktuellen Place ausgeführt. Mit dem Konstrukt `at` wird hingegen ein Task zur synchronen Verarbeitung auf einem festgelegten Place erzeugt.

Zusätzlich zu den eben beschriebenen Tasks erlaubt das Konstrukt `asyncAny` das Erzeugen von lokalitätsflexiblen Tasks. Hierbei handelt es sich um Tasks, die an keinen Place gebunden sind. Zu bemerken ist, dass lokalitätsflexible Tasks nicht mit Aktivitäten gleichzusetzen sind. Mit `asyncAny` erzeugte Tasks werden stets asynchron verarbeitet. In der verwendeten Version von APGAS können lokalitätsflexible Tasks nicht mit “normalen” Tasks vermischt verwendet werden. Daher und aufgrund notwendiger Unterschiede auf Implementierungsseite, gibt es ein zu dem regulären `finish` analoges Konstrukt `finishAsyncAny` für lokalitätsflexible Tasks.

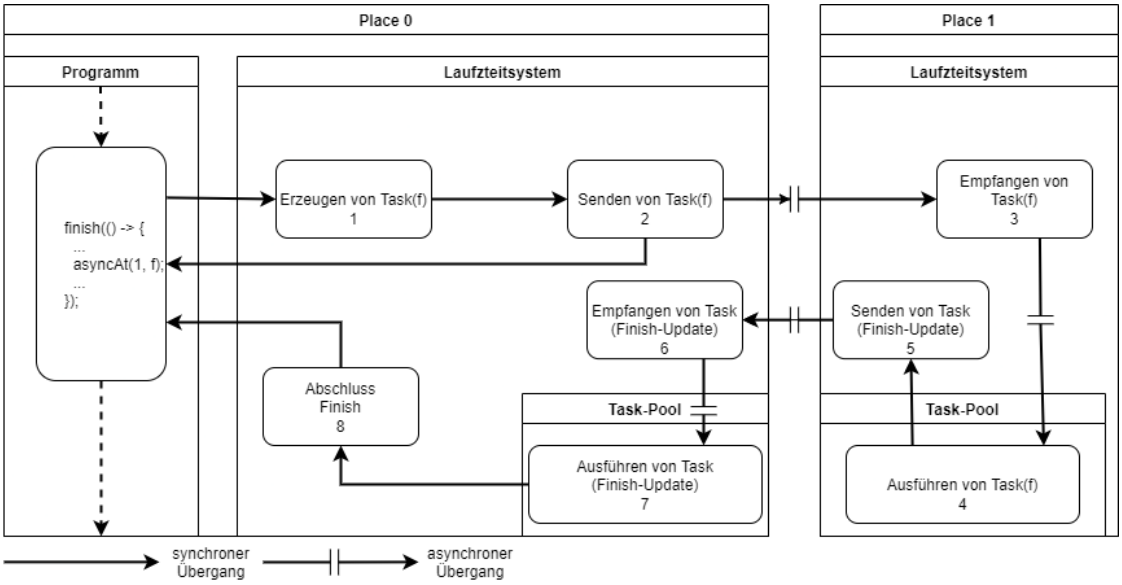

Abbildung 2.1.: Das `asyncAt` Konstrukt der ursprünglichen APGAS Bibliothek betrachtet im Hinblick auf Kommunikation

Aus Sicht der Netzwerkschicht sind APGAS-Konstrukte nicht mit Kommunikation gleichzusetzen:

- `async(task)` Ein neuer Task wird dem lokalen Pool hinzugefügt. Entsprechend ist keine Kommunikation notwendig.
- `asyncAt(place, task)` Falls es sich bei `place` um den lokalen Place handelt, so verhält sich `asyncAt` wie `async`. Andernfalls muss der Task an den Ziel-Place übertragen werden. Dies kann anhand von Abbildung 2.1 nachvollzogen werden.
  Zunächst wird eine Repräsentation des Tasks erzeugt (1). Diese enthält den auszuführenden Code-Block, das zugehörige Finish und den Ziel-Place. Diese Repräsentation wird nun an Hazelcast zur Übermittlung weitergegeben (2). Hierbei wird nicht auf die vollständige Übertragung gewartet. Bei Empfang durch den Ziel-Place wird die erhaltene Nachricht von Hazelcast wieder in die Task Repräsentation übersetzt (3). Der Task kann nun in den lokalen `ForkJoinPool` zur Verarbeitung eingereiht werden (4). Wird der Task ausgeführt, so muss auch das zugehörige Finish verarbeitet werden (5).
  Im Normalfall (`DefaultFinish`) bedeutet dies, dass ein neuer Task zurück an den Place, auf dem das `finish` aufgerufen wurde, kommuniziert wird. Auch hierbei wird der Tasks serialisiert, übertragen, deserialisiert und in den entsprechenden Task-Pool eingereiht (5, 6). Der Task dient der Verarbeitung der `finish` Logik auf dem ursprünglichen Place (7). Haben sich so alle transitiv

erzeugten Tasks eines `finish` zurückgemeldet, so kann dieses abgeschlossen und der Programmfluss fortgesetzt werden (8).

- `asyncAny(task)` Ein neuer lokalitätsflexibler Task wird dem Pool des Places hinzugefügt, auf dem das Konstrukt aufgerufen wird. Es erfolgt keine Kommunikation als Konsequenz des Aufrufs von `asyncAny`. Lokalitätsflexible Tasks werden nur im Rahmen der Lastenbalancierung durch das Laufzeitsystem übertragen.
- `finish(activity)` Jeder Task, der direkt aus der Finish-Aktivität heraus erzeugt wurde, kommuniziert seine Fertigstellung zu dem Place auf welchem `finish` aufgerufen wurde.
- `finishAsyncAny(activity)` Im Gegensatz zu `finish` wird nach dem Aufruf von `finishAsyncAny` periodisch zwischen allen Places kommuniziert wie viele Tasks noch ausstehend sind. Dies vermeidet die Kommunikation des Abschlusses von jedem einzelnen Task.
- `at(task, place)` Das `at(task, place)` Konstrukt entspricht einem Aufruf von `finish(() → asyncAt(task, place))`.

Wie bereits erwähnt, nehmen lokalitätsflexible Tasks in der momentanen Version von APGAS eine Sonderrolle ein. Die Tatsache, dass lokalitätsflexible Tasks nicht an einen Place zur Ausführung gebunden sind, erlaubt dem Laufzeitsystem größere Kontrolle. Dies ist Grundlage für Inter-Place Lastenbalancierung und ermöglicht Fehlertoleranz auf Ebene von Tasks. Da ein mit `asyncAny` erzeugter Task zunächst dem lokalen Task-Pool hinzugefügt wird, ist es möglich, dass keine Kommunikation aus dem Aufruf resultiert. Die Fertigstellung eines solchen Tasks wird nicht explizit kommuniziert. Stattdessen wird der Place, auf dem das Konstrukt `finishAsyncAny` aufgerufen wurde, von allen anderen beteiligten Places einmalig benachrichtigt. Diese Benachrichtigung findet dann statt, wenn auf einem Place alle zugeordneten lokalitätsflexiblen Tasks verarbeitet wurden. Lokalitätsflexible Tasks verbrauchen entsprechend nur dann Kommunikationsressourcen, wenn eine Übertragung auf einen anderen Place zur Lastenbalancierung durchgeführt wird.

Lastenbalancierung in APGAS-Anwendungen kann mittels zweier konkurrierender Ansätze erreicht werden. Ein Ansatz basiert auf der Verwendung des gerade beschriebenen `asyncAny`-Konstrukts. Das ist konzeptuell in dem Programmiersystem HabaneroUPC++ [14] wiederzufinden, um Lastenbalancierung nicht nur innerhalb von, sondern auch zwischen Places zu ermöglichen. Andernfalls kann ein APGAS Port der Global Load Balancing (GLB) Bibliothek [9] verwendet werden. Global Load Balancing (GLB) übernimmt die Lastenbalancierung von Task-basierten Anwendungen und wurde ursprünglich für die Programmiersprache X10 [5] entwickelt. Beide Ansätze basieren auf Work-Stealing Algorithmen. Basis dieser Algorithmen

ist, dass Places ohne Tasks Arbeit mittels einer Anfrage an ausgelastete Places übertragen bekommen. Für diese Arbeit sind beide Ansätze von Relevanz, da beide Varianten im Rahmen der durchgeführten Experimente zum Einsatz gekommen sind. Sowohl das GLB-Framework als auch Lastenbalancierung mittels `asyncAny` setzen voraus, dass Tasks Ergebnisse von einem einheitlichen Typ erzeugen. Diese Ergebnisse werden wiederum mittels eines vom Anwendungsprogrammierer festgelegten Reduktionsoperators zusammengefügt. Um Tasks in beliebiger Reihenfolge ausführen zu können, wird weiterhin vorausgesetzt, dass dieser Operator kommutativ und assoziativ ist.

Im GLB-Framework sind so genannte Worker für die Lastenbalancierung und Verarbeitung der Tasks zuständig. Jeder Place führt genau einen GLB-Worker aus. Alle Worker arbeiten solange im gesamten System noch Tasks zu berechnen sind. Falls möglich werden im Wechsel lokale Tasks ausgeführt und Anfragen anderer Worker beantwortet. Sind keine Tasks lokal verfügbar, setzt ein Worker eine Anfrage ab, um Tasks von einem anderen Worker zu erhalten. Sprich, es wird ein Work-Stealing Algorithmus eingesetzt, um die gleichmäßige Auslastung von Workern zu ermöglichen. Das GLB-Famework erlaubt nur ein Worker pro Place. Als Konsequenz ist es notwendig mehrere Places auf einem Multicore-Rechner zu starten, soll dieser voll ausgelastet werden.

Lastenbalancierung basierend auf `asyncAny` erlaubt hingegen Multicore-Rechner durch das Starten von einem Place pro Rechner auszulasten. Der `asyncAny` Ansatz vermeidet im Vergleich zu dem GLB-Framework zusätzliche Kommunikation und Speicherverbrauch. Dieser Mehraufwand wird innerhalb des Frameworks durch das Starten von mehreren Places auf einem Rechner verursacht. Innerhalb eines Places wird die Lastenbalancierung durch `ForkJoinPools` [13] realisiert. Zwischen Places kommt ein Work-Stealing Algorithmus analog zu GLB zum Einsatz. Zur Bestimmung welche Anfrage an welchen Place gestellt wird, setzen sowohl `asnycAny`, als auch GLB auf ein so genanntes Lifeline-Schema. [12]

APGAS ist eine reine Java-Bibliothek im Sinne, dass nur Abhängigkeiten zu weiteren Java-Bibliotheken bestehen. Zum einen wird die Bibliothek Kryo [15] als Alternative zu Javas nativer Serialisierung verwendet. Die Verwendung von Kryo ist optional und wird zum Programmstart festgelegt. Im Rahmen dieser Arbeit wurde Kryo nicht benutzt. Zum anderen besteht eine Abhängigkeit von Hazelcast [2]. Hazelcast dient der Kommunikation zwischen JVMs und der Realisierung des Konzepts eines global adressierbaren gemeinsamen Speichers. Hazelcast wird im folgenden Abschnitt 2.2 genauer beschrieben.

2.2. Die Hazelcast Bibliothek

Hazelcast ist eine Java Platform, die unter anderem eine verteilte Speicher Implementierung und stream-basierte Datenverarbeitung ermöglicht. Die Hazelcast Speicher Implementierung kann für die Zwecke von APGAS als verteilte Variante einer `ConcurrentMap`[16] verstanden werden. Hazelcasts verteilte Speicherlösung läuft unter dem Begriff In-Memory Data Grid (IMDG) und stellt sich als Alternative zu In-Memory Datenbanken dar [17]. Für APGAS ist nur die Hazelcast Core Bibliothek relevant.Die Core Bibliothek enthält die IMDG Implementierung und grundlegende APIs zur verteilten Datenverarbeitung. Hazelcasts IMDG Implementierung ist eine erprobte Basis, die sich eignet, um das Speicherkonzept des APGAS Modells zu realisieren. Weiterhin ist es möglich IMDG Anwendungen elastisch zu skalieren. Betrachtet man Schlüssel des IMDG als Speicheradressen, Clients als Places, so zeigt sich schnell, dass mit Hazelcasts IMDG das APGAS Speicherkonzept direkt umsetzbar ist. Darüber hinaus sind zusätzlich verteilte Datenstrukturen, wie Listen und Maps, verfügbar. Neben dem IMDG sind Hazelcasts Möglichkeiten zur Kommunikation zwischen JVMs für APGAS relevant. [2]

Der Ablauf einer Hazelcast Anwendung beginnt mit dem Konfigurieren und Erzeugen einer Hazelcast Instanz. Mit der Erzeugung einer Instanz werden anhand der zu Beginn festgelegten Konfiguration Verbindungen zwischen Prozessen aufgebaut. Durch die Instanz können benannte verteilte Datenstrukturen erzeugt und manipuliert werden. Darüber hinaus bietet die Instanz Zugriff auf weitere zentrale APIs wie die `Executor` Schnittstelle. Der `Executor` kann analog zu der Standard Java Variante (`java.util.concurrent.Executor` [18]) als Einstiegspunkt zum Erzeugen und Ausführen von Tasks verstanden werden. Der Hazelcast Executor ist jedoch im Vergleich zu der Java Variante für die Nutzung in einer verteilten Anwendung designt. Die Anwendung endet für Hazelcast mit dem Aufruf der `shutdown`-API auf den Instanzen. Da Hazelcast, wie bereits beschrieben, elastisch ist, kann ein Prozess jederzeit über das Erzeugen einer passenden Instanz einer Anwendung hinzugefügt oder durch `shutdown` entfernt werden.

Kommunikation findet aus Sicht einer Hazelcast Anwendung generell nicht als Übertragung von Bytes statt. Stattdessen werden verteilte Datenstrukturen manipuliert oder Tasks zur Realisierung komplexerer Operationen erzeugt. Ein Task kann sowohl von der lokalen als auch durch eine entfernte Hazelcast Instanz verarbeitet werden. Das Erzeugen von Tasks ist hierbei über die Executor APIs `execute` bzw. `submit` möglich. Die Varianten von beiden APIs unterscheiden sich nur dadurch, ob ein Rückgabewert (`submit`) erzeugt wird oder nicht. Unabhängig davon, welche der beiden APIs verwendet wird, lässt sich bestimmen welche Hazelcast Instanz den zu erzeugenden Task ausführen soll.

2.3. Message Passing Interface

Das Message Passing Interface (MPI) ist, wie zu Beginn dieser Arbeit beschrieben, die klassische System-Wahl für HPC-Anwendungen. MPI bietet eine standardisierte API, umgesetzt durch jahrelang erprobte Implementierungen. Eine breite Unterstützung für Hardware und Plattformen ist gegeben. Die MPI API wurde in der ursprünglichen Version nicht für Anforderungen wie Multithreading, Elastizität und Fehlertoleranz designt. Da neuere Versionen von MPI an der ursprünglichen API und dem miteinhergehenden Design festhalten, müssen bei Anwendungen mit entsprechenden Anforderungen Einschränkungen in Kauf genommen werden. Vor allem handelt es sich hierbei um Performanceeinbußen und zusäzliche Komplexität für Entwickler und Anwendung. Hinzu kommt, dass Fehler-Handling und API-Design, unabhängig von spezielleren Anforderungen, den Entwicklungsprozess von fehlerfreien und effizienten MPI-Anwendungen verkomplizieren.

Eine klassische MPI-Anwendung beginn mit dem Aufruf des Konstrukts `MPI_Init` oder `MPI_Init_thread`. Letzteres erlaubt zusätzlich die Angabe, des benötigten Level an Multithreading-Unterstützung. Beide Konstrukte dienen der Initialisierung von MPI. Verbindungen zwischen allen beteiligten Prozessen werden so aufgebaut, dass alle Prozesse erreichbar sind. Jedoch wird nicht pauschal von jedem Prozess eine Verbindung zu jedem anderen aufgebaut, sondern generell gilt, dass Socket-Verbindungen nur nach Bedarf aufgebaut werden. Nach der Initialisierung von MPI können Konstrukte zur Kommunikation zwischen allen Prozessen benutzt und das eigentliche Problem gelöst werden. Die Anwendung endet aus Sicht von MPI mit dem Konstrukt `MPI_Finalize`. Neben diesem klassischen Ablauf ist es seit Version 2 möglich, dynamisch Einfluss auf die Anzahl an einer Anwendung beteiligten Prozesse zu nehmen. Zum Beispiel können einer laufenden Anwendung neue Prozesse hinzuzufügen werden. Dies ist jedoch nicht ohne Kosten, denn Synchronisation zwischen allen Prozessen ist bei jeder solchen Operation vonnöten.

Kommunikation zwischen zwei Prozessen findet generell mittels Varianten von `MPI_Send` und `MPI_Recv` statt. Für jeden Aufruf von `MPI_Send` muss im empfangenden Prozess ein passender Aufruf von `MPI_Recv` erfolgen. Mittels `MPI_Probe` können vorab Informationen, wie zum Beispiel die Größe, zu einer eingehenden Nachricht festgestellt werden. Neben der Kommunikation zwischen zwei Prozessen gibt es auch Operationen, die Kommunikation zwischen mehreren/allen Prozessen realisieren. Ein Beispiel für die sogenannten kollektiven Operationen ist `MPI_Bcast`. Dieses Konstrukt erlaubt das effiziente Senden einer Nachricht von einem Prozess an mehrere andere Prozesse. Auch die kollektiven Operationen benötigen jeweils einen API Aufruf durch jeden beteiligten Prozess. Zusätzlich zu diesen Methoden der zweiseitigen Kommunikation bietet MPI Konstrukte zur einseitigen Kommunikation im Sinne von Remote Memory Access (RMA). Um RMA

in MPI zu benutzen, müssen zunächst Speicherbereiche, hier als Window bezeichnet, für diesen Zweck reserviert werden. Folgend kann jedes Window mit einfachen Schreib- und Leseoperationen, einer Anzahl von atomaren Operationen, wie Compare and Swap, sowie Konstrukten zur Synchronisation manipuliert werden.

Multithreading steht in MPI in vier Formen zur Verfügung:

- `MPI_THREAD_SINGLE` Kein Multithreading
- `MPI_THREAD_FUNNELED` Die Anwendung ist multithreaded, jedoch laufen alle MPI Aufrufe über den Haupt-Thread.
- `MPI_THREAD_SERIALIZED` Die Anwendung ist multithreaded, jedoch finden keine MPI Aufrufe durch verschiedene Threads zum gleichen Zeitpunkt statt.
- `MPI_THREAD_MULTIPLE` Alle Threads können die API ohne weitere Einschränkungen verwenden.

Wie eine aktuelle US-Umfrage zur MPI Nutzung [19] zeigt, wird uneingeschränktes Multithreading (`MPI_THREAD_MULTIPLE`) häufig mit der Begründung von schlechterer Performance nicht eingesetzt.

Ähnlich sieht das Bild im Bezug auf MPI und Fehlertoleranz aus. Wie von Laguna [20] vorgestellt, gibt es eine Reihe von Ansätzen für fehlertolerante MPI-Anwendungen. Jedoch gibt es keine standardisierte Lösung und es ist immer ein Trade-off zwischen Entwicklungsaufwand, Performanceeinbußen und im Fehlerfall redundant auszuführenden Berechnungen zu treffen. Auf der einen Seite stehen Ansätze wie MPI Reinit [20], die einen minimalen Eingriff in die Programmstruktur erfordern, jedoch im Fehlerfall alle Berechnungen erneut durchführen. Auf der anderen Seite stehen Ansätze basierend auf Checkpointing. Checkpointing verfolgt die Idee, den Programmzustand zu gewählten Zeitpunkten zu sichern und die Ausführung im Fehlerfall von der letzten Sicherung aus fortzuführen. Hier entscheidet die Häufigkeit der Sicherungen über die Anzahl an Berechnungen, die im Zweifelsfall erneut durchgeführt werden müssen. Da sowohl Inhalt als auch geeignete Zeitpunkte für Sicherungen abhängig von der Anwendung sind, kann dies nicht durch eine Bibliothek übernommen werden. Eingriffe in die Programmstruktur sind entsprechend von Nöten.

Für MPI gibt es eine Reihe von Implementierungen. Neben den größeren Open-Source Implementierungen MPICH [21] und OpenMPI [22], die aus dem Bereich der Forschung entstanden sind, sind kommerziell entwickelte Implementierungen; unter anderem von Microsoft [23] und Intel [24] verfügbar. Im Rahmen der Experimente wurde OpenMPI verwendet. Entsprechend wird im Folgenden auf Eigenheiten von OpenMPI eingegangen.

Die MPI Implementierung OpenMPI wurde als Plattform für Forschung und Industrie erdacht. Das Design von OpenMPI ist von Modularität geprägt um zum einen experimentelle Ansätze, zum anderen proprietäre Lösungen erproben und produktiv verwenden zu können. Im Kern folgt die Entwicklung von OpenMPI der Open-Source Philosophie. Diese Ideen sind ausschlaggebend für die im Folgenden vorgestellte Architektur von OpenMPI. [25]

OpenMPI basiert im Kern auf der Modular Component Architecture (MCA). Die MCA ist im Hinblick auf HPC-Anwendungen designt und entsprechend leichtgewichtig. Aufgaben der Softwarearchitektur beschränken sich hauptsächlich auf das Finden, Laden und Freigeben von Services, die innerhalb von OpenMPI verwendet werden. Services übernehmen hierbei einzelne Aufgaben, wie das Übertragen von Bytes zwischen Prozessen. Abbildung 2.2 zeigt einen groben Überblick über die MCA im Kontext einer OpenMPI Anwendung. Anwendungen interagieren mit OpenMPI durch die MPI API. Diese wird von OpenMPI nur mittels minimaler Geschäftslogik realisiert und delegiert die meisten Aufrufe direkt zur MCA.

Die MCA setzt sich aus den Konzepten Service, Framework, Komponente und Module zusammen. Ein MCA Service besteht aus einem Framework, sowie den zugehörigen Komponenten und Modulen. Ein Framework kann als öffentliche Schnittstelle eines Services gesehen werden, während Komponenten als Implementierungen des Services zu verstehen sind. Typischerweise gibt es zu einem Framework mehrere Komponenten. Zum Beispiel gehört zu dem Byte Transfer Layer (BTL) Framework eine Komponente zur Realisierung von TCP und eine für Infiniband Kommunikation (OpenIB). Beide erlauben die byteweise Übertragung zwischen Prozessen, erreichen dies jedoch auf unterschiedlichen Wegen. Ein Modul ist eine Laufzeit-Instanz einer Komponente. Im Fall der BTL TCP-Komponente wird ein Modul pro TCP-Interface (z.B eth0 im Fall von Linux) instanziiert. Das bedeutet in der Praxis, dass auf Systemen ohne TCP-Interface keine Module instanziiert werden und somit die TCP-Komponente nicht genutzt werden kann. Sind jedoch mehrere TCP-Interfaces verfügbar, so gibt es auch entsprechend viele Module, die TCP-Kommunikation ermöglichen.

Die Abläufe innerhalb der MCA können durch Konfigurationsmöglichkeiten auf Framework- und Komponenten-Ebene gesteuert werden. Die Konfiguration erfolgt mit dem Start der Anwendung über Umgebungsvariable oder Kommandozeilenparameter. Zum Beispiel kann bestimmt werden, welche BTL Komponenten überhaupt geladen werden können, für welche TCP-Interfaces Module erzeugt werden oder über welche API Infiniband-Karten angesteuert werden. [26]

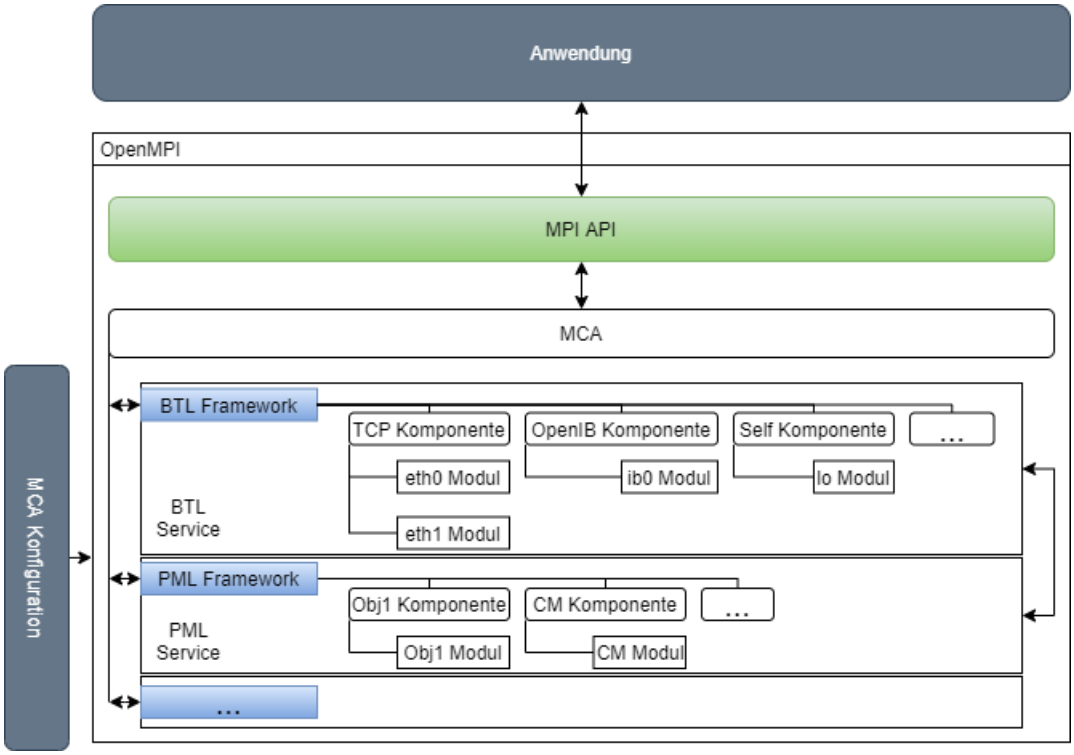

Abbildung 2.2.: Die Modular Component Architecture von OpenMPI[1]

Betrachtet man Peer-To-Peer Kommunikation innerhalb der MCA im Detail, so sind die Services, die durch das BTL und das PML-Framework geboten werden, von zentraler Bedeutung. Während der BTL Service, wie bereits erwähnt, Low-Level Byteübertragung realisiert, ist der PML Service für die verschiedenen Formen von Kommunikation zwischen zwei Prozessen, die von der MPI API benötigt werden, zuständig. Anforderungen, wie gepufferte oder synchrone Kommunikation, werden von dem PML geboten, während intern unterschiedliche Protokolle und BTL Komponenten gehandhabt werden. Folglich durchläuft ein `MPI_Send` Aufruf die MPI API Schicht, um in der MCA durch dem PML Service verarbeitet zu werden. Dieser verwendet intern wiederum den BTL Service, um die zu sendenden Daten letzten Endes zu übertragen. [28]

[1] Vergleiche Darstellung in [27]

3. Konzept

3.1. Aufgaben der Netzwerkschicht

Die zentrale Aufgabe einer APGAS Netzwerkschicht ist die Realisierung von Kommunikation von jedem Place zu jedem anderen Place. Angelehnt an Peer-To-Peer Kommunikation übernimmt die Netzwerkschicht Verwaltungsaufgaben wie die eindeutige Identifizierung von JVM Prozessen als spezifische Places und die Serialisierung beziehungsweise Deserialisierung von Java Objekten. Weitergehend gilt es Verbindungen zwischen Places aufzubauen und zu verwalten, um im Laufzeitsystem auf das Hinzufügen und Entfernen von Places reagieren zu können. Darüber hinaus dient die Netzwerkschicht als eine Factory für verschiedene Implementierungen des `finish` Konstrukts, da diese teilweise von der Implementierung der Netzwerkschicht abhängen.

3.2. Anforderungen an die Netzwerkschicht

Ausgehend von den Aufgaben der Netzwerkschicht ergeben sich weitere Anforderungen, die ausschlaggebend für Korrektheit und Performance sind. Es muss sichergestellt werden können, dass alle Places arbeitsbereit beziehungsweise terminiert sind, um mit der Abarbeitung der eigentlichen Anwendung beginnen zu können sowie um APGAS Anwendungen sauber beenden zu können. Da generell Java Objekte kommuniziert werden, müssen sowohl Serialisierung und Deserialisierung als auch der eigentliche Übertragungsprozess Daten beliebiger Größe verarbeiten können. Der Programmfluss einer APGAS Anwendung sollte zudem nicht durch Kommunikation blockiert werden können, da dies leicht zu vermeidbaren Performanceeinbußen führt.

Über die aus den Aufgaben der Netzwerkschicht erwachsenden Anforderungen hinaus, sind Anforderungen bezüglich Design, Struktur und Integration in das bestehende Laufzeitsystem zu realisieren. Um die Einsatzmöglichkeit verschiedener Netzwerkschichten für APGAS Anwendungen so einfach wie möglich zu halten, ist es sinnvoll, dass Netzwerkschichten dynamisch gegeneinander austauschbar sind.

Grundlage hierfür sind klar zu definierende Schnittstellen, die Funktionalität der Netzwerkschichten von der eigentlichen Implementierung abstrahieren. Aus der Tatsache, dass es sich bei APGAS um eine reine Java Bibliothek handelt, erwächst die Notwendigkeit, dass Bibliotheken, die innerhalb der neuen Netzwerkschicht Verwendung finden, auch in Java einsetzbar sind. Weiterhin ist es sinnvoll beim Einsatz von Bibliotheken deren Nutzung innerhalb der jeweiligen Netzwerkschicht zu isolieren. Grund hierfür ist, den Aufwand APGAS auf Plattformen zu verwenden die nicht alle Netzwerkschichten unterstützen können beziehungsweise auf denen nicht alle Bibliotheken verfügbar sind, gering zu halten. Unabhängig von den zuvor genannten Anforderungen gilt es weiterhin, Anpassungen am bestehenden Laufzeitsystem so minimal wie möglich zu halten, da andernfalls Wartung und Weiterentwicklung von APGAS erschwert werden würden.

3.3. Warum MPI?

Bei der Entwicklung einer neuen Netzwerkschicht kommen mehrere Ansätze in Frage. Am einen Ende des Spektrums steht der Ansatz die Entwicklung von Grund auf, basierend auf den von Java gegebenen Möglichkeiten zur Interprozesskommunikation, durchzuführen. Am anderen Ende stehen Ansätze, die eine Netzwerkschicht durch das Einbinden von C-Bibliotheken wie OpenMPI oder GASNet [29] realisieren.

Eine Implementierung der Netzwerkschicht in reinem Java, die die zuvor dargelegten Aufgaben und Anforderungen erfüllt und darüber hinaus zumindest in Bereichen eine Verbesserung zu dem bestehenden APGAS Netzwerkcode darstellt, würde den Rahmen dieser Arbeit übersteigen. Entgegen einer Netzwerkschicht auf Basis bereits etablierter Bibliotheken ist eine eigenentwickelte Netzwerkschicht nicht durch eine breite Anzahl von Anwendungen erprobt beziehungsweise vermessen. Sollte zum Beispiel der bestehende Netzwerkcode bessere Laufzeiten erreichen als eine neu entwickelte Netzwerkschicht, so ist es schwer zu sagen, ob dies in der Qualität des Netzwerkcodes begründet ist oder die Performance der neuen Netzwerkschicht ausbaufähig ist. Somit scheint es schwierig, eine sinnvolle Evaluation des bestehenden Netzwerkcodes mithilfe einer solchen Implementierung durchzuführen. Folglich wurde die Eigenentwicklung einer Netzwerkschicht in Java als Ansatz verworfen.

Ein möglicher Ansatz eine neue Netzwerkschicht zu implementieren ist die Verwendung der Bibliothek Global-Address Space Networking (GASNet) [29]. Bei GASNet handelt es sich um eine C-Bibliothek zur Netzwerkkommunikation. Desigt wurde die Bibliothek für den HPC-Bereich, mit Augenmerk auf dem PGAS Modell. GASNet wird unter anderem in den Laufzeitsystemen der Sprachen UPC [30]

und Chapel [31] sowie dem Programmiersystem Legion [32] eingesetzt. GASNet bietet APIs für RMA und Active Messaging. Bei Active Messaging handelt es sich um ein leichtgewichtiges Netzwerkprotokoll mit dem Ziel, durch Ausnutzung spezialisierter Hardware und der Vermeidung von Zwischenspeichern bei der Nachrichtenübertragung bessere Performance zu erreichen. Nachrichten im Active Messaging Protokoll enkodieren die Adresse eines Handlers zusammen mit dem eigentlichen Inhalt und erlauben so die direkte Verarbeitung bei Empfang von Nachrichten. RMA in GASNet arbeitet nach Möglichkeit mit direkter Hardwareunterstützung. Ist diese jedoch nicht gegeben, werden die APIs in Software gestützt auf Active Messaging realisiert. Für GASNet Anwender wird der Fokus primär auf RMA Kommunikation gelegt, dies spiegelt sich in veröffentlichten Performance-Daten und der Dokumentation wieder [33].

Die Entwicklung einer GASNet Netzwerkschicht für APGAS scheint auf den ersten Blick sinnvoll. Moderne PGAS Programmiersysteme setzen GASNet ein und Benchmarks auf der offiziellen Website [33] zeigen, dass GASNet RMA bessere Performance als MPI erreichen kann. Folglich scheinen Experimente mit einer auf GASNet basierenden Netzwerkschicht sinnvoll, um die Performance des ursprünglichen APGAS Netzwerkcodes zu bewerten. Betrachtet man jedoch den Aufwand, der zur Implementierung einer GASNet Netzwerkschicht betrieben werden muss, so scheint GASNet nicht die erste Wahl zu sein. Bestehende APIs in APGAS können weder RMA noch Active Messaging direkt ausnutzen, sondern zusätzlicher Aufwand müsste betrieben werden, um entweder Hazelcasts Kommunikations-APIs zu emulieren oder es müssten alternative Schnittstellen entwickelt und in großen Teilen des APGAS Laufzeitsystem zum Einsatz gebracht werden. Darüber hinaus fehlen Möglichkeiten GASNet in Java einzusetzen. Entsprechend müssten zusätzlich Java Native Interface (JNI) Bindings für GASNet entwickelt werden. Darüber hinaus ist nicht klar, ob auf potentiellen Plattformen eine native Implementierung verfügbar oder GASNet nur in emulierter Form verfügbar ist. Emulierte Unterstützung würde bedeuten, dass GASNet gestützt auf MPI oder direkter IP-Kommunikation arbeitet. Wird GASNet emuliert, so ist mit Performanceeinbußen zu rechnen.

Es gibt im Bereich des High Performance Computing kein etablierteres Programmiersystem als MPI. Entsprechend bietet es sich an, Vergleichbarkeit mit dem verbreitetsten Maßstab herzustellen. Nachteile von MPI, wie die nicht unerhebliche Komplexität für Anwendungsprogrammierer, schlagen hier nicht wirklich ins Gewicht, da die Netzwerkschicht komplett innerhalb von APGAS gekapselt ist. Typischerweise bieten MPI Implementierungen keine direkte Unterstützung für Java. OpenMPI hingegen unterstützt die Verwendung in Java mittels JNI Bindings. MPI stellt mit breiter Unterstützung von Netzwerkhardware / -technologien und einem erprobten Set von Tuning-Optionen bessere Performance in Aussicht. Kann im Ver-

gleich zwischen bestehendem Netzwerkcode und einer neuen MPI Netzwerkschicht bessere Performance zu Gunsten der neuen Netzwerkschicht gezeigt werden, so deutet dies auf Defizite im bestehenden Netzwerkcode hin. Stellt sich umgekehrt die Performance einer MPI Netzwerkschicht als schlechter heraus, so können nach Ausschluss von Implementierungsfehlern Rückschlüsse auf die Qualität des bestehenden Netzwerkcodes getroffen werden. Es kann also ungeachtet der tatsächlichen Ergebnisse im Vergleich ein aussagekräftiges Bild über den bestehenden Netzwerkcode sowie eine potentielle MPI Netzwerkschicht gewonnen werden.

Von Interesse um eine mögliche Verwendung von MPI einzuordnen ist eine aktuelle Umfrage [19] bezüglich des Einsatzes von MPI in US exascale Projekten. Diese bestätigt die Aktualität und Bedeutung von MPI für Projekte im HPC-Bereich. So setzen über 90% der Projekte MPI in irgendeiner Form ein. Weiterhin wird festgestellt, dass die meisten Projekte mit MPI durch eine Bibliothek oder Abstraktionsschicht interagieren und die Kommunikations-Features von MPI für über die Hälfte komplett ausreichen. Die Verwendung von MPI durch eine Bibliothek beziehungsweise Abstraktionsschicht deckt sich mit dem im Rahmen dieser Arbeit geplanten Einsatzzweck als Basis für eine neue APGAS Netzwerkschicht.

Die Verwendung von MPI als Basis der neuen Netzwerkschicht liegt schlussendlich in der Aussicht auf bessere Performance, ohne die Probleme und den Entwicklungsaufwand, die GASNet oder eine von Grund auf entwickelte Netzwerkschicht mit sich bringen würden. Weiterhin lässt sich das gesetzte Ziel der Evaluation der Hazelcast Netzwerkschicht im Vergleich mit einer neuen MPI Netzwerkschicht verwirklichen.

3.4. MPI in Java Anwendungen

Wie zuvor beschrieben war einer der Kernpunkte, um eine Bibliothek für die Entwicklung der Netzwerkschicht verwenden zu können, die Unterstützung für den Einsatz in Java. MPI ist laut Carpenter [34] seit der Entwicklung von Java von Interesse für die Entwicklung von HPC-Anwendungen mit Java. Entsprechend gibt es eine Reihe von Ansätzen MPI oder vergleichbare Funktionalität in Java verfügbar zu machen. Viele dieser Projekte wurden auf Basis bereits bestehender, teilweise standardisierter, Java APIs realisiert. Die APIs vereint das Ziel funktional der MPI API zu entsprechen. Wobei der Umfang replizierter Schnittstellen und das zu Grunde liegende Design die hauptsächlichen Unterscheidungsmerkmale zwischen de APIs darstellen. Im Kern wurden zur Implementierung dieser APIs zwei Ansätze verfolgt. Zum einen wurden APIs in reinem Java implementiert. Zum anderen wurden Implementierungen basierend auf C-Bibliotheken mithilfe vom Java Native Interface (JNI), welches den Einsatz selbiger Bibliotheken in Java ermöglicht,

entwickelt. Beispiele für entsprechende Projekte sind unter anderem Java Object-Passing Interface (JOPI) [35], mpiJava [36] und MPJ [37].

Im Allgemeinen ist festzustellen, dass viele Projekte rein akademischer Natur sind und/oder durch geringe Unterstützung an Relevanz verloren haben. So gibt es für viele APIs genau eine Implementierung, die meistens nur die Funktionalität älterer MPI-Standards bietet und nicht mehr aktiv weiterentwickelt wird.

Betrachtet man die verschiedenen Ansätze MPI in Java einzusetzen, von dem Gesichtspunkt Performance aus, zeigt sich, dass die verfolgten Wege Nachteile im Vergleich zu einem Einsatz von MPI in C/C++ mit sich bringen. Für einen rein auf Java basierenden Ansatz sind, obwohl dies auf den ersten Blick nicht offensichtlich ist, ebenfalls Nachteile in Bezug auf die Performance zu erwarten. In nativem Java werden zur Kommunikation nur Sockets beziehungsweise Remote Method Invocation (RMI) direkt unterstützt. Somit kann High-Perfomance Netzwerkhardware, wie Infiniband, nicht voll ausgenutzt werden. Soll trotzdem Infiniband-Hardware verwendet werden, so kann eine wie bei Ethernet übliche IP-Schnittelle emuliert werden. Diese Emulation hat den Nachteil, dass zusätzlicher Aufwand in Form des IP-Stacks betrieben werden muss. Alternativ muss auf die Unterstützung von C-Bibliotheken zurückgegriffen werden, was wiederum konträr zur ausschließlichen Verwendung von nativem Java steht. Details zur Verwendung und Perfomance von Infiniband in Java können dem von Nothaas et al. durchgeführten Benchmark [38] entnommen werden. Alternative auf JNI basierenden Ansätze haben hingegen neben dem geringen zusätzlichen Aufwand bei Aufrufen vor allem Problemen mit Speicher- und Lebenszeit-Management. So können weder Java Byte-Arrays noch ByteBuffer [39] als uneingeschränkter Ersatz für in C allozierte Speicher zum Einsatz als Puffer dienen. Unabhängig von gewählter Repräsentation für Puffer ergibt sich die Notwendigkeit, entweder Lebenszeit von Puffern bis zum Abschluss von einer Operation zu verlängern oder Kopien zu erstellen. Neben den offensichtlichen Problemen mit JNI basierten Ansätzen bieten diese jedoch die Möglichkeit, ein breites Spektrum an Bibliotheken und Systemen zum Einsatz zu bringen. Darunter finden sich hardwarenahe Bibliotheken, die die Verwendung von Spezialhardware ohne Performanceeinbußen ermöglichen oder auch erprobte Systeme zur Kommunikation zwischen Prozessen, die effiziente Implementierungen für Kommunikationsmuster und Hardwareunterstützung gleich mit sich bringen. Diese Vorteile scheinen die Probleme im Zusammenhang mit JNI aufzuwiegen, vor allem, da reine Java Implementierungen keine wirkliche Alternative zu sein scheinen. Folglich die Entscheidung für den Einsatz von MPI durch JNI.

Mit den Entscheidungen für MPI und JNI galt es eine konkrete Implementierung von MPI auszuwählen. Es wurde die MPI-Implementierung OpenMPI vor allem aufgrund der bereits von Vega-Gisbert et al. entwickelten JNI-Bindings [3] verwendet. Die JNI-Bindings von Vega-Gisbert et al. werden zusammen mit dem

OpenMPI Sourcecode instandgehalten und die MPI-3.1 Standard API wird weitestgehend unterstützt. Die von Vega-Gisbert et al. implementierte API basiert auf mpiJava. Ziel der Bindings war es, Aufrufe aus der Java-Umgebung möglichst direkt auf die C-API von OpenMPI zu übertragen. Um die Bindings verwenden zu können, müssen diese gegen ein Java Development Kit (JDK) zusammen mit dem eigentlichen Quellcode von OpenMPI gebaut werden. Wie zu erwarten hat der Einsatz der OpenMPI-Bindings im Vergleich zu einer nativen C Anwendung generell Nachteile im Bezug auf die Performance. Vega-Gisbert et al. zeigen jedoch, dass es durchaus Szenarien gibt, in denen vergleichbare Performance erreicht werden kann.

3.5. Überblick

Aus Aufgaben und Anforderungen sowie den zuvor diskutierten Entscheidungen ergibt sich die folgend dargelegte Vorgehensweise. Der Erweiterung von APGAS um eine neue Netzwerkschicht geht die Restrukturierung des bestehenden Netzwerkcodes voraus. In diesem Zug ist ein Umbau der bestehenden Schnittstellen notwendig, um auf dessen Basis eine neue Netzwerkschicht realisieren zu können. Am Ende dieses Prozesses ist das Ziel APGAS Anwendungen die Auswahl zwischen zwei austauschbaren Netzwerkschichten zu bieten. Zum einen eine Hazelcast Netzwerkschicht, die im Zuge der Restrukturierung aus bestehendem Netzwerkcode realisiert wird, zum anderen eine neu implementierte MPI Netzwerkschicht.

Der Kern des APGAS Laufzeitsystems übernimmt aktuell noch einige Aufgaben die der Netzwerkschicht zuzuordnen sind. Neben dem Warten auf den Abschluss des Verbindungsaufbaus sind die Auswahl der Finish Implementierung sowie Teile der Implementierung eines resilienten Finishs im Kern des Laufzeitsystems implementiert. Um die Zielsetzungen bezüglich Austauschbarkeit umzusetzen und passende Schnittstellen für Netzwerkschichten entwickeln zu können, wurde diese Funktionalität als Teil der Hazelcast Netzwerkschicht in den restlichen Netzwerkcode integriert. Weiterhin wurden einige Schnittstellen angepasst, um Abhängigkeiten auf Implementationsdetails der Netzwerkschicht und Hazelcast innerhalb des Laufzeitsystems zu eliminieren. Sowohl Resilienz als auch der Einsatz von Kryo zur Serialisierung sind auf Basis von Hazelcast implementiert. Als Konsequenz sind beide Features Teil der Hazelcast Netzwerkschicht.

Innerhalb der neu erstellten Netzwerkschicht wird Aufbau und Verwaltung der Verbindungen zwischen den JVMs durch MPI realisiert. Schnittstellen zur Peer-To-Peer Kommunikation lassen sich ebenfalls direkt durch entsprechende MPI-Konstrukte realisieren. Serialisierung ist hingegen mittels der Standard Java Mechanismen umgesetzt. Da Standard Serialisierung APIs nur das Schreiben in Ar-

rays ermöglichen, die JNI-Bindings jedoch teilweise nur mit `ByteBuffer` arbeiten, ist Kopie von Array zu `ByteBuffer` bei jedem Senden/Empfangen notwendig. Um diesen Aufwand zu eliminieren wurden eigene Varianten der für De-/Serialisierung zuständigen Klassen implementiert. Resilienz wird von der MPI Netzwerkschicht nicht unterstützt, denn im Gegensatz zu Hazelcast verfügt MPI über keine standardisierten Mechanismen, um die Anwendung über den Ausfall eines Prozesses zu informieren und dies programmatisch verarbeiten zu können. Weiterhin umfassen die verwendeten OpenMPI JNI-Bindings keine Variante der ergänzend zu dem MPI Standard entwickelten APIs, die zur Entwicklung resilienter Anwendungen eingesetzt werden könnten. Die Kryo Serialisierung ist in der ursprünglichen Codebase als Plugin für Hazelcasts Serialisierung implementiert und wurde daher nicht in der MPI Netzwerkschicht eingesetzt.

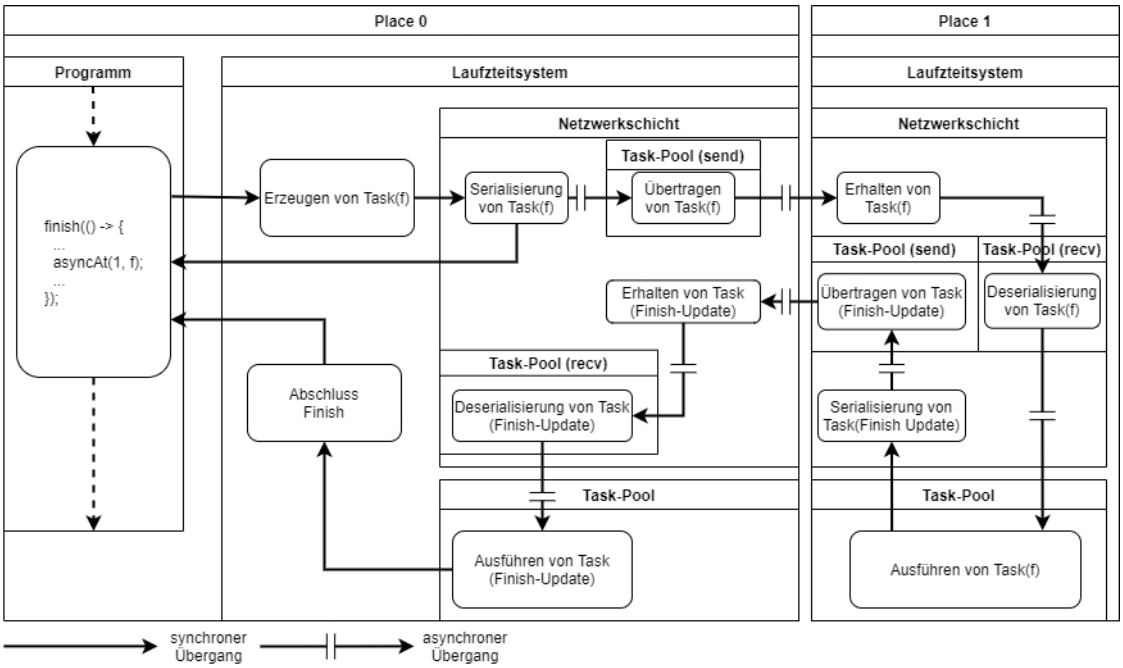

Abbildung 3.1.: Das `asyncAt` Konstrukt im Kontext der neuen Netzwerkschicht.

Aus Sicht einer APGAS Anwendung bestimmt die Netzwerkschicht also, ob Kryo Serialisierung beziehungsweise Resilienz verfügbar sind. Alle anderen Features von APGAS sind unabhängig von der Netzwerkschicht verfügbar. Betrachtet man die Implementierung der APGAS Konstrukte, so ist deren Verhalten unabhängig von der gewählten Netzwerkschicht. In Abbildung 3.1 ist exemplarisch veranschaulicht an welcher Stelle das Konzept Netzwerkschicht innerhalb des APGAS Laufzeitsystems einzuordnen ist. Weiterhin wird ein Einblick in die Verarbeitung von Tasks innerhalb der MPI Netzwerkschicht gegeben. Beim Vergleich zwischen dem in Abschnitt 2.1 beschriebenen Ablauf (siehe Abbildungen 2.1) mit Abbildung 3.1, un-

terscheidet sich dieser nur innerhalb der Netzwerkschicht. Zum einen sind Deserialisierung und Serialisierung explizite Schritte innerhalb der MPI Netzwerkschicht. Zum anderen wird Nebenläufigkeit während des Sendens beziehungsweise Empfangens von Tasks durch Task-Pools realisiert. Hintergrund dieser Unterschiede ist, dass sowohl Nebenläufigkeit bei der Übertragung, als auch Serialisierung innerhalb von Hazelcast und nicht als Teil des Netzwerkcodes von APGAS realisiert waren.

Wie in Abbildung 3.1 zu sehen übernehmen die Task-Pools `send` und `recv` die Übertragung von zu sendenden Tasks sowie die Deserialisierung von empfangenen Tasks. Die Task-Pools sind wichtig, um asynchrone Übergänge in den Abläufen analog zu der Hazelcast Netzwerkschicht zu realisieren. Andernfalls würde parallel ausführbare Arbeit zur Kommunikation sequenzialisiert werden und somit die Performance negativ beeinflusst werden. Der `send` Task-Pool kann nur die eigentliche Übertragung von Tasks übernehmen, da bis zum Abschluss der Serialisierung ursprüngliche Java Objekte verwendet werden. Würde nach dem Erzeugen eines Tasks der eigentliche Programmfluss fortgesetzt werden, so führte dies durch eine Race Condition zwischen dem Abschluss der Serialisierung und potentieller Mutation der Java Objekte des Tasks zu inkorrekten Programmergebnissen. Nachrichten können aufgrund der verwendeten MPI APIs nur nacheinander entgegengenommen werden. Um die Nachrichten dennoch möglichst effizient verarbeiten zu können, wird die Deserialisierung durch den `recv` Task-Pool parallelisiert. Der zum Ausführen der Tasks eingesetzt Task-Pool konnte nicht anstelle der beiden neuen Task-Pools wiederverwendet werden. Anpassungen an der `asyncAny` Lastenbalancierung, die voraussichtlich mit Performanceeinbußen einhergehen würden, wären andernfalls unvermeidbar gewesen.

Für den allgemeinen Programmablauf ergeben sich keine Änderungen durch die MPI Netzwerkschicht. APGAS Anwendungen, die die MPI Netzwerkschicht einsetzen, müssen jedoch alle benötigten JVM Prozesse mithilfe des Kommandozeilen-Tools `mpirun` starten. In diesem Zuge muss die bereits existierende Launcher Implementierung `NoLauncher` verwendet werden, um das Erzeugen zusätzlicher Prozesse durch APGAS zu verhindern. Damit auch der Programmstart unabhängig von der Netzwerkschicht dem gleichen Ablauf folgen kann und trotz zusätzlicher Komplexität möglichst einfach für den Endanwender bleibt, wurde ein neues Skript `apgasrun.sh` entwickelt. Das Skript übernimmt unter anderem die Auswahl der Netzwerkschicht, den Aufruf und die Konfiguration von `mpirun`, sowie das Setzen vergleichbarer Optionen beim Einsatz der Hazelcast Netzwerkschicht.

4. Implementierung

Ausgehend von dem zuvor vorgestellten Konzept werden in diesem Abschnitt die für eine APGAS Netzwerkschicht relevanten Schnittstellen und die Implementierung der MPI Netzwerkschicht genauer beschrieben. Weiterhin wird ein kurzer Überblick über die Hazelcast Netzwerkschicht gegeben.

Im Kern sind die für APGAS Netzwerkschichten relevanten Schnittellen als Teil der Java Interfaces `Backend` und `Transport` definiert. Während das `Backend` Interface die zentrale Instanz einer Netzwerkschicht darstellt, ist das `Transport` Interface für Verbindungsmanagement und Datenübertragung zuständig. Das `Backend` Interface dient als Factory für Implementierungen von `Transport` und `Finish`. Darüber hinaus werden Schnittstellen definiert, die dem Laufzeitsystem Zugriff auf Eigenschaften der Netzwerkschicht ermöglichen. Aktuell beläuft sich dies auf den Prefix `StartupCommand`, der von `Launcher` Komponenten zum Starten neuer Prozesse verwendet wird. `Transport` bietet Schnittstellen, um Verbindungen auf- beziehungsweise abzubauen, Places zu identifizieren und Daten zwischen Places zu übertragen.

Im Folgenden wird die MPI Netzwerkschicht, die sich aus der Implementierung der zuvor beschriebenen Interfaces zusammensetzt, beschrieben. Kern der Implementierung ist die `MPIBackend` Komponente, welche die Schnittstellen des `Backend` Interfaces realisiert. Die zugehörige `Transport` Implementierung `MPITransport` ist momentan nur mit der `Finish` Implementierung `DefaultFinish` kompatibel, entsprechend werden von `MPIBackend` auch nur diese beiden Komponenten instantiiert. Wie bereits beschrieben ist eine Aufgabe von `MPITransport` die Identifikation von Places. In der aktuellen Implementierung wird zur Identifikation MPIs Rank-Nummer verwendet.

Für den Einsatz der MPI Netzwerkschicht wird zunächst `MPITransport` durch das `MPIBackend` instantiiert. Nach diesem Schritt muss die `MPITransport` Instanz, noch konfiguriert beziehungsweise initialisiert werden. In diesem Schritt wird MPI sowie benötigte Threads beziehungsweise Thread-Pools initialisiert und konfiguriert. Daraufhin wartet die Anwendung auf den Verbindungsaufbau der konfigurierten Anzahl von Places. Folgend ist die Netzwerkschicht bereit zum Senden und Empfangen von Nachrichten.

Die Nachrichtenübertragung folgt im Allgemeinen einem Ablauf wie er in Abschnitt 3 dargestellt wurde (siehe Abbildung 3.1). Ist ein zu übertragender Task bereittbs auf dem Ziel-Place wird dieser direkt in den `ForkJoinPool` zum Ausführen eingereiht. Andernfalls werden Tasks, wie zuvor beschrieben, serialisiert, um daraufhin übertragen zu werden. Um die Serialisierung performanter zu gestalten, wird vorab ein `ByteBuffer` fixer Größe alloziert. Dieser Puffer wird inkrementell mit der Byterepräsentation des Java Objekts befüllt. Die Größe des Puffers ist hierbei konfigurierbar und beträgt standardmäßig 8096 Byte. Da sich die für eine Nachricht benötigte Größe vorab nicht sinnvoll feststellen lässt, können Kopien und Reallozierung nur dann vermieden werden, wenn der Puffer bereits vorab groß genug ist. Im Umkehrschluss ist es notwendig, dass der Puffer bei Bedarf vergrößert werden kann. Um die Übertragung gemäß den zuvor gestellten Anforderungen nebenläufig zu realisieren, ist der Task-Pool `send` durch einen `ForkJoinPool` implementiert.

Der Ablauf auf Empfängerseite kann im Detail anhand von Listing 4.1 nachvollzogen werden. Fehlerbehandlung und nebensächliche Details wurden aus dem Auszug aus Gründen der Anschaulichkeit gekürzt. Die Schleife wird von einem Thread, der mit der Initialisierung der Netzwerkschicht gestartet wird, bis zum Ende der Anwendung ausgeführt. Entsprechend wird auf jedem Place eine Instanz der Schleife ausgeführt. Zu Beginn wird die Verfügbarkeit einer eingehenden Nachricht geprüft. Diese wird im Folgenden in einen neuen Puffer übertragen.Zuletzt wird die im Puffer enthaltene Byte-Repräsentation der Nachricht an den `ForkJoinPool recv` übergeben. Dieser dient, wie in Abschnitt 3 beschrieben, der parallelen Deserialisierung von Nachrichten (vergleiche Listing 4.1 Zeile 15-18).

Der Einsatz von `MPI.COMM_WORLD.recv(...)` (`MPI_Recv`) ist darin begründet, dass das Konstrukt genau dann zum eigentlichen Programmfluss zurückkehrt, wenn der übergebene Puffer mit einer Nachricht gefüllt wurde. Alternative APIs wie `MPI.COMM_WORLD.iRecv(...)` würden lediglich Mehraufwand auf Seiten der Netzwerkschicht mit sich bringen, denn `MPI.COMM_WORLD.iProbe(...)` verhindert, dass die Schleife durch mehrere Threads auf einem Place ausgeführt werden kann. Die Parallelisierung ist nicht möglich, da aus Sicht eines Threads nicht gewährleistet ist, dass die von `MPI.COMM_WORLD.iProbe(...)` erhaltenen Informationen zu einer folgend empfangenen Nachricht passen. Diese Garantie hält nur dann, wenn es keine Möglichkeit gibt, dass ein anderer Thread Nachrichten zwischen `MPI.COMM_WORLD.iProbe(...)` und einer Variante von `MPI.COMM_WORLD.recv(...)` empfängt.

Auf Seite des Sendenden wird analog zum Empfangen `MPI.COMM_WORLD.send(...)` eingesetzt. Alternative Konstrukte wie `MPI.COMM_WORLD.iSend(...)` ziehen manuelles Speichermanagement in größerem Ausmaß nach sich. Der Rückgabewert

von `MPI.COMM_WORLD.iSend(...)` ist ein Objekt vom Typ `Request`. Um zu vermeiden, dass Speicher freigegeben wird, während dieser noch von MPI verwendet wird, müssen über `Request` Objekte Referenzen auf verwendete Speicherressourcen gehalten werden. Dies muss so lange geschehen, bis sicher ist, dass jegliche Arbeit auf den jeweiligen Speicherressourcen abgeschlossen wurde. In der Praxis bedeutet dies, dass alle `Request` Objekte gesammelt werden müssen. Weiterhin gilt es die gesammelten `Request` Objekte regelmäßig zu überprüft, ob diese freigegeben werden können.

```
1  long sleepMS = Long.getLong(Config.APGAS_MPI_RECV_SLEEP_MS, 5);
2  for (;;) {
3    Status probeStatus = MPI.COMM_WORLD.iProbe(MPI.ANY_SOURCE, MPI.ANY_TAG);
4    if (probeStatus == null) { // Keine ausstehende Nachricht
5      TimeUnit.MILLISECONDS.sleep(sleepMS);
6    } else { // Nachricht kann empfangen werden
7      int tag = probeStatus.getTag();
8      int source = probeStatus.getSource();
9      int byteCount = probeStatus.getCount(MPI.BYTE);
10     ByteBuffer bytes = MPI.newByteBuffer(byteCount);
11
12     MPI.COMM_WORLD.recv(bytes, byteCount, MPI.BYTE, source, tag);
13     switch (tag) {
14     case TAG_RUNNABLE:
15       recvPool.execute(() -> {
16         SerializableRunnable f = deserialize(bytes);
17         f.run();
18       });
19       break;
20     case TAG_TERMINATE:
21       MPI.Finalize();
22       shutdown();
23       return;
24     }
25   }
26 }
```

Listing 4.1.: Receive Loop der MPI Netzwerkschicht (aus Gründen der Anschaulichkeit gekürzt).

Erreicht eine APGAS-Anwendung ihr Ende, so wird dies durch die Laufzeitumgebung an die `Transport` Instanz auf Place 0 mitgeteilt. Über den Aufruf von `shutdown()` werden nun alle Places benachrichtigt, dass das Ende der Anwendung erreicht ist und die Kommunikation eingestellt werden kann.

Die MPI Netzwerkschicht setzt aus Performancegründen die MPI Konfigurationsoptionen `mpi_assert_allow_overtaking` und `mpi_assert_exact_length`. Die

Option `mpi_assert_allow_overtaking` setzt die vorgeschriebene Reihenfolgegarantie bezüglich der Send- und Recv-Konstrukte außer Kraft. Sprich, finden mehrere Aufrufe der Konstrukte statt, so werden Konstrukt-Paare beliebig anstatt in strikter Reihenfolge zugeordnet. Da die Hazelcast Netzwerkschicht keine Reihenfolgegarantie bietet macht es Sinn, keine strikte Reihenfolge durch MPI zu erzwingen. Ist `mpi_assert_exact_length` gesetzt, so wird vorausgesetzt, dass beim Empfangen einer Nachricht die Puffergröße immer der Nachrichtengröße entspricht. Im Fall der MPI Netzwerkschicht entspricht die Puffergröße der Nachrichtengröße, denn vor dem Empfangen wird immer ein neuer passender Puffer alloziert. Beide Optionen beeinflussen ausschließlich Peer-To-Peer Kommunikation.

In der ursprünglichen Variante des Receive Loop (Listing 4.1) wurde die blockierende API `MPI_Probe` beziehungsweise `MPI.COMM_WORLD.probe(...)` verwendet. Als Konsequenz haben die für das Entgegennehmen von Nachrichten zuständigen Threads, unabhängig vom Nachrichtenaufkommen, die volle Auslastung eines Kerns verursacht. Auch der dokumentierte MCA Konfigurationsparameter `mpi_yield_when_idle`, welcher von aktivem Warten auf eine CPU sparende Alternative umschalten sollte, war ohne Effekt. Als Lösung wurde die nicht blockierende API `MPI.COMM_WORLD.iProbe(...)` verwendet und ein konfigurierbares Warteintervall eingefügt. In dieser Zeit schläft der Thread, sollten keine Nachrichten zum Empfang ausstehen (vergleiche Listing 4.1 Zeile 3-5).

Die MPI Netzwerkschicht kommt mit den im Folgenden aufgelisteten zusätzlichen Konfigurationsparametern:

- `apgas.mpi.serialization.buffer.defaultsize`
  Standard Puffergröße die zu Beginn jeder Serialisierung alloziert wird.

- `apgas.mpi.send.threads`
  Anzahl Threads mit denen der `recv` Thread-Pool initial zu arbeiten beginnt.

- `apgas.mpi.send.threads.max`
  Maximale Anzahl Threads die der `recv` Thread-Pool einsetzten kann.

- `apgas.mpi.recv.threads`
  Anzahl Threads mit denen der `send` Thread-Pool initial zu arbeiten beginnt.

- `apgas.mpi.recv.threads.max`
  Maximale Anzahl Threads die der `send` Thread-Pool einsetzten kann.

- `apgas.mpi.recv.sleep`
  Anzahl Millisekunden die immer dann gewartet wird, wenn gerade keine Nachricht empfangen wurde und deren Verarbeitung ansteht.

Wie bereits in Abschnitt 3 beschrieben, wurde zusätzlich zur Entwicklung der Netzwerkschicht das Bash Skript `apgasrun.sh` erstellt. Das Skript dient der Vereinfachung von Konfiguration und dem Starten von APGAS Anwendungen in einer Terminalumgebung. Die Konfiguration für eine Anwendung wird von `apgasrun.sh` aus Kommandozeilenparametern und Umgebungsvariablen, die durch den Slurm Workload Manager [40] gesetzt werden, zusammengefügt. Das Skript ist notwendig, um eine einheitliche Schnittstelle für beide Netzwerkschichten zu bieten. Im Fall der MPI Netzwerkschicht erfordert OpenMPI das Setzen einiger Optionen beim Starten der Prozesse mittels `mpirun`. Neben zu verwendender Netzwerkschicht und Hardware (Ethernet, Infiniband oder Auto) kann die Anzahl von Places und Threads sowie Debug- und Profile-Optionen gesetzt werden. Eine gezielte Verteilung von Places auf Knoten ist momentan nur über die von Slurm definierten Umgebungsvariablen möglich. Sind diese nicht gesetzt, werden alle Places auf dem aktuellen Knoten gestartet.

5. Experimente

Im Rahmen dieser Arbeit wurde eine Anzahl von Experimenten durchgeführt, um sowohl den restrukturierten Hazelcast Netzwerkcode als auch die neu entwickelte MPI Netzwerkschicht evaluieren zu können. Zu diesem Zweck wurden Laufzeitmessungen, die die beiden Netzwerkschichten gegenüberstellen, durchgeführt. Von Interesse ist das Verhalten beider Netzwerkschichten unter verschiedenen Kommunikationsmustern sowie bei unterschiedlichem Kommunikationsvolumen. Da OpenMPI intern verschiedene Komponenten für die Unterstützung von Ethernet und Infiniband verwendet (siehe Abschnitt 2.3), wurden die Experimente jeweils mit beiden Technologien durchgeführt. Grenzen der Netzwerkschichten sind, wenn auch kein primärer Faktor für die Auswahl der durchgeführten Experimente, dennoch von Interesse. Beispiel sind Fragestellungen wie nach der Obergrenze an unterstützten Places oder auch die maximale Menge an Nachrichten/Daten, die übertragen werden können.

Die Performance der Netzwerkschichten wurde empirisch anhand von Laufzeitmessungen im Rahmen der Benchmarks Betweenness Centrality (BC) und Unbalanced Tree Search (UTS) ermittelt. Für die Auswahl der Benchmarks sprechen die komplett verschiedenen Kommunikationsmuster und Datenmengen. Wie im weiteren Verlauf dieses Abschnitts beschrieben, werden Nachrichten im Bereich von einzelnen Bytes hin zu mehreren MB pro Nachricht übertragen. Neben einfacher Peer-To-Peer Kommunikation kommen komplexere Operationen, wie Scatter und Reduktion, zum Einsatz. Bei Scatter handelt es sich um das Verteilen eines Datensatzes von einem Place an alle Places. Reduktion hingegen dient dem Zusammenfügen von Teilergebnissen aller Places an einem Ziel. Beide Benchmarks sind jeweils mit Lastenbalancierung basierend auf `asnycAny` und dem GLB-Framework implementiert. Da das GLB-Framework unter anderem eine größere Anzahl Places erfordert und generell andere Kommunikationseigenschaften aufweist, wurden beide Varianten betrachtet. Weitergehend wurden Latenz und Bandbreite in Abhängigkeit von zu übertragender Datengröße vermessen. Die zugehörigen Experimente für Latenz und Bandbreite werden im Folgenden als LAT und BW bezeichnet. Bandbreite ist ein Maß für die Menge an Daten, die in einem bestimmten Zeitraum übertragen werden kann. Latenz hingegen beschreibt die Zeit, die ein Datenpaket bis zum Ziel benötigt. Die Experimente LAT und BW dienen der systematischen Untersu-

chung des Zusammenhangs von Nachrichtengröße und Performance, da die bei den Benchmarks BC und UTS gemessenen Laufzeiten kein klares Bild vermitteln.

5.1. Aufbau

Die Experimente wurden auf dem "Cluster für die wissenschaftliche Datenverarbeitung" der Universität Kassel durchgeführt. 12 Knoten mit jeweils 2 Intel Xeon E5-2643-v4 CPUs und 256 GB Arbeitsspeicher standen zur Verfügung. Jede der Intel CPUs besteht aus 6 physischen Kernen. Hyperthreading war zum Zeitpunkt der Experimente deaktiviert. Zur Netzwerkkommunikation sind Ethernet und Infiniband Schnittstellen vorhanden. Für Infiniband sind auf Mellanox ConnectX-3 56 Gb/s basierende Supermicro Netzwerkkarten verbaut. Angesteuert wurden die Knoten über ein Batch-System.

Die für diese Arbeit modifizierte APGAS Implementierung wurde mit dem Java OpenJDK Version 12.0.1 Update 12 gebaut. Die Hazelcast Bibliothek wurde in der offiziellen Release Version 3.10.6 verwendet. Für die MPI Netzwerkschicht hingegen wurde das Quellcode-Release der OpenMPI Version 4.0.3 selbst konfiguriert und gebaut. Zu diesem Zweck wurde die GNU Compiler Collection (GCC) Version 7.1.0 verwendet. Details zur OpenMPI Build-Konfiguration sind im Anhang A dokumentiert.

Die Experimente wurden mithilfe des Batch-Systems Slurm [40] auf dem Cluster ausgeführt. Wie in Abschnitt 4 beschrieben, wurde ein Skript `apgasrun.sh` entwickelt, um unter anderem die Konfiguration und das Starten der APGAS Anwendung zu vereinheitlichen. Für jeden Durchlauf eines Experiments wird entsprechend von Slurm eine Instanz des `apgasrun.sh` Skripts mit der Benchmark Konfiguration ausgeführt.

Im Rahmen jedes Experiments wurden die Laufzeitmessungen jeweils einmal mit Ethernet und einmal mit Infiniband durchgeführt. Es war nur möglich Infiniband via IP über Infiniband (IPoIB) im Datagram Modus [41] zu nutzen. Soll heißen, die Infiniband Karten wurden durch Treiber Emulation wie eine IP basierte Netzwerkschnittstelle eingebunden und konnten auch nur über eben diese Schnittstelle angesprochen werden. Zwar unterstützt OpenMPI direkte Infiniband Kommunikation, jedoch konnte diese aufgrund der Konfiguration des verwendeten Clusters nicht genutzt werden. Die OpenMPIMCA Komponenten für Infiniband (BTL OpenIB und PML UCX) meldeten Fehler beim direkten Zugriff auf die Hardware zurück und konnten entsprechend nicht initialisiert werden. Hazelcast und somit auch die entsprechende Netzwerkschicht unterstützen von Grund auf nur IP basierte Kommunikation. Infiniband Hardware wurde also nur über die IPoIB

Emulation verwendet. Direkter Zugriff auf die Hardware verspricht im Vergleich zu IPoIB bessere Performance, denn für IPoIB ist der IP-Stack notwendig, während bei direktem Zugriff dieser Overhead wegfallen würde. Im Folgenden ist mit Infiniband immer IP über Infiniband im Datagram Modus gemeint. Unterschiede bezüglich zu erwartender Performance im Vergleich von IPoIB mit direktem Einsatz von Infiniband Hardware können beispielhaft anhand der von Rahman et al. [42] durchgeführten Experimente nachvollzogen werden.

Wie in Abschnitt Grundlagen 2.1 beschrieben, müssen, sofern das GLB-Framework verwendet wird, mehrere Places auf einer Maschine gestartet werden. Nur so kann mit dem GLB-Framework ein Multicore-Rechner ausgelastet werden. Darüber hinaus erfordert das Framework eine größere Anzahl von APGAS Threads als eine vergleichbare Anwendung die `asyncAny` zur Lastenbalancierung einsetzt, um fehlerfrei arbeiten zu können. Dem entsprechend wurde APGAS pro Knoten mit 12 Places, sprich einem Place für jeden CPU-Kern und wiederum 16 Threads pro Place konfiguriert. Bei der Berechnung des eigentlichen Experiments wird innerhalb des GLB-Frameworks ein verteilter Log bezüglich Lastenbalancierungs-Ereignissen erzeugt. Dieser Log wird zum Ende der Anwendung auf Place 0 zusammengeführt. Auf dem GLB-Framework für APGAS basierende Experimente liefern die Metriken:

- *Process Time*: Zeit die gebraucht wurde, um alle Teilergebnisse zu berechnen.
- *Result Reduction Time*: Zeit die benötigt wurde, um die zuvor berechneten Teilergebnisse zusammenzuführen. Das GLB-Framework führt diese Reduktion auf Place 0 durch. Es werden zunächst die Teilergebnisse von jedem Place nach Place 0 übertragen und dann lokal reduziert.
- *Log Reduction Time*: Zeit die das Experiment gebraucht hat Logs zur Lastenbalancierung auf Place 0 zusammenzufügen. Es werden zunächst alle Daten an Place 0 übertragen und dann zusammengefügt.

Analog zu dem GLB-Framework liefern die auf `asyncAny` basierenden Benchmarks die Metriken *Process Time* und *Result Reduction Time*. Der zur Reduktion verwendete Algorithmus unterscheidet sich jedoch von dem des GLB-Frameworks. Daten, die während der Reduktion an Place 0 übertragen wurden, werden sofort mit dem lokalen Datensatz zusammengefügt. Experimente, die `asyncAny` Lastenbalacierung einsetzen wurden mit einem Place und 12 Threads pro Knoten, sprich einem Thread pro CPU-Kern ausgeführt.

Um Varianz in den Ergebnissen auszugleichen, wurden alle Experimente fünf Mal wiederholt. Im Folgenden dargestellte Metriken sind das Mittel aus drei Durchläufen. Um den Einfluss von Ausreißern in der Messung zu minimieren, fließen minimal beziehungsweise maximal gemessene Werte nicht in den endgültigen Wert ein.

5.2. Benchmarks und Kennzahlen

Betweenness Centrality

Das Betweenness Centrality (BC) Benchmark berechnet für jeden Knoten eines Graphen einen BC-Wert. Der BC-Wert eines Knotens entspricht der Anzahl der kürzesten Wege innerhalb des Graphen, die den Knoten beinhalten. Die so bestimmten Werte dienen als Maß für die Zentralität der einzelnen Knoten.

Für die Experimente wurden Benchmark Implementierungen von Posner [4, 43] für APGAS, basierend auf Bader et al. [44], verwendet. In allen Experimenten wurde das Benchmark mit $N = 2^{19}$ Knoten und initialem Seed $s = 2$ konfiguriert.

Zu Beginn werden die Knoten des Graphen gleichmäßig allen Places zugeteilt. Jeder Place arbeitet unabhängig seine Knotenmenge ab. Kommunikation findet nur zur Lastenbalancierung und zum Zusammenführen der Ergebnisse statt, wobei jedes Einzelergebnis aus N 64-Bit Werten besteht.

Unbalanced Tree Search

Unbalanced Tree Search (UTS) wurde 2016 von Olivier et al. [45] als Benchmark vorgestellt. UTS beschreibt einen Baum anhand der Parameter Verzweigungsfaktor, Baumtiefe und einem Seed für Zufallswerte. Das Benchmark berechnet die Anzahl der Knoten des Baumes. Um Zufallszahlen zu erzeugen wird ein deterministisches Verfahren verwendet, um bei gleicher Konfiguration immer den gleichen Baum zu erhalten. Es ist nicht möglich von der Konfiguration auf die Anzahl der Knoten zu schließen. Folglich ist der gesamte Baum zu traversieren. Ob und wie viele Kinder ein Knoten hat, hängt nur von den Konfigurationsparametern des Baumes ab. Diese Eigenschaft erlaubt es jeden Knoten unabhängig zu betrachten. Jeder Task in UTS berechnet eine Zwischensumme für eine Gruppe von Knoten. Für jeden Knoten werden die Anzahl von Kindknoten bestimmt und Tasks für diese Kindknoten zur Bearbeitung eingereiht. Am Ende eines Tasks wird die Anzahl bearbeiteter Knoten zu einem Zwischenergebnis addiert. Zwischenergebnisse der Tasks werden pro Thread aggregiert. Alle Zwischenergebnisse werden zum Ende der Berechnung mittels Reduktion zu einer Gesamtsumme zusammengefügt. Ziel des Benchmarks ist es, über die Laufzeit eine ungleiche Lastenverteilung herbeizuführen, die es auszugleichen gilt. Für die Experimente wurde ein Fork der UTS Implementierung von Posner [4] für APGAS verwendet. Diese ist online unter [43] verfügbar. UTS wurde mit Verzweigungsfaktor b = 4, Baumtiefe d = 17 und Seed = 19 konfiguriert.

Bandbreite

Das Experiment zur Vermessung der Bandbreite BW ermittelt die Zeit T, die gebraucht wird, um ein Array der Größe N X-mal von Place 0 nach Place 1 zu übertragen. Aus der Zeit T wird daraufhin die Bandbreite ($bw = \frac{X*N}{T}$) bestimmt. Die Konstrukte `asyncAt` und `finish` wurden verwendet, um das gewünschte Verhalten zu realisieren. Um kalten Cache, Einflüsse durch den Java JIT Compiler und verzögerter Initialisierungen innerhalb von MPI zu vermeiden wird zu Beginn ein Durchlauf, dessen Ergebnisse verworfen werden, durchgeführt. Für Nachrichtengrößen im Bereich kleiner 2^{14} Bytes wird $X = 50000$ verwendet. Für größere N wird $X = 5000$ gesetzt. Dies ist notwendig um das gesamte Nachrichtenvolumen in sinnvollen Größen zu halten und die Laufzeit des Experiments nicht unnötig zu verlängern. Zusätzlich zu den Durchläufen von beiden Netzwerkschichten mit Ethernet und Infiniband wurde das Experiment mit veränderter Standardgröße des Serialisierungs-Puffers durchgeführt. Grund hierfür ist der starke Einfluss den diese Einstellung auf die Performance der MPI Netzwerkschicht ausübt.

Latenz

Das LAT Experiment ist Analog zu BW konzipiert. Um die Latenz ($lat = \frac{T}{2*X}$) zu bestimmen wird die Zeit T ermittelt, die es braucht ein Array der Größe N X-mal von Place 0 nach Place 1 und zurück zu übertragen. Die in diesem Experiment ermittelten Zeiten T sind systematisch durch Effekte wie Pipelining beeinflusst. Pipelining meint hier, dass die Netzwerkschicht mit der Übertragung der zweiten Nachricht von Place 0 nach Place 1 schon begonnen haben kann, obwohl vorherige Nachrichten noch nicht vollständig abgearbeitet sind. Als Konsequenz sind die errechneten Latenzen nur proportional zu den tatsächlichen Latenzen. Für einen Vergleich der Netzwerkschichten reicht dies jedoch aus. Wie auch zur Bestimmung der Bandbreite wurde auf die Konstrukte `asyncAt` und `finish` zurückgegriffen, um das Experiment zu implementieren. Zu Beginn des Experiments wird ebenfalls ein Leerlauf aus den in der Beschreibung des Experiments BW genannten Gründen durchgeführt. Für Nachrichtengrößen im Bereich kleiner 2^{14} Bytes wird $X = 50000$ verwendet, andernfalls ist $X = 5000$. Wie auch im BW Experiment wurden zusätzliche Durchläufe mit veränderter Standardgröße des Serialisierungs-Puffers durchgeführt. Analog aufgebaute Benchmarks sind unter anderem unter den OSU Micro-Benchmarks [46] oder Intels MPI Benchmarks [47] zu finden.

5.3. Ergebnisse

Im Folgenden werden die Ergebnisse der durchgeführten Experimente vorgestellt. Für die Benchmarks BC und UTS werden die für die zuvor beschriebenen Metriken ermittelten Werte dargestellt und interpretiert. Analog folgt eine Beschreibung der Ergebnisse von BW und LAT.

Betweenness Centrality

Global Load Balancing

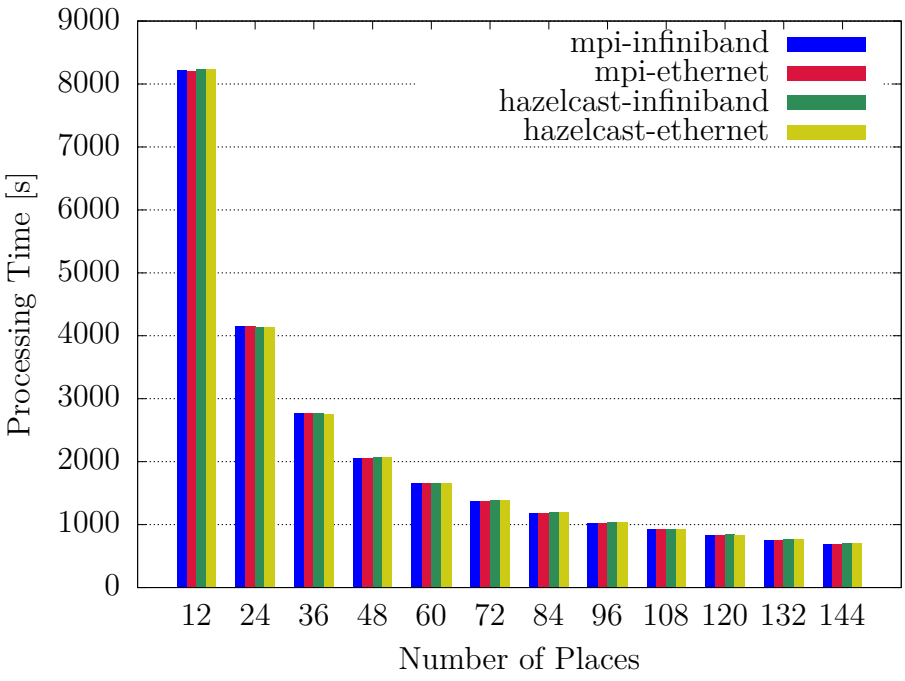

Abbildung 5.1.: Ergebnisse für Experiment Betweenness Centrality - GLB - Process Time

In den Abbildungen 5.1, 5.2 und 5.3 sind die für das Experiment Betweenness Centrality mit GLB Lastenbalancierung ermittelten Laufzeiten aufgetragen. Die gemessene *Process Time* der MPI Netzwerkschicht ist in fast allen Fällen marginal kürzer. Die Metriken *Result Reduction Time* und *Log Reduction Time* zeigen hier jedoch kein klares Bild. Die Hazelcast Netzwerkschicht zeigt bei der Reduktion der BC Ergebnisse ab einer größeren Anzahl von Places bessere Laufzeiten. Für die *Log Reduction Time* wurden kürzere Laufzeiten für die MPI Netzwerkschicht, falls

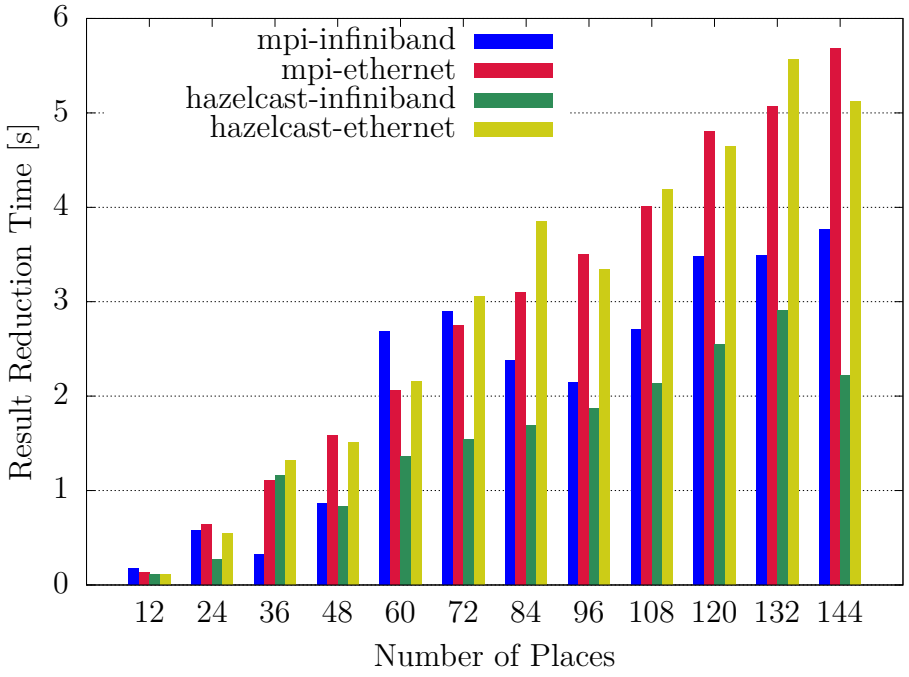

Abbildung 5.2.: Ergebnisse für Experiment Betweenness Centrality - GLB - Result Reduction Time

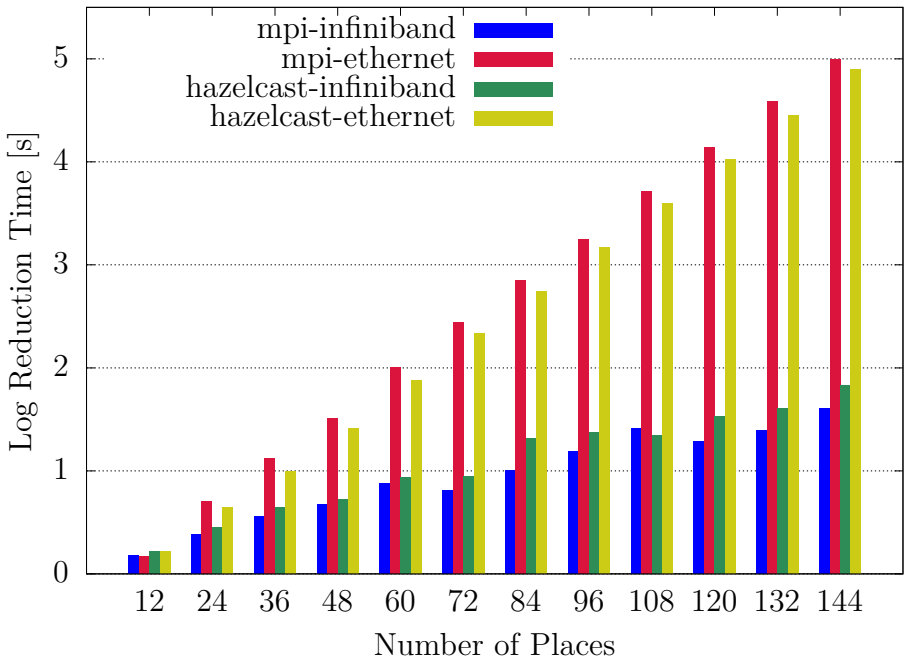

Abbildung 5.3.: Ergebnisse für Experiment Betweenness Centrality - GLB - Log Reduction Time

über Infiniband kommuniziert wird, festgestellt. Unter Verwendung von Ethernet performt jedoch die Hazelcast Implementierung besser.

Die in Abbildung 5.1 aufgetragene *Process Time* Metrik zeigt Abweichungen zwischen den Netzwerkschichten von bis zu 1.9%. Wenn auch geringfügig, so ist die MPI Netzwerkschicht bei allen getesteten Place Konfigurationen außer 36 Places (3 Knoten) schneller. Dies trifft sowohl auf die Experimente mit Ethernet, als auch auf die Durchläufe mit Infiniband zu. Festzustellen ist ebenfalls, dass mit steigender Anzahl Places die MPI Netzwerkschicht zunehmend besser performt. Die fast identischen *Process Time* Werte liegen im hauptsächlich geringen Kommunikationsvolumen während der Berechnung der BC-Werte begründet.

In Abbildung 5.2 sind die ermittelten Werte für *Result Reduction Time* dargestellt. Beim Zusammenführen der Ergebnisse wurde pro Knoten 8 Byte $* 2^{19} =$ 4194304 Byte $\approx$ 4MB übertragen. Es zeichnet sich ab, dass Hazelcast vor allem beim Einsatz von Infiniband mit einer größeren Zahl von Knoten bessere Laufzeiten erzielt. Ein Grund ist die Implementierung von Serialisierung innerhalb der MPI Netzwerkschicht. Wie in Abschnitt 4 dargelegt, wird ein Puffer konfigurierbarer Größe verwendet, um die Byterepräsentation einer Nachricht im Speicher zu halten. Dieser Puffer reicht nicht aus, um ein 4MB Array zu serialisieren. Als Konsequenz müssen Daten kopiert und der Puffer neu alloziert werden. Zusätzlich wirkt sich die Notwendigkeit der vollständigen Serialisierung, bevor mit der Übertragung begonnen werden kann, negativ aus. Der Einfluss von Nachrichtengröße auf Bandbreite und Latenz wurde mit den Experimenten BW und LAT (Abschnitt 5.2) näher betrachtet.

Abbildung 5.3 zeigt eine kürzere *Log Reduction Time* für die Kombination MPI Netzwerkschicht mit Infiniband, während die Hazelcast Ethernet Durchläufe im Vergleich bessere Werte aufzeigen. Im Gegenzug zu *Result Reduction Time* passt die Größe eines Log-Eintrags mit ungefähr $2 * 150 = 300$ Byte an Daten in den Puffer. Ebenfalls zu bemerken ist, dass in diesem Fall beide Netzwerkschichten annähernd gleich gut skalieren.

AsyncAny

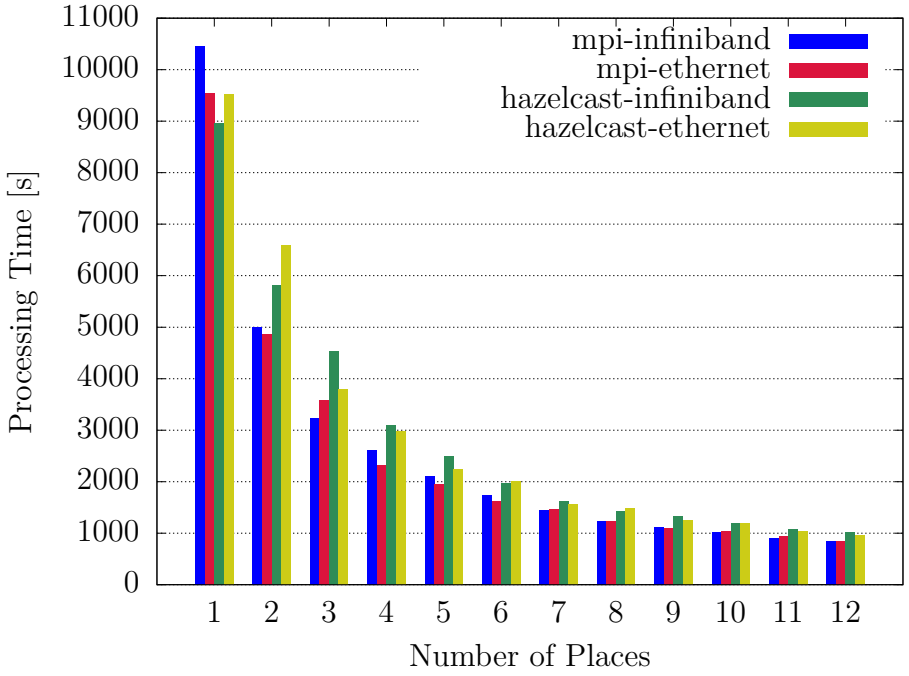

Abbildung 5.4.: Ergebnisse für Experiment Betweenness Centrality - AsyncAny - Process Time

Die Abbildungen 5.4 und 5.5 stellen die für das Experiment Betweenness Centrality mit `AsyncAny` Lastenbalancierung ermittelten Laufzeiten für die respektiven Metriken *Process Time* und *Result Reduction Time* dar. Die *Process Time* der MPI Netzwerkschicht ist, außer in den Durchläufen mit nur einem Place, merklich kürzer als die Hazelcast Variante. Die gemessene *Result Reduction Time* verhält sich hingegen konträr. Hier performt die MPI Netzwerkschicht tendenziell schlechter als Hazelcast.

Die *Process Time* der MPI Netzwerkschicht, aufgetragen in Abbildung 5.4, ist im Vergleich für Konfigurationen mit mehr als einem Place bis zu 40% schneller. Mit zunehmender Anzahl von Places nimmt dieser Unterschied zunächst ab. Bei 7 Places ist MPI mit Infiniband 12.4% schneller. Die weitere Tendenz ist jedoch, dass ab diesem Punkt die Laufzeiten wieder stärker divergieren, bis bei 12 Places MPI ungefähr 20% (Infiniband) bzw. ungefähr 13% (Ethernet) schneller ist.

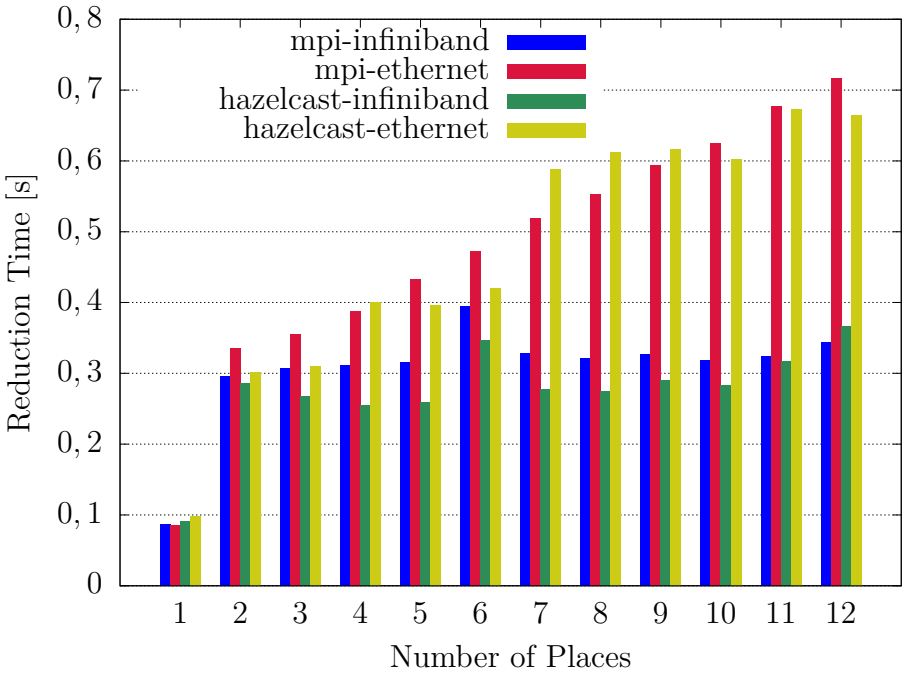

Abbildung 5.5.: Ergebnisse für Experiment Betweenness Centrality - AsyncAny - Result Reduction Time

Die *Result Reduction Time* in Abbildung 5.5 verhält sich analog zu den mit GLB Lastenbalancierung ermittelten Ergebnissen. Auch hier ist die Serialisierung der 4MB Daten pro Place ein Problem für die MPI Netzwerkschicht. Im Unterschied zu der GLB Lastenbalancierung entsprechen die Ergebnisse jedoch grundlegenden Erwartungen, wie, dass Infiniband besser als Ethernet performt.

Unbalanced Tree Search

Global Load Balancing

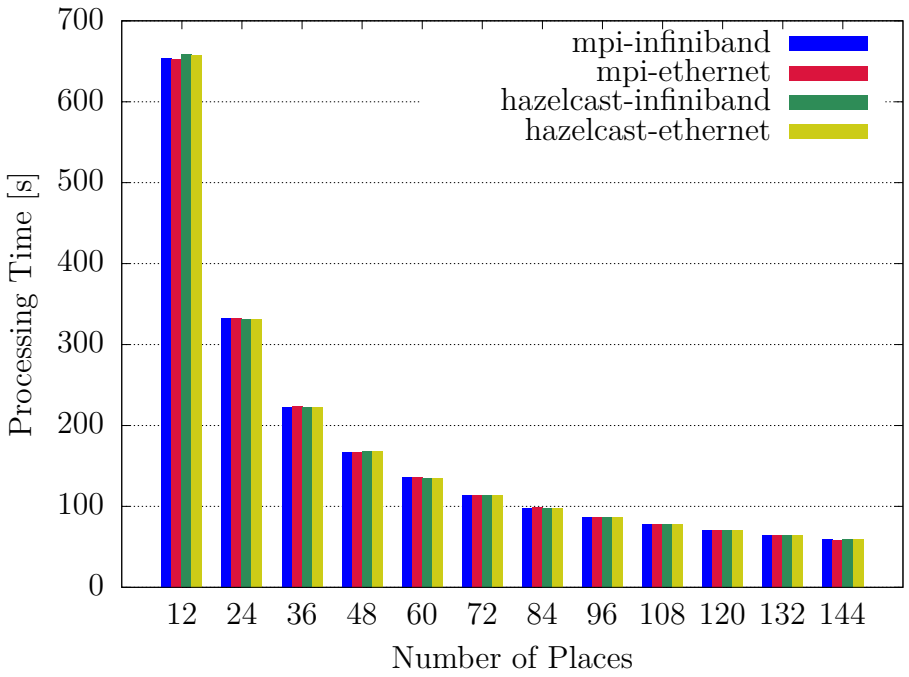

Abbildung 5.6.: Ergebnisse für Experiment Unbalanced Tree Search - GLB - Process Time

Die Abbildungen 5.6, 5.7 und 5.8 zeigen die ermittelten Laufzeiten für Unbalanced Tree Search mit GLB Lastenbalancierung. Die ermittelte *Process Time* ist für beide Netzwerkschichten effektiv gleich, während mit der Hazelcast Netzwerkschicht sowohl für *Result Reduction Time* als auch für *Log Reduction Time* tendenziell kürzere Zeiten ermittelt wurden. Bei beiden Metriken entsteht, im Gegensatz zu dem BC - GLB Experiment, kein klares Bild. So sind zum Beispiel einige Durchläufe, unabhängig von der Netzwerkschicht, mit Ethernet schneller oder deutlich langsamer als mit Infiniband.

Die in Abbildung 5.6 aufgetragene *Process Time* zeigt eine Abweichung von durchschnittlich 0.2%. Entsprechend ist kein signifikanter Unterschied zwischen den Netzwerkschichten festzustellen.

Bei Betrachtung der *Result Reduction Time* in Abbildung 5.7 ist vor allem die wesentlich schlechtere Performance von Ethernet bei zunehmender Anzahl Places festzustellen. So verschlechtert sich die *Result Reduction Time* bei Einsatz der Hazelcast Netzwerkschicht von 36 auf 48 Places um Faktor ≈ 3. Im Fall von MPI ist

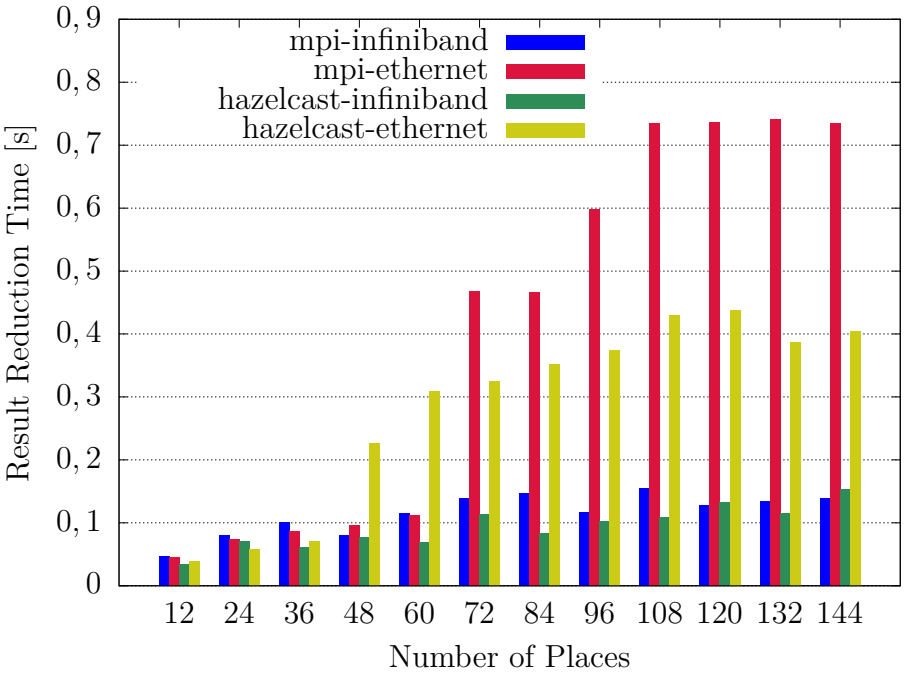

Abbildung 5.7.: Ergebnisse für Experiment Unbalanced Tree Search - GLB - Result Reduction Time

ähnliches Verhalten beim Übergang von 60 auf 72 Places mit einem Faktor von ≈ 4 festzustellen. Insgesamt braucht die MPI Netzwerkschicht in den meisten Fällen länger. Lässt man Ethernet jedoch außen vor, so befinden sich die gemessenen Laufzeiten beider Netzwerkschichten in der gleichen Größenordnung.

Auch bezüglich der in Abbildung 5.8 dargestellten *Log Reduction Time* ist die Tendenz zu Gunsten der Hazelcast Netzwerkschicht. Generell scheint sich jedoch kein klares Bild abzuzeichnen. So scheint jede mögliche Kombination zwischen Netzwerkschicht und Technologie mal die schnellste Option.

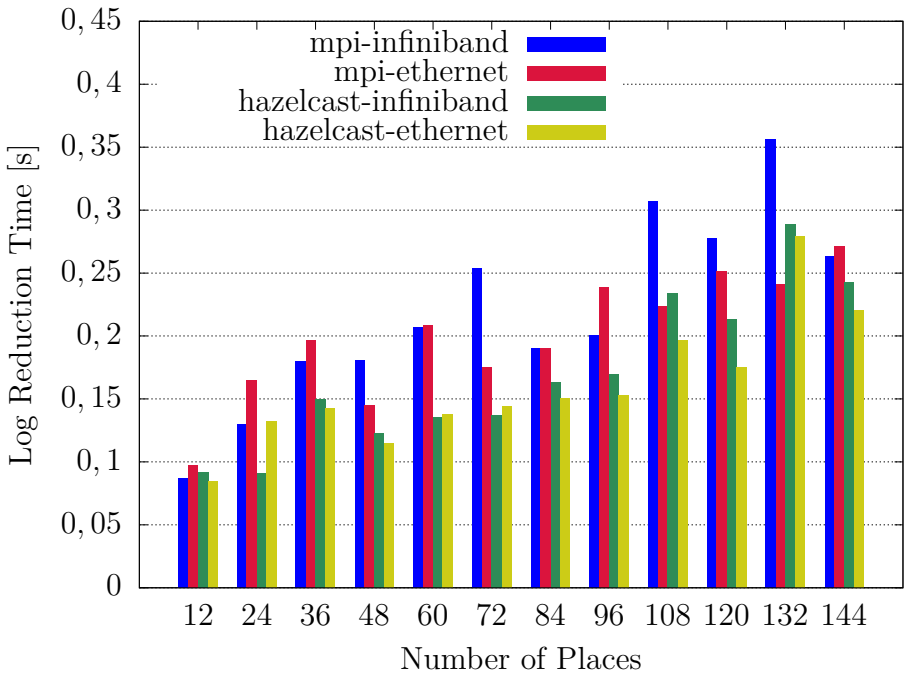

Abbildung 5.8.: Ergebnisse für Experiment Unbalanced Tree Search - GLB - Log Reduction Time

AsyncAny

In den beiden Abbildungen 5.9 und 5.10 sind die für Unbalanced Tree Search mit AsyncAny Lastenbalancierung ermittelten Laufzeiten aufgetragen. Während bei Betrachtung der *Process Time* Metrik die MPI Netzwerkschicht in fast allen Fällen merklich kürzere Laufzeiten ermöglicht, ist die *Reduction Time* weitestgehend merklich langsamer.

Abbildung 5.9 beziehungsweise die dargestellte *Process Time* zeigt vor allem bei einer kleinen Anzahl von Places einen Performancevorteil der MPI Netzwerkschicht. So braucht die Hazelcast Netzwerkschicht bei einem Place ≈ 1.2 mal so lange. Mit zunehmender Anzahl von Places nimmt dieser Unterschied jedoch ab.

Die *Reduction Time* in Abbildung 5.10 scheint durch den im Vergleich zu den Experimenten mit GLB Lastenbalancierung besseren Reduktionsalgorithmus fast konstant zu skalieren. Dies ist unabhängig von der eingesetzen Netzwerkschicht zu beobachten. Es ist festzustellen, dass MPI, außer im trivialen Fall von einem Place, durchschnittlich um einen Faktor von ≈ 2.2 langsamer ist.

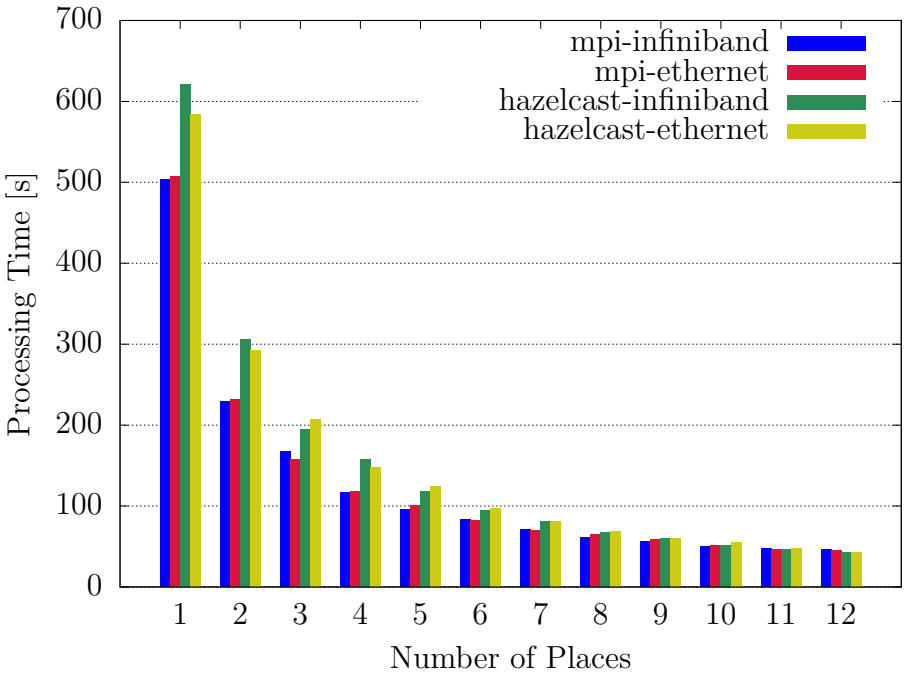

Abbildung 5.9.: Ergebnisse für Experiment Unbalanced Tree Search - AsyncAny - Process Time

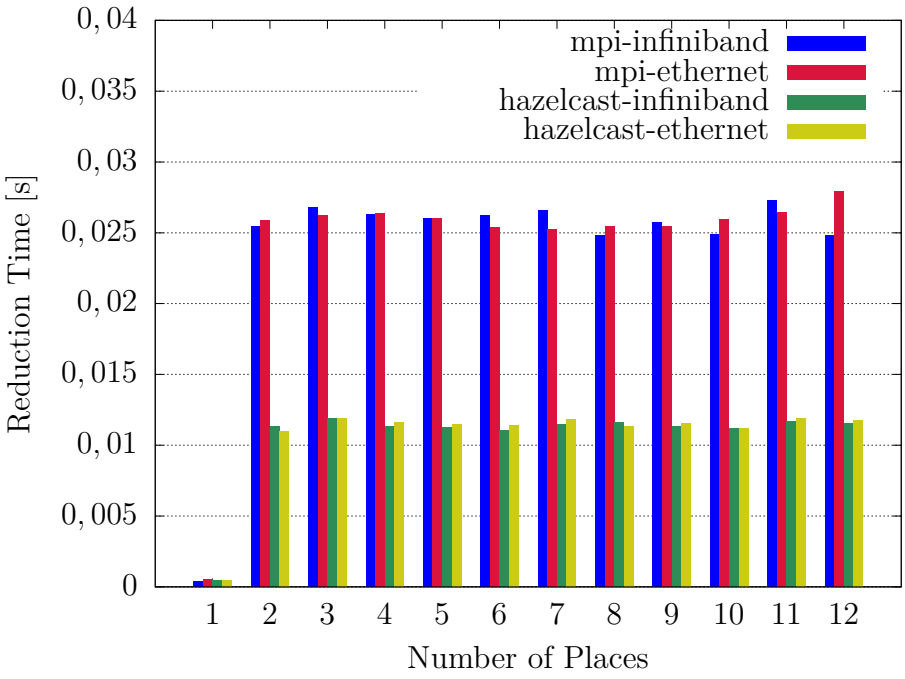

Abbildung 5.10.: Ergebnisse für Experiment Unbalanced Tree Search - AsyncAny - Reduction Time

Bandbreite

Die im Rahmen des Experiments BW erreichten Bandbreiten sind in den Abbildungen 5.11 und 5.12 in Relation zu der jeweiligen Nachrichtengröße aufgetragen. Im Gegensatz zu anderen Abbildungen bedeutet hier ein größerer Wert, dass mehr Daten pro Sekunde übertragen werden können und somit besser ist. Während Abbildung 5.11 Nachrichtengrößen bis zu $2^{13} = 8192$ Byte umfasst, sind in Abbildung 5.12 Nachrichtengrößen von $2^{13} = 8192$ bis zu $2^{22} = 4194304$ Byte ($\approx$ 4MB) dargestellt. Diese Aufteilung entspricht jeweils der Anzahl von Iterationen $X = 50000$ in Abbildung 5.11 und $X = 5000$ in Abbildung 5.12.

Bei näherer Betrachtung von Abbildung 5.11 fällt auf, dass die MPI Netzwerkschicht nicht in der Lage ist das Potential von Infiniband, bei der Verarbeitung vieler kleiner Nachrichten, auszuschöpfen. So erreicht die Hazelcast Netzwerkschicht eine Bandbreite von $\approx$ 265 MB/s bei einer Nachrichtengröße von 2^{13} Byte ($\approx$ 8kB), während für die MPI Netzwerkschicht $\approx$ 74 MB/s ermittelt wurden. Im Gegensatz dazu hat die Hazelcast Netzwerkschicht Probleme bei größerem Kommunikationsvolumen. Die MPI Netzwerkschicht erreicht bei 2^{16} pro Nachricht eine Bandbreite von $\approx$ 240 MB/s, die Hazelcast Netzwerkschicht hingegen ist mit $\approx$ 87 MB/s vermessen (Abbildung 5.12). Durch Anpassung der Standardgröße des Serialisierungs-Puffers lässt sich die MPI Netzwerkschicht anpassen, um bestimmte Nachrichtengrößen besser verarbeiten zu können. Aufgetragen unter mpi-16384 wird durch eine Verdoppelung der Standardgröße des Serialisierungs-Puffers eine Bandbreite von bis zu $\approx$ 315 MB/s erreicht. Entgegen der großen Unterschiede, die beim Einsatz von Infiniband aufgetreten sind, liegen die erreichten Bandbreiten für Ethernet meist im gleichen Bereich. Tendenziell performt im Fall von Ethernet MPI besser bei kleineren Nachrichtengrößen und Hazelcast bei Nachrichten im Bereich von 2^{15} bis 2^{17}. Bei einer Nachrichtengröße von 2^{22} Byte ($\approx$ 4MB) ist Hazelcast jedoch nicht in der Lage das Nachrichtenaufkommen über Ethernet zu verarbeiten und läuft in einen internen Fehler. Dieser Fehler lässt sich weder auf Anwenderseite noch durch APGAS handeln. Weiterhin ist das Auftreten des Fehlers nur durch ausbleibenden Fortschritt des Programms festzustellen. Hazelcast konnte der Fehler nur durch zusätzliches Logging innerhalb der Bibliothek zugeordnet werden.

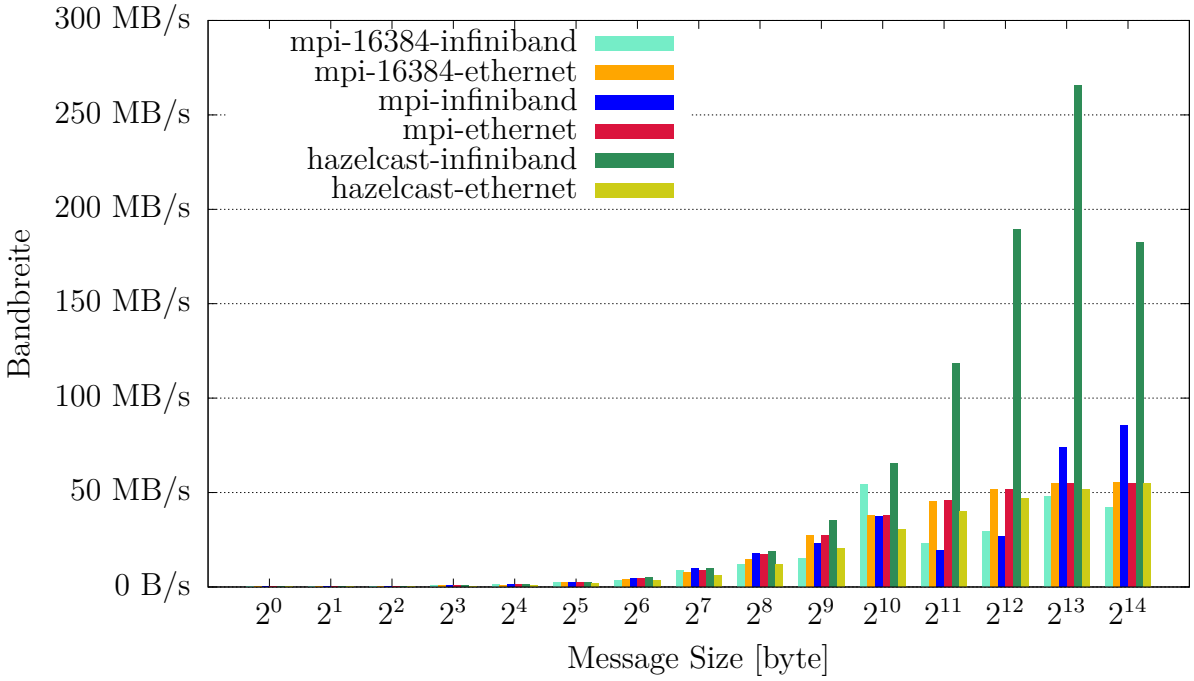

Abbildung 5.11.: Bandbreite für Nachrichtengrößen kleiner 2^{14} Byte (größer ist besser)

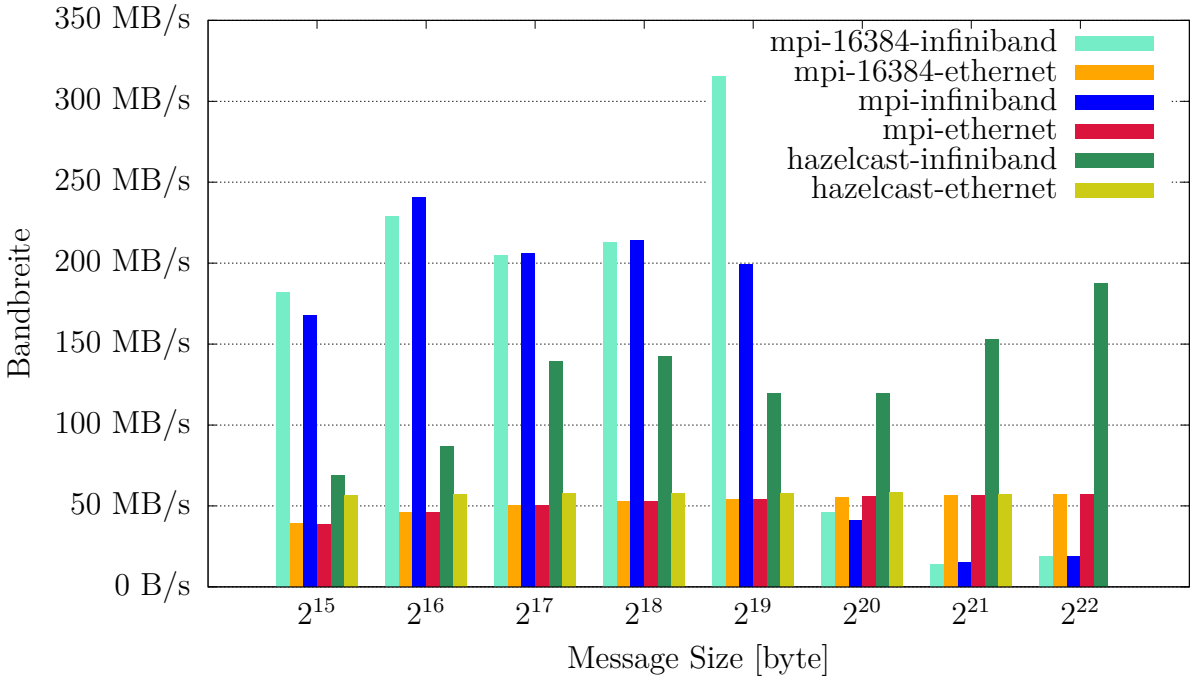

Abbildung 5.12.: Bandbreite für Nachrichtengrößen ab 2^{14} Byte (größer ist besser)

Latenz

Die Ergebnisse des Experiments LAT sind in Abbildung 5.13 und Abbildung 5.14 dargestellt. Die Aufteilung auf die Abbildungen ist analog zu BW gestaltet. So umfasst Abbildung 5.13 Nachrichtengrößen bist zu $2^{13} = 8192$ Byte, die mit in $X = 50000$ Iterationen übertragen wurden. Nachrichtengrößen bis einschließlich $2^{20} = 1048576 \approx 1\text{MB}$ sind in Abbildung 5.14 wieder zu finden.

Abbildung 5.13 zeigt deutlich, dass die Latenz für Datenmengen bis zu $2^8 = 256$ Byte pro Nachricht unabhängig von Netzwerkschicht und Technologie als effektiv konstant angesehen werden kann. Im Bereich bis zu einer Nachrichtengröße von $2^{12} = 8192$ Byte performt Hazelcast - Infiniband am Besten. Für die Interpretation der Ergebnisse relevante Kennzahlen sind die maximale Übertragungseinheit (MTU) die für Ethernet auf 1500 Byte ($\approx 2^{10}$) und Infiniband auf 2044 Byte ($\approx 2^{11}$) eingestellt waren. Die MTU bestimmt die maximale Paketgröße, mit der Nachrichten im IP-Protokoll übertragen werden können. Dieser Einschnitt zwischen Nachrichten kleiner beziehungsweise größer der MTU lässt sich in Abbildung 5.13 wiederfinden. So deckt sich der Bereich von Nachrichten kleiner der MTU mit dem zuvor beschriebenen Bereich effektiv konstanter Latenz. Wichtig zu bemerken ist, dass mit einer Größe von $\approx 2^{13}$ Byte beziehungsweise $\approx 2^{14}$ Nachrichten nicht mehr in den Standard Puffer für Serialisierung in der MPI Netzwerkschicht passen und so zusätzliche Kopien und Allozierungen notwendig werden. MPI - Infiniband ist bei Nachrichtengrößen kleiner 2^{12} mit einer größeren Latenz behaftet als Hazelcast - Infiniband und MPI - Ethernet. Für Nachrichtengrößen im Bereich zwischen 2^{13} und 2^{18} wurden für MPI - Infiniband jedoch die kürzeste Latenz ermittelt. Ab Nachrichtengrößen von $2^{19} \approx 0.5\text{MB}$ beziehungsweise 2^{20} im Fall von mpi-16384 liegt die Latenz der MPI Netzwerkschicht jedoch weit über allen anderen Alternativen. Analog zu BW lässt sich auch hier feststellen, dass durch Tuning der Standardgröße des Serialisierungs-Puffers die Latenz merklich zu Gunsten bestimmter Nachrichtengrößen beeinflussbar ist.

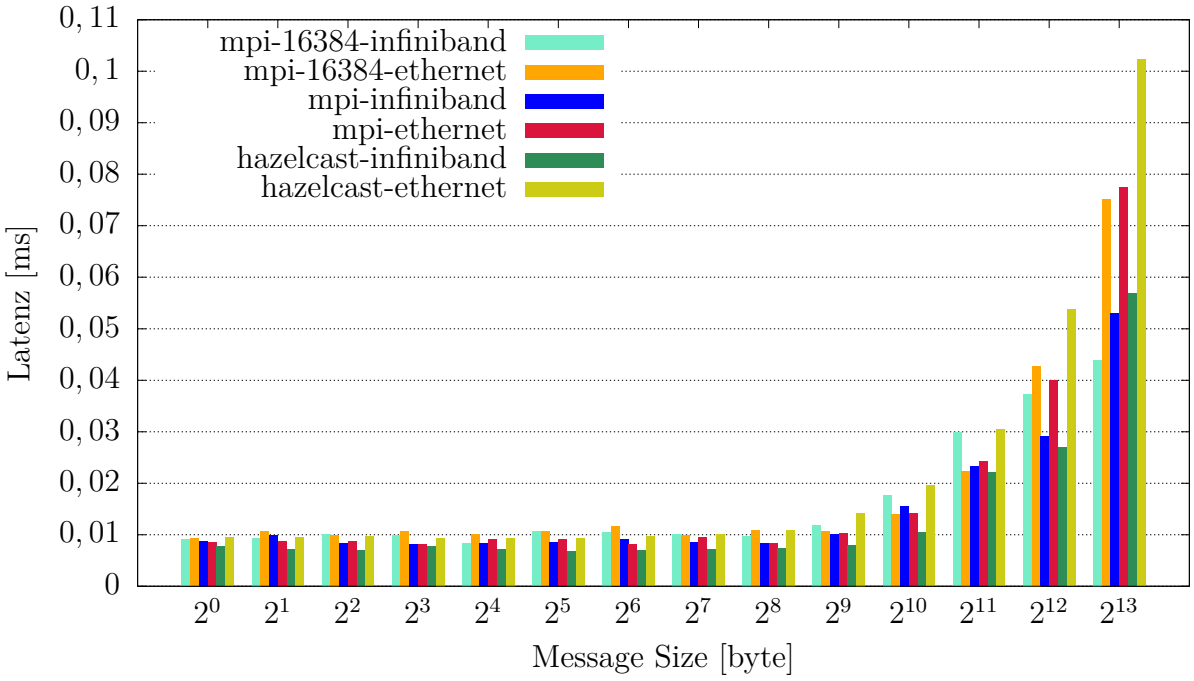

Abbildung 5.13.: Latenz für Nachrichtengrößen kleiner 2^{14} Byte

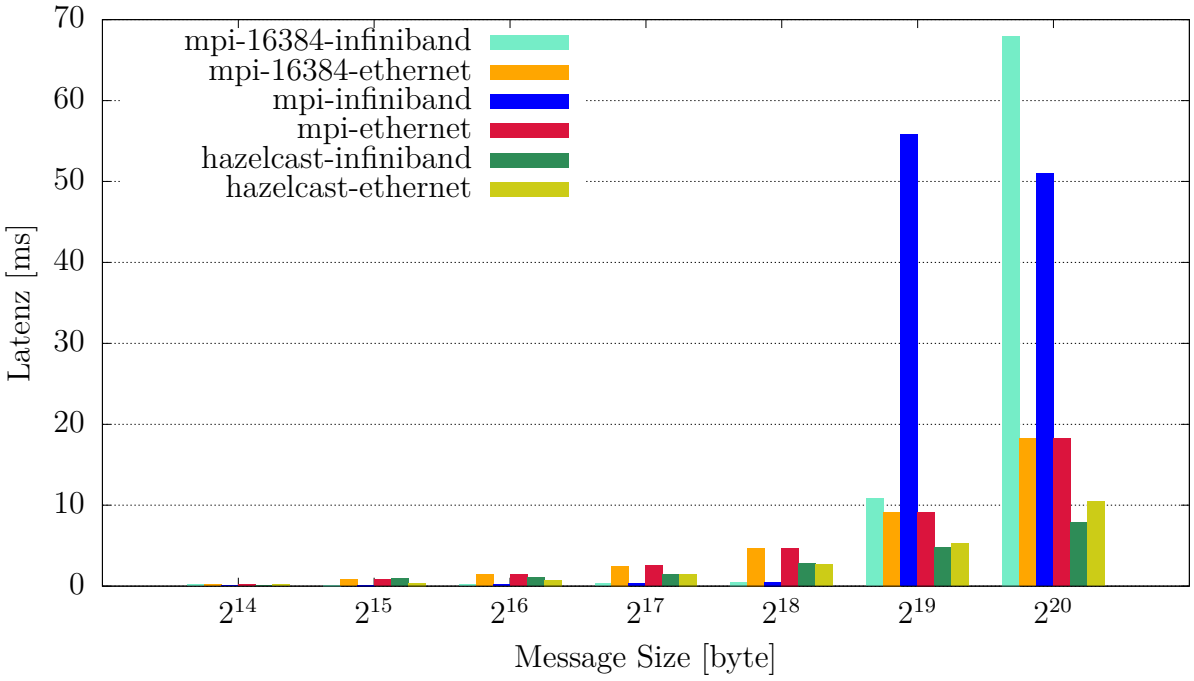

Abbildung 5.14.: Latenz für Nachrichtengrößen ab 2^{14} Byte

5.4. Diskussion

Betrachtet man die durchgeführten Benchmarks, so lässt sich keine allgemeingültige Aussage über die Performance der Netzwerkschichten treffen. So sind zum Beispiel die Laufzeiten der MPI Netzwerkschicht bei der verteilten Berechnung der Ergebnisse (*Process Time*) kürzer, während Hazelcast bei den Reduktionsaufgaben bessere Laufzeiten erreicht. In Tabelle 5.1 wird ein grober Überblick über die Ergebnisse aller durchgeführten Experimente geboten. Auch wenn die Reduktionen im Hinblick auf die gesamte Laufzeit der Benchmarks nicht wirklich ins Gewicht fallen, ist die Performance der MPI Netzwerkschicht hier unerwartet schlecht. Die ermittelten Laufzeiten bei dem Benchmark BC befinden sich im Bereich von 800 bis 10000 Sekunden für *Process Time*, während die Reduktion der Ergebnisse weniger als 6 Sekunden in Anspruch nahm. Bei dem UTS Benchmark bewegt sich die *Process Time* zwischen 40 und 700 Sekunden, während die Reduktionen weniger als eine Sekunde dauern.

Die im Rahmen der Benchmark Reduktionen ermittelten Laufzeiten ließen auf einen Zusammenhang zwischen Nachrichtengröße und den auffälligeren Zeiten schließen. Dieser Eindruck erhärtet sich durch die Ergebnisse der Experimente BW und LAT. Ebenfalls ist in beiden Experimenten ein deutlicher Einfluss der innerhalb der MPI Netzwerkschicht eingesetzten Java Serialisierung zu sehen. So wurde nach Anpassung der Standardparameter die MPI Netzwerkschicht eine Bandbreite von $\approx$ 315 MB/s möglich. Die Hazelcast Netzwerkschicht erreicht $\approx$ 265 MB/s, während mit der Standardkonfiguration $\approx$ 240 MB/s ermittelt wurden. Weiterhin ist festzuhalten, dass Hazelcast nicht in der Lage war das Nachrichtenvolumen des BW Experiments zu verarbeiten.

Ein weiterer problematischer Punkt seitens Hazelcast ist, dass von jedem Knoten aus zu jedem anderen Knoten Socket-Verbindungen geöffnet werden. Skaliert man also eine Anwendung auf eine große Knotenanzahl, so ist die Anzahl möglicher Prozesse durch die quadratisch wachsende Anzahl von Sockets begrenzt (für ein Standard Linux System liegt die Grenze $\leq$ 1024 Prozessen). MPI hingegen öffnet nur dann eine Verbindung, wenn auch darüber kommuniziert wird. Zusammenfassend ist festzuhalten, dass die MPI Netzwerkschicht bei den Experimenten besser oder zumindest vergleichbar zu der Hazelcasst Netzwerkschicht performt. Obwohl die Ergebnisse hinter den Erwartungen zurückblieben, ist das Potential einer MPI Netzwerkschicht noch nicht ausgeschöpft.

Netzwerkschicht		Hazelcast		MPI	
Netzwerktechnologie		Inifinband	Ethernet	Inifinband	Ethernet
GLB Lastenbalancierung					
BC	*Process Time*	=	=	=	=
BC	*Result Reduction Time*	+	–	–	+
BC	*Log Reduction Time*	–	+	+	–
BC	Gesamtlaufzeit	=	=	=	=
UTS	*Process Time*	=	=	=	=
UTS	*Result Reduction Time*	+	+	–	–
UTS	*Log Reduction Time*	++	+	– –	–
UTS	Gesamtlaufzeit	=	=	=	=
AsyncAny Lastenbalancierung					
BC	*Process Time*	–	–	+	+
BC	*Reduction Time*	+	=	–	=
BC	Gesamtlaufzeit	–	–	+	+
UTS	*Process Time*	–	–	+	+
UTS	*Reduction Time*	++	++	– –	– –
UTS	Gesamtlaufzeit	–	–	+	+
BW	$[2^0 - 2^8]$ Byte	=	–	=	+
BW	$[2^9 - 2^{14}]$ Byte	++	–	– –	+
BW	$[2^{15} - 2^{19}]$ Byte	– –	–	++	+
BW	$[2^{20} - 2^{22}]$ Byte	++	=	– –	=
LAT	$[2^0 - 2^8]$ Byte	=	=	=	=
LAT	$[2^9 - 2^{12}]$ Byte	+	–	–	+
LAT	$[2^{14} - 2^{18}]$ Byte	– –	+	++	–
LAT	$[2^{19} - 2^{20}]$ Byte	++	++	– –	– –

++ deutlich besser, + besser, = gleich, – schlechter, – – deutlich schlechter

Tabelle 5.1.: Überblick über die Ergebnisse der Experimente

6. Fazit und Ausblick

In dieser Arbeit wurde eine neue auf MPI basierende Netzwerkschicht für APGAS entwickelt. Um MPI in Java verwenden zu können wurden JNI-Bindings von Vega-Gisbert et al. [3] genutzt. Experimentell wurde die Performance der Hazelcast und MPI Netzwerkschichten gegenübergestellt. Es wurde ermittelt, dass trotz der Verwendung der Infiniband Hardware mittels IPoIB und JNI Overhead die MPI Netzwerkschicht bessere Performance erreichen kann. So wurden, betrachtet man die Gesamtlaufzeit der Benchmarks, in den meisten Fällen mit Hazelcast vergleichbare und teilweise kürzere Laufzeiten gemessen. Auch wenn es sich bei den Reduktionsanteilen um einen geringen Anteil an Gesamtlaufzeit handelt, bleibt festzuhalten, dass die MPI Netzwerkschicht hier meistens schlechter performt hat. Weiterhin wurde festgestellt, dass die MPI Netzwerkschicht robuster im Bezug auf größere Kommunikationsvolumen ist. So ist die auf Hazelcast basierende Netzwerkschicht weder in der Lage das BW Experiment über Ethernet mit $\approx$ 4MB Nachrichtengröße durchzuführen, noch wird die Möglichkeit geboten, die Anwendung im Fehlerfall sauber herunterzufahren. In Bezug auf die Komplexität für den Endanwender ist festzuhalten, dass die Verwendung von OpenMPI den Bauprozess und die Installation von APGAS verkompliziert. Weiterhin wurde zusätzlich entstandene Komplexität bei der Konfiguration und dem Starten von APGAS-Anwendungen durch das Skript `apgasrun.sh` mitigiert. Alle Features von APGAS sind weiterhin in vollem Umfang bei Verwendung der auf Hazelcast basierenden Netzwerkschicht verfügbar. Einzig Resilienz und Kryo Serialisierung sind nicht mit der MPI Netzwerkschicht kompatibel.

Über die in dieser Arbeit entwickelte Implementierung der MPI Netzwerkschicht hinaus verspricht MPI Potential für Performanceverbesserungen. In zukünftigen Arbeiten könnten kollektive Operationen häufige Kommunikationsmuster wie Broadcast, Scatter, Gather oder auch Reduktion sowohl intern in der Laufzeitumgebung, als auch über neue Konstrukte auf Anwendungsseite verfügbar gemacht werden. Einseitige Kommunikation bzw. MPIs Remote Memory Access (RMA) Konstrukte bieten sich an, um die Implementierung des Speichermodells zu optimieren. Nicht standardisierte Ansätze für Resilienz von MPI Anwendungen, wie User Level Fault Mitigation (ULFM) [48] könnten evaluiert und verwendet werden, um das Feature-Set beider Netzwerkschichten anzugleichen. Darüber hinaus wäre es interessant die Performance des MPI-Konstrukts `MPI_Mprobe` zu untersuchen. Dieses Konstrukt

scheint es, im Gegensatz zu dem verwendeten `MPI_Iprobe`, zu erlauben mit mehreren Threads gleichzeitig Nachrichten dynamischer Größe zu empfangen. Weiterhin bietet die Serialisierung von Tasks in APGAS Optimierungspotential. Die zur Serialisierung verwendeten Java Mechanismen sind problematisch und sollten angepasst beziehungsweise ersetzt werden. In aktueller Form wird jede Nachricht in einen zuvor allozierten Puffer in einem Durchgang serialisiert. Ob die Puffergröße ausreicht kann nur während des Prozesses festgestellt werden. Dies kann vor allem bei großen Nachrichten zu erneuten Allozierungen und Kopien führen. Weiterhin lässt der Prozess das Übertragen der Nachricht erst zu nachdem die komplette Serialisierung abgeschlossen ist. Hier ist es denkbar, nach Anpassung der Serialisierung, größere Nachrichten in Teilen zu übertragen. Neben den Problemen der Serialisierung selbst, müssen Tasks so oft serialisiert werden, wie sie ausgeführt werden sollen. Ein möglicher Ansatz wäre ein separates Konstrukt um, wenn gewünscht, die Serialisierung einmal vorab durchführen zu können. Sowohl die vorgeschlagene Anpassung bezüglich Serialisierung, als auch der Einsatz der entsprechenden MPI Konstrukte scheinen passend, um die eher schlechte Performance der neuen Netzwerkschicht bei Reduktion verbessern zu können.

Literatur

[1] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky und O. Tardieu. «The asynchronous partitioned global address space model». In: *1st ACM SIGPLAN Workshop on Advances in Message Passing (AMP'10)*. ACM Press, 2010.

[2] Hazelcast Inc. *The Leading Open Source In-Memory Data Grid*. URL: `https://hazelcast.com` (aufgerufen 2020).

[3] O. Vega-Gisbert, J. E. Roman und J. M. Squyres. «Design and implementation of Java bindings in Open MPI». In: *Parallel Computing* 59 (2016), S. 1–20.

[4] J. Posner. «Global load balancing and intra-node synchronization with the Java framework APGAS». Magisterarb. Master Thesis, Department for Electric Engineering/Computer Science, Universität Kassel, 2016.

[5] IBM. *X10: Performance and Productivity at Scale*. URL: `http://x10-lang.org/` (aufgerufen 2020).

[6] O. Tardieu. «The APGAS Library: Resilient Parallel and Distributed Programming in Java 8». In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. New York, NY, USA: Association for Computing Machinery, 2015, 25–26. DOI: `10.1145/2771774.2771780`. URL: `https://doi.org/10.1145/2771774.2771780`.

[7] O. Tardieu. *The APGAS Library*. 2015. URL: `http://x10.sourceforge.net/documentation/papers/X10Workshop2015/slides/tardieu.pdf`.

[8] J. Posner. *Extended APGAS library repository*. commit [2bb9883ddebd89d3e46406b5043e00b34e71468a]. 2018. URL: `https://github.com/posnerj/PLM-APGAS` (aufgerufen 2020).

[9] J. Posner und C. Fohry. «Cooperation vs. coordination for lifeline-based global load balancing in APGAS». In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. 2016, S. 13–17.

[10] J. Posner und C. Fohry. «Fault Tolerance for Cooperative Lifeline-Based Global Load Balancing in Java with APGAS and Hazelcast». In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, S. 854–863.

[11] J. Posner und C. Fohry. «A Combination of Intra- and Inter-place Work Stealing for the APGAS Library». In: 2018, S. 234–243. doi: 10.1007/978-3-319-78054-2_22.

[12] J. Posner und C. Fohry. «Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library». In: *The Journal of Supercomputing* 74.4 (2018), S. 1435–1448.

[13] Oracle. *ForkJoinPool*. URL: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ForkJoinPool.html (aufgerufen 2020).

[14] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić und V. Sarkar. «HabaneroUPC++ a Compiler-free PGAS Library». In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 2014, S. 1–10.

[15] Esoteric Software LLC. *Kryo: Java binary serialization and cloning*. URL: https://github.com/EsotericSoftware/kryo (aufgerufen 2020).

[16] Oracle. *ByteBuffer*. URL: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ConcurrentMap.html (aufgerufen 2020).

[17] Hazelcast Inc. *Hazelcast IMDG Reference Manual*. URL: https://docs.hazelcast.org/docs/latest/manual/html-single/#what-is-hazelcast-imdg (aufgerufen 2020).

[18] Oracle. *Executor*. URL: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Executor.html (aufgerufen 2020).

[19] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz und G. R. Vallee. «A survey of MPI usage in the US exascale computing project». In: *Concurrency and Computation: Practice and Experience* 32.3 (2020).

[20] Ignacio Laguna. *A Simple and Efficient Fault-Tolerance Model for MPI Applications*. 2016. URL: http://mug.mvapich.cse.ohio-state.edu/static/media/mug/presentations/2016/MUG-Reinit-talk-small.pdf (aufgerufen 2020).

[21] Argonne National Laboratory. *MPICH | High-Performance Portable MPI*. URL: https://www.mpich.org/ (aufgerufen 2020).

[22] The Open MPI Project. *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (aufgerufen 2020).

[23] Microsoft Corporation. *Microsoft-MPI*. URL: https://github.com/Microsoft/Microsoft-MPI (aufgerufen 2020).

[24] Intel Corporation. *Intel MPI Library*. URL: https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html (aufgerufen 2020).

[25] E. Gabriel et al. «Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation». In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 2004, S. 97–104.

[26] The Open MPI Project. *Developer-level technical information on the internals of Open MPI*. URL: `https://www.open-mpi.org/faq/?category=developers` (aufgerufen 2020).

[27] The Open MPI Project. *Modular Component Architecture*. 2006. URL: `https://www.open-mpi.org/papers/workshop-2006/mon_06_mca_part_1.pdf` (aufgerufen 2020).

[28] The Open MPI Project. *Open MPI point to point architecture*. 2006. URL: `https://www.open-mpi.org/papers/workshop-2006/wed_01_pt2pt.pdf` (aufgerufen 2020).

[29] D. Bonachea und P. Hargrove. *GASNet Specification, v1.8.1*. 2017. DOI: `10.2172/1398512`.

[30] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks und K. Warren. *Introduction to UPC and language specification*. Techn. Ber. CCS-TR-99-157. IDA Center for Computing Sciences, 1999.

[31] D. Callahan, B. L. Chamberlain und H. P. Zima. «The cascade high productivity language». In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE. 2004, S. 52–60.

[32] M. Bauer, S. Treichler, E. Slaughter und A. Aiken. «Legion: Expressing locality and independence with logical regions». In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, S. 1–11.

[33] GASNet. *GASNet Communication System*. URL: `https://gasnet.lbl.gov/` (aufgerufen 2020).

[34] Bryan Carpen. *MPI-based Approaches for Java*. URL: `https://www.open-mpi.org/papers/mpi-java-presentation/mpi-java1995.pdf` (aufgerufen 2020).

[35] N. Mohamed, J. Al-Jaroodi, H. Jiang und D. Swanson. «JOPI: a Java object-passing interface». In: *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. 2002, S. 37–45.

[36] B. Carpenter, G. Fox, S.-H. Ko und S. Lim. *mpiJava 1.2: API specification*.

[37] B. Carpenter, V. Getov, G. Judd, A. Skjellum und G. Fox. «MPJ: MPI-like message passing for Java». In: *Concurrency: Practice and Experience* 12.11 (2000), S. 1019–1038.

[38] S. Nothaas, F. Ruhland und M. Schöttner. «A Benchmark to Evaluate InfiniBand Solutions for Java Applications». In: *arXiv preprint arXiv:1910.02245* (2019).

[39] Oracle. *ByteBuffer*. URL: `https://docs.oracle.com/javase/10/docs/api/java/nio/ByteBuffer.html` (aufgerufen 2020).

[40] SchedMD LLC. *Slurm Workload Manager*. URL: `https://slurm.schedmd.com` (aufgerufen 2020).

[41] V. K. J. Chu. *Transmission of IP over InfiniBand (IPoIB)*. RFC 4391. Internet Engineering Task Force, 2006. URL: `https://ietf.org/rfc/rfc4391.txt`.

[42] M. Wasi-ur Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang und D. K. D. Panda. «High-performance RDMA-based design of Hadoop MapReduce over InfiniBand». In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, S. 1908–1917.

[43] J. Posner. *Extended APGAS library example repository*. commit [7f9f86c20bf18ab7e32e0621e0936936a012532c]. 2018. URL: `https://github.com/posnerj/PLM-APGAS-Applications` (aufgerufen 2020).

[44] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann und T. Meuse. *HPCS Scalable Synthetic Compact Applications 2: Graph Analysis*. 2006. URL: `http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.0.pdf` (aufgerufen 2020).

[45] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan und C.-W. Tseng. «UTS: An Unbalanced Tree Search Benchmark». In: *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*. LCPC'06. Berlin, Heidelberg: Springer-Verlag, 2006, 235–250.

[46] Network-Based Computing Laboratory. *OSU MPI Latency Test*. URL: `https://github.com/forresti/osu-micro-benchmarks/blob/master/mpi/pt2pt/osu_latency.c` (aufgerufen 2020).

[47] Intel Corporation. *Intel MPI Benchmarks*. URL: `https://github.com/intel/mpi-benchmarks/` (aufgerufen 2020).

[48] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca und J. J. Dongarra. «An evaluation of user-level failure mitigation support in MPI». In: *European MPI Users' Group Meeting*. Springer. 2012, S. 193–203.

Anhang

A. OpenMPI Build Configuration

```
          Configured by: ********
          Configured on: ********
         Configure host: ********
 Configure command line: '--enable-mpi-java'
                         '--disable-mpi-fortran'
                         '--prefix=********'
                         '--with-ucx=no'
                         '--with-jdk-bindir=*/jdk-12.0.1/bin'
                         '--with-jdk-headers=*/jdk-12.0.1/include'
               Built by: ********
               Built on: ********
             Built host: ********
             C bindings: yes
           C++ bindings: no
            Fort mpif.h: no
           Fort use mpi: no
      Fort use mpi size: deprecated-ompi-info-value
       Fort use mpi_f08: no
Fort mpi_f08 compliance: The mpi_f08 module was not built
 Fort mpi_f08 subarrays: no
          Java bindings: yes
 Wrapper compiler rpath: runpath
             C compiler: gcc
    C compiler absolute: */gcc/7.1.0/bin/gcc
 C compiler family name: GNU
     C compiler version: 7.1.0
            C char size: 1
            C bool size: 1
           C short size: 2
             C int size: 4
            C long size: 8
           C float size: 4
          C double size: 8
         C pointer size: 8
           C char align: 1
           C bool align: skipped
            C int align: 4
```

```
          C float align: 4
         C double align: 8
           C++ compiler: g++
  C++ compiler absolute: */gcc/7.1.0/bin/g++
          Fort compiler: gfortran
      Fort compiler abs: */gcc/7.1.0/bin/gfortran
        Fort ignore TKR: no
  Fort 08 assumed shape: no
     Fort optional args: no
         Fort INTERFACE: no
   Fort ISO_FORTRAN_ENV: no
      Fort STORAGE_SIZE: no
     Fort BIND(C) (all): no
     Fort ISO_C_BINDING: no
Fort SUBROUTINE BIND(C): no
      Fort TYPE,BIND(C): no
Fort T,BIND(C,name="a"): no
           Fort PRIVATE: no
         Fort PROTECTED: no
          Fort ABSTRACT: no
      Fort ASYNCHRONOUS: no
         Fort PROCEDURE: no
        Fort USE...ONLY: no
          Fort C_FUNLOC: no
Fort f08 using wrappers: no
        Fort MPI_SIZEOF: no
      Fort integer size: 4
      Fort logical size: 4
Fort logical value true: 77
         Fort real size: skipped
     Fort dbl prec size: skipped
         Fort cplx size: skipped
     Fort dbl cplx size: skipped
     Fort integer align: skipped
        Fort real align: skipped
    Fort dbl prec align: skipped
        Fort cplx align: skipped
    Fort dbl cplx align: skipped
            C profiling: yes
          C++ profiling: no
  Fort mpif.h profiling: no
 Fort use mpi profiling: no
  Fort use mpi_f08 prof: no
         C++ exceptions: no
         Thread support: posix (MPI_THREAD_MULTIPLE: yes,
                         OPAL support: yes,
                         OMPI progress: no,
                         ORTE progress: yes,
                         Event lib: yes)
          Sparse Groups: no
```

```
           Build CFLAGS: -O3 -DNDEBUG -finline-functions
                         -fno-strict-aliasing -mcx16 -pthread
         Build CXXFLAGS: -O3 -DNDEBUG -finline-functions -pthread
          Build FCFLAGS:
          Build LDFLAGS:
             Build LIBS: -lrt -lm -lutil  -lz
    Wrapper extra CFLAGS: -pthread
  Wrapper extra CXXFLAGS: -pthread
   Wrapper extra FCFLAGS:
   Wrapper extra LDFLAGS: -Wl,-rpath -Wl,@{libdir} -Wl,
                          --enable-new-dtags
      Wrapper extra LIBS: -lm -ldl -lutil  -lrt -lz
  Internal debug support: no
  MPI interface warnings: yes
     MPI parameter check: runtime
Memory profiling support: no
Memory debugging support: no
              dl support: yes
   Heterogeneous support: no
 mpirun default --prefix: no
       MPI_WTIME support: native
     Symbol vis. support: yes
   Host topology support: yes
            IPv6 support: no
      MPI1 compatibility: no
          MPI extensions: affinity, cuda, pcollreq
   FT Checkpoint support: no (checkpoint thread: no)
   C/R Enabled Debugging: no
  MPI_MAX_PROCESSOR_NAME: 256
    MPI_MAX_ERROR_STRING: 256
     MPI_MAX_OBJECT_NAME: 64
        MPI_MAX_INFO_KEY: 36
        MPI_MAX_INFO_VAL: 256
       MPI_MAX_PORT_NAME: 1024
  MPI_MAX_DATAREP_STRING: 128
```