Elastisches Ressourcenmanagement: Vergleich von Asynchronous Many-Task (AMT) und Dynamic Processes with PSets (DPP)

Bachelor Thesis

Vorgelegt im Fachbereich 16 – Elektrotechnik/Informatik
Fachgebiet Software Engineering
der Universität Kassel

von Nick Bietendorf

Erstgutachter:
Dr. Jonas Posner

Zweitgutachter:
Prof. Dr. Gerd Stumme

Eingereicht am 20. Februar 2025 in Kassel

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht.

Kassel, 20. Februar 2025

Nick Bietendorf

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

In der heutigen, zunehmend datengetriebenen Welt steigen die Anforderungen an die Rechenleistung und Datenverarbeitung moderner Computersysteme kontinuierlich an. Komplexe Problemstellungen, wie die Simulation von Klimamodellen, die Vorhersage von Wetterphänomenen oder die Analyse großer Datensätze, erfordern eine immense Rechenkapazität, die oft die Möglichkeiten herkömmlicher Einzelrechner übersteigt. Um diese Herausforderungen zu bewältigen, werden Supercomputer entwickelt – leistungsstarke Cluster, die aus einer Vielzahl von unabhängigen Rechnern (Knoten) bestehen, die über ein Hochgeschwindigkeitsnetzwerk miteinander verbunden sind und gemeinsam komplexe Probleme lösen können.

Ein zentrales Merkmal von Supercomputern ist die verteilte Parallelisierung, bei der die Arbeitslast auf zahlreiche Knoten aufgeteilt wird. Der Entwickler muss dabei sicherstellen, dass komplexe Probleme in kleinere, unabhängige Teilaufgaben zerlegt werden, die parallel von den einzelnen Knoten bearbeitet werden können. Dieser Ansatz ermöglicht eine erheblich schnellere Verarbeitung von Berechnungen im Vergleich zu einem einzelnen Rechner.

Die Verwaltung der Ressourcen für parallele Berechnungen übernehmen Job-Scheduler (auch Ressourcen-Manager genannt) wie Slurm [1]. Sie sind dafür verantwortlich, Ressourcen (z. B. Knoten) an die jeweiligen Jobs zuzuweisen und zu reservieren. Ein Job repräsentiert dabei ein Programm, das auf einem oder mehreren Knoten ausgeführt wird. Job-Scheduler berücksichtigen dabei verschiedene Parameter, wie die Anzahl der benötigten Knoten und die maximale Laufzeit, um eine effiziente Ressourcennutzung sicherzustellen.

Aktuelle Job-Scheduler arbeiten mit sogenannten *rigid Jobs*, die eine feste Anzahl von Knoten benötigen und während der Ausführung nicht angepasst werden können. Dies führt zu ineffizienter Ressourcennutzung und langen Wartezeiten, insbesondere wenn nicht genügend Ressourcen verfügbar sind [2]. Ein anschauliches Beispiel: Wenn zwei Jobs jeweils drei Knoten benötigen, aber nur fünf Knoten insgesamt zur Verfügung stehen, muss einer der Jobs warten, bis die benötigten Knoten frei werden.

In der Zwischenzeit bleiben zwei Knoten ungenutzt. Solche Wartezeiten sind nicht nur ineffizient, sondern können auch die Leistung des gesamten Supercomputers beeinträchtigen, da Anwendungen nicht ausgeführt werden können, obwohl Ressourcen verfügbar sind.

Um diese Herausforderungen zu bewältigen, wurde das Konzept der Elastizität entwickelt. Elastizität bezeichnet die Fähigkeit von Anwendungen, ihre Ressourcennutzung während der Laufzeit dynamisch anzupassen. Ein spezieller Fall sind malleable Jobs, bei denen entweder der Ressourcen-Manager oder die Anwendung die Initiative für eine Prozessänderung ergreift. Falls die Anwendung die Änderung initiiert, erfolgt eine Abstimmung mit dem Ressourcen-Manager, um die Anzahl der genutzten Knoten zur Laufzeit anzupassen.

Diese Anpassung geschieht durch *Grow*- und *Shrink*-Operationen, bei denen Ressourcen entweder hinzugefügt oder freigegeben werden. Beispielsweise kann ein Job, der ursprünglich drei Knoten benötigt, durch eine Shrink-Operation auf zwei Knoten reduziert werden. Die dadurch freigewordenen Ressourcen stehen anderen Jobs zur Verfügung, was eine effizientere Nutzung der Gesamtressourcen ermöglicht.

Diese dynamische Anpassungsfähigkeit verbessert nicht nur die Effizienz der Ressourcennutzung, sondern kann auch die Wartezeiten für Jobs verringern [3]. Zudem ermöglicht sie eine schnellere Ausführung von Anwendungen, wenn zusätzliche Ressourcen verfügbar sind. Die Umsetzung von Elastizität erfordert jedoch Unterstützung des gesamten HPC (High Perfomance Computing) Software Stacks um diese flexible Ressourcenverwaltung zu ermöglichen.

Das weit verbreitete Message Passing Interface (MPI) [4], das auf den meisten Supercomputern als Standard eingesetzt wird, bietet derzeit keine native Unterstützung für elastische Jobs. Zwar wurden verschiedene Ansätze entwickelt, wie etwa [5, 6, 7, 8, 9, 10], diese konnten sich bisher in der Praxis aber nicht durchsetzen. Eine neuere Entwicklung ist das Prinzip der *Dynamic Processes with PSets (DPP)* [11], das die dynamische Anpassung der Anzahl von Prozessen zur Laufzeit ermöglicht. Basierend auf diesem Ansatz wurde das Programmiermodell MPI-DPP [12] entwickelt, das die Verwaltung von Prozessgruppen, sogenannten Prozess-Sets (PSets), unterstützt. PSets ermöglichen es, Prozesse dynamisch durch Grow- und Shrink-Operationen zu erweitern oder zu verkleinern, wodurch eine flexible Ressourcenanpassung während der Laufzeit realisiert wird.

Allerdings führt die Möglichkeit der dynamischen Anpassung auch zu einer erhöhten Komplexität in der Anwendungsentwicklung. Entwickler müssen sich mit den internen Details des Ressourcenmanagements auseinandersetzen und ihre Anwen-

dungen entsprechend anpassen, um die Vorteile der dynamischen Prozessverwaltung vollständig nutzen zu können. Dies erfordert ein tieferes Verständnis der zugrunde liegenden Mechanismen und wird den Entwicklungsaufwand im Vergleich zum traditionellen MPI erhöhen.

Ein vielversprechender alternativer Ansatz ist das Asynchronous Many-Tasking (AMT)-Modell. Dieses Konzept basiert darauf, ein Problem in viele kleine, unabhängige Tasks zu zerlegen und diese in einem zentralen Task-Pool zu verwalten. Die Tasks werden anschließend asynchron und dynamisch von den Prozessen abgearbeitet.

Im Gegensatz zur klassischen Parallelisierung ermöglicht AMT eine feingranulare Aufgabenverteilung, ein dynamisches Load Balancing, das während der Laufzeit vom AMT-Modell automatisch übernommen wird, und eine höhere Flexibilität in der Ressourcennutzung. Dadurch eignet sich dieses Modell besonders für komplexe und ungleichmäßige Arbeitslasten.

Eine Umsetzung dieses Prinzips bietet das APGAS-Modell (*Asynchronous Partitioned Global Address Space*)[13]. Es baut auf dem PGAS-Modell auf und erleichtert die Entwicklung effizienter paralleler Anwendungen. Eine Weiterentwicklung, das APGAS-GLB (*Global Load Balancing*)[14], erweitert das Modell um globales Load Balancing sowie eine automatische Ressourcenanpassung. Darüber hinaus abstrahiert APGAS-GLB die Komplexität des Ressourcenmanagements und der Arbeitsverteilung, wodurch die Implementierung von malleable Programmen erheblich vereinfacht wird.

Obwohl diese Programmiermodelle existieren, sind sie bisher wenig verbreitet und werden in der Praxis kaum eingesetzt. Das Ziel dieser Arbeit ist die Untersuchung und Evaluierung der beiden elastischen Programmiermodelle MPI-DPP und APGAS-GLB. Dabei werden die Programmiermodelle anhand der Benchmarks PI und UTS miteinander verglichen. Während die Implementierungen der Benchmarks für APGAS-GLB bereits vorhanden sind, mussten sie für MPI-DPP neu entwickelt werden, wobei die Funktionalität des APGAS-GLB in MPI-DPP nachgebildet wurde, um eine hohe Vergleichbarkeit zu gewährleisten.

Die Arbeit untersucht die Programmiermodelle hinsichtlich ihrer Programmierproduktivität, Skalierbarkeit und Effizienz bei Ressourcenänderungen. Dabei zeigt sich, dass APGAS-GLB im Allgemeinen eine höhere Programmierproduktivität aufweist. Dies liegt daran, dass es weniger Codezeilen und Konstrukte benötigt, da das Ressourcenmanagement und die Datenverteilung wegabstrahiert sind. Zudem bietet APGAS-GLB eine bessere Skalierbarkeit. Im Gegensatz dazu ist die Laufzeit der Ressourcenanpassung in MPI-DPP effizienter, und die Gesamtlaufzeit

der Benchmarks ist geringer. MPI-DPP bietet außerdem eine größere Flexibilität und Kontrolle über die Anwendung, da Entwickler sich intensiver mit den internen Details des Ressourcenmanagements auseinandersetzen müssen.

Die Arbeit gliedert sich in fünf Kapitel: Nach der Einleitung bietet Kapitel 2 eine Einführung in das Konzept der Ressourcen-Elastizität sowie einen Überblick über die Programmiermodelle APGAS, GLB, MPI und DPP. In Kapitel 3 werden die Benchmarks *PI* und *UTS* vorgestellt und die Programmiermodelle evaluiert, wobei die Ergebnisse der Experimente präsentiert und diskutiert werden. Kapitel 4 widmet sich verwandten Arbeiten, bevor in Kapitel 5 die Ergebnisse zusammengefasst und ein Ausblick auf zukünftige Forschung gegeben wird.

2 Hintergrund

Im folgenden Abschnitt werden die Grundlagen der Ressourcen-Elastizität (2.1) sowie die Konzepte von APGAS (2.2), GLB (2.3), MPI (2.4) und DPP (2.5) erläutert.

2.1 Ressourcen-Elastizität

Ressourcen-Elastizität beschreibt die Fähigkeit eines Supercomputers mithilfe von Software, sich dynamisch an wechselnde Arbeitslasten, wie z. B. parallele Berechnungen, Datenverarbeitung oder Echtzeit-Simulationen, anzupassen, indem Ressourcen bedarfsgerecht hinzugefügt oder entfernt werden. Diese Anpassungsfähigkeit ermöglicht eine bessere Auslastung der Rechenleistung und steigert die Gesamtperformance des Supercomputers.

Eine der zentralen Herausforderungen der Ressourcen-Elastizität ist die Skalierbarkeit von Anwendungen. Skalierbarkeit beschreibt die Fähigkeit einer Anwendung, ihre Leistung durch die Nutzung zusätzlicher Ressourcen zu steigern. Eine effiziente Skalierung ist entscheidend, um die Vorteile der Ressourcen-Elastizität vollständig auszuschöpfen. Eine unzureichende Skalierbarkeit kann hingegen zu Ressourcenverschwendung und ineffizienter Nutzung führen. Daher ist es essenziell, Anwendungen so zu entwickeln, dass sie sich flexibel und effizient an veränderte Ressourcenbedingungen anpassen können.

Ein weiterer kritischer Aspekt der Ressourcen-Elastizität ist der effiziente Ressourcenwechsel. Da sich die verfügbaren Ressourcen dynamisch ändern, müssen Anwendungen in der Lage sein, ihre Arbeitslast während der Laufzeit effizient zu verteilen und zu koordinieren. Schnelle und häufige Anpassungen sind erforderlich, um in Echtzeit auf wechselnde Anforderungen reagieren zu können. Dies erfordert eine präzise Kommunikation und Koordination zwischen den Prozessen, um die optimale Leistung des Supercomputers sicherzustellen.

Wenn diese Herausforderungen erfolgreich gemeistert werden, kann die Ressourcen-

Elastizität die Leistungsfähigkeit und Effizienz von Supercomputern erheblich steigern.

2.2 APGAS

Der folgende Abschnitt erläutert die Grundlagen von APGAS (Asynchronous Partitioned Global Address Space) und die Implementierung in Java. Dabei wird auf die Funktionsweise von Places, die dynamische Anpassung von der Anzahl von Places und die Grow- und Shrink-Mechanismen eingegangen.

2.2.1 Einführung in PGAS und APGAS

APGAS (Asynchronous Partitioned Global Address Space) ist ein asynchrones Programmiermodell [13], das auf dem PGAS-Modell (Partitioned Global Address Space) basiert. PGAS ermöglicht es, den verteilten Speicher mehrerer Knoten als einen gemeinsamen, global adressierbaren Speicher zu betrachten, der in logische Partitionen unterteilt ist. Jede Partition verfügt über einen eigenen lokalen Speicher, bleibt aber Teil des globalen Adressraums. Dadurch können Speicherbereiche auf anderen Knoten direkt adressiert werden, ohne dass explizite Kommunikationsmechanismen wie bei MPI erforderlich sind.

APGAS erweitert PGAS, indem es asynchrone Tasks erlaubt, die sowohl lokal als auch entfernt ausgeführt werden können. Dies ermöglicht die parallele Ausführung von Tasks zur Laufzeit, ohne auf deren Ergebnisse warten zu müssen.

2.2.2 Implementierung von APGAS in Java

Diese Arbeit verwendet die Open-Source-Implementierung von APGAS [1] in Java. Im Folgenden bezieht sich der Begriff *APGAS* ausschließlich auf diese Implementierung.

Die Partitionen des globalen Adressraums werden als *Places* bezeichnet, denen jeweils ein *Knoten* zugeordnet ist. Die Places sind fortlaufend nummeriert (0 bis n-1) und enthalten mehrere Prozessorkerne, die als *Worker* bezeichnet werden. Diese Worker führen asynchrone Tasks aus, die sowohl lokal als auch remote erzeugt werden können.

[1] https://github.com/projectwagomu/apgas.git

Jeder Place ist ein eigenständiger Prozess, der in einer separaten JVM (Java Virtual Machine) läuft. Die Worker innerhalb eines Place sind hingegen Threads. Places nutzen Multi-Threading, um ihre Worker-Threads zu verwalten und parallele Berechnungen effizient durchzuführen.

Die Methode `asyncAt()` ermöglicht die Ausführung einer Task an einem bestimmten Place. Um sicherzustellen, dass alle asynchronen Tasks beendet sind, bevor das Programm terminiert, wird `asyncAt()` innerhalb eines finish-Blocks ausgeführt. Dieser Block garantiert, dass das Programm erst dann beendet wird, wenn alle innerhalb des Blocks gestarteten Tasks abgeschlossen sind.

Listing 2.1 zeigt ein simples *Hello World*-Programm, das APGAS in Java verwendet, während Listing 2.2 die mögliche Ausgabe dieses Programms darstellt. Da die Tasks asynchron ausgeführt werden, kann die Reihenfolge der Ausgabe variieren und ist nicht deterministisch.

```
1  import apgas.Configuration;
2  import apgas.Constructs;
3  import apgas.Place;
4  import static apgas.Constructs.*;
5
6  public class HelloWorldExample {
7   public static void main(String[] args) {
8       finish(() -> {
9           for (Place p : places()) {
10              asyncAt(p, () -> {
11                  System.out.println("Hello from Place " + here());
12              });
13          }
14      });
15  }
16 }
```

Listing 2.1: Beispiel für die Verwendung von APGAS in Java

```
1      Hello from Place 1
2      Hello from Place 3
3      Hello from Place 0
4      Hello from Place 2
```

Listing 2.2: Mögliche Ausgabe des Programms aus dem Code Beispiel 2.1

2.2.3 Dynamische Anpassung der Places

APGAS bietet die Möglichkeit, die Anzahl der *Places* dynamisch anzupassen. Dabei wird zwischen zwei Ansätzen unterschieden: *malleable*[3] und *evolving*[15].

- `Malleable:` Die Anpassung erfolgt extern, z. B. durch einen Job-Scheduler, der die Ressourcenverteilung steuert, oder durch die Anwendung in Zusammenarbeit mit dem Ressourcen-Manager.

- `Evolving:` Die Anpassung wird von der Anwendung selbstständig vorgenommen, entschieden und erfolgt ohne externe Steuerung.

In dieser Arbeit wird der `malleable` Ansatz verwendet, bei dem das Programm in festen Zeitintervallen eigenständig Ressourcenänderungen vornimmt. Diese Änderungen simulieren eine Initialisierung durch einen Ressourcen-Manager, wobei Places zur Laufzeit hinzugefügt oder entfernt werden. Die Anpassung erfolgt durch `Grow-` und `Shrink-`Operationen, die über das `MalleableHandler-Interface` realisiert werden.

Entwickler müssen die folgenden Methoden implementieren:

- `List<Place> preShrink()`: Vorbereitung vor dem Entfernen von Places.

- `void postShrink(int nbPlaces, List<Place> removedPlaces)`: Abschluss nach dem Entfernen von Places.

- `void preGrow(int nbPlaces)`: Vorbereitung vor dem Hinzufügen neuer Places.

- `void postGrow(int nbPlaces, List<Place> newPlaces)`: Abschluss nach dem Hinzufügen neuer Places.

Shrink: Beim Entfernen von Places wird zuerst `preShrink` aufgerufen, um Tasks auf verbleibende Places zu verteilen. Anschließend werden die Places entfernt, und `postShrink` wird ausgeführt, um die Transition abzuschließen.

Grow: Beim Hinzufügen neuer Places wird zuerst `preGrow` aufgerufen, um Tasks für die neuen Places vorzubereiten. Nach dem Hinzufügen der Places wird `postGrow` ausgeführt, um die Transition abzuschließen.

2.3 GLB

Das APGAS-GLB[2] (Global Load Balancing) [14] erweitert das APGAS-Programmiermodell, um einen globalen Lastenausgleich zu ermöglichen. Es basiert auf der Einrichtung fester Kommunikationskanäle, über die Places Tasks von anderen Places „stehlen“ können. Ursprünglich wurde dieses Programmiermodell in der X10-Bibliothek [14] entwickelt und später in APGAS integriert.

Der Lastenausgleich folgt dem Work-Stealing-Prinzip: Wenn ein Place keine eigenen Tasks mehr hat, sucht er gezielt bei anderen Places nach Tasks. Die Kommunikation zwischen den Places wird dabei durch einen LifeLine-Graphen gesteuert. Dieser Graph definiert, von welchen spezifischen Nachbarn ein Place Tasks stehlen darf, wodurch die Interaktionen effizient und strukturiert bleiben. Der LifeLine-Graph ist nach dem Prinzip eines Hyperwürfels [16] aufgebaut.

GLB arbeitet mit Bags, die als Arbeitswarteschlangen (Working Queues) fungieren. Jeder Place besitzt einen eigenen Bag, in dem er Tasks speichert. Wenn ein Place keine eigenen Tasks mehr hat, kann er Tasks aus den Bags anderer Places stehlen. Um das GLB-Programmiermodell zu nutzen, müssen die folgenden Methoden überschrieben werden und mit einer dementsprechenden Logik für das jeweilige Problem gefüllt werden. Work-Stealing und Ressourcenanpassungen werden dann automatisch durch das APGAS-GLB verwaltet.

- `boolean process(int)`: Verarbeitet die angegebene Anzahl von Tasks aus dem Bag des Workers.
- `B split(boolean)`: Teilt den Bag des Workers in zwei Hälften auf. Wenn der Parameter *boolean* auf *true* gesetzt ist, wird der gesamte Bag übergeben.
- `void merge(B)`: Fügt die Tasks aus dem übergebenen Bag in den eigenen Bag des Workers ein.
- `boolean isEmpty()`: Gibt *true* zurück, wenn der Bag des Workers leer ist.
- `boolean isSplittable()`: Gibt *true* zurück, wenn der Bag des Workers geteilt werden kann.
- `void submit(R)`: Überträgt die Ergebnisse aus dem Bag in eine Ergebnisinstanz *R*.

[2] https://github.com/projectwagomu/lifelineglb.git

2.4 MPI

Das Message Passing Interface (MPI) [4] ist ein weit verbreiteter Standard für die parallele Prozesskommunikation. Es ermöglicht den Nachrichtenaustausch sowohl zwischen Prozessen auf verschiedenen Knoten (Nodes) als auch innerhalb eines einzelnen Knotens. Jeder Prozess stellt dabei eine eigenständige Einheit dar, die unabhängig arbeitet und über definierte Kommunikationsmechanismen mit anderen Prozessen interagiert.

MPI organisiert Prozesse in Kommunikationsgruppen, wodurch eine strukturierte und effiziente Datenübertragung möglich ist. Der Standard wurde speziell für HPC entwickelt und bietet eine leistungsfähige und flexible Schnittstelle zur Prozesskoordination.

Für die Nutzung von MPI in C++ gehört Open MPI[17] zu den bekanntesten und am weitesten verbreiteten Implementierungen. Open MPI setzt auf Open PMIx (Process Management Interface for Exascale)[18], eine Schnittstelle zur effizienten Verwaltung von Prozessen und Ressourcen. PMIx [19] fungiert als Middleware zwischen dem MPI-Laufzeitsystem und dem darunterliegenden Betriebssystem und erleichtert die Koordination mit Job-Schedulern wie Slurm [1].

Zusätzlich verwendet Open MPI PRRTE (PMIx Reference Runtime Environment) [20] als Laufzeitumgebung. PRRTE stellt die erforderliche Infrastruktur bereit, um MPI-Prozesse in einem Cluster zu starten, zu verwalten und zu terminieren. Es dient als Referenzimplementierung für PMIx und ermöglicht eine nahtlose Integration mit Open MPI.

Die enge Verzahnung von Open MPI, Open PMIx und PRRTE sorgt für eine effiziente, flexible und skalierbare Ausführung von MPI-Anwendungen auf Hochleistungsrechnern und Supercomputern.

Wichtige MPI Funktionen

Die folgenden Funktionen bilden die Grundlage für die Arbeit mit MPI:

- `MPI_Session_Init`: Initialisiert MPI und erstellt eine Kommunikationsumgebung.
- `MPI_set_info`: Legt Konfigurationsinformationen für einen Kommunikator fest.

- `MPI_get_info`: Ruft Konfigurationsinformationen eines Kommunikators ab.
- `MPI_Comm_size`: Gibt die Anzahl der Prozesse in einem Kommunikator zurück.
- `MPI_Comm_rank`: Bestimmt den Rang (eine eindeutige Kennung) eines Prozesses innerhalb eines Kommunikators.
- `MPI_Comm_create_from_group`: Erstellt einen neuen Kommunikator aus einer definierten Gruppe von Prozessen.
- `MPI_Send`: Sendet synchron eine Nachricht an einen anderen Prozess.
- `MPI_Recv`: Empfängt synchron eine Nachricht von einem anderen Prozess.
- `MPI_IRecv`: Führt einen asynchronen Empfang einer Nachricht aus.
- `MPI_Disconnect`: Trennt die Verbindung zu einem Prozess oder einer Gruppe von Prozessen.
- `MPI_Finalize`: Beendet MPI und gibt alle zugehörigen Ressourcen frei.

Ein Beispiel für die Verwendung von MPI in C++ ist in Listing 2.3 dargestellt. Es zeigt ein einfaches *"Hello World-Programm"*, das MPI initialisiert (Zeile 10), eine Kommunikationsgruppe erstellt (Zeile 12 - 14), den Rang des aktuellen Prozesses bestimmt (Zeile 16) und eine entsprechende Ausgabe generiert (Zeile 18). Eine mögliche Ausgabe ist in Listing 2.4 gezeigt.

Da die Prozesse asynchron arbeiten, kann die Reihenfolge der Ausgabe variieren und ist nicht deterministisch.

```c
#include <mpi.h>
#include <iostream>

int main(int argc, char** argv) {
  MPI_Session session;
  MPI_Group world_group;
  MPI_Comm comm;
  int rank;

  MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session);

  MPI_Group_from_session_pset(session,"mpi://WORLD",&world_group);

  MPI_Comm_create_from_group(world_group, "0", MPI_INFO_NULL,
     MPI_ERRORS_ARE_FATAL, &comm);

  MPI_Comm_rank(comm, &rank);

  printf("Hello from %d \n", rank);

  MPI_Comm_disconnect(&comm);
  MPI_Group_free(&world_group);
  MPI_Session_finalize(&session);

  return 0;
}
```

Listing 2.3: MPI-basiertes Hello World Programm in C++

```
Hello from 1
Hello from 2
Hello from 0
Hello from 3
```

Listing 2.4: Mögliche Ausgabe des Programms aus 2.3

2.5 DPP

Der Ansatz *Dynamic Processes with PSets (DPP)* [11] definiert grundlegende Designprinzipien für die dynamische Verwaltung von Prozessen in parallelen Anwendun-

gen. Das bedeutet, dass die Anzahl der Prozesse zur Laufzeit dynamisch angepasst werden kann, um Anwendungen flexibler und effizienter zu gestalten. DPP basiert auf einer engen Zusammenarbeit zwischen Anwendungen und Ressourcenverwaltungssoftware. Das Ziel ist es, sowohl eine flexible Umsetzung verschiedener paralleler Programmiermuster zu ermöglichen als auch die Ressourcennutzung im gesamten System effizient zu optimieren.

In den folgenden Abschnitten werden die Designprinzipien von DPP (2.5.1) sowie die Funktionalität (2.5.2), die verfügbaren Operationen und Funktionen (2.5.3) und die Einschränkungen (2.5.4) einer MPI-Implementierung von DPP (MPI-DPP [12]) näher beschrieben.

2.5.1 DPP-Designprinzipien

DPP stützt sich auf zwei zentrale Designprinzipien, die die Grundlage für die dynamische Prozessverwaltung bilden:

- `Dynamic Process Management (DPM)`: DPM basiert auf Prozess-Sets (*PSets*), die Gruppen von Prozessen repräsentieren, die gemeinsam verwaltet werden. Ergänzt wird dies durch Prozess-Set-Operationen (*PSetOps*), welche die Erstellung, Verwaltung und Abfrage von PSets ermöglichen. Aus graphentheoretischer Sicht kann man sich PSets als Knoten eines Graphen vorstellen, während PSetOps die Kanten darstellen, die die Beziehungen zwischen den PSets beschreiben. Der PSet-Graph verändert sich dynamisch durch die von der Anwendung oder dem Ressourcen-Manager ausgeführten PSetOps.

- `Dynamic Resource Allocation (DRA)`: DRA basiert auf Optimierungsinformationen, die mittels einer *Collaborative Optimization Language* (COL) [11] ausgedrückt werden. Diese Sprache enthält lokale Optimierungsdaten und ermöglicht die Formulierung kollektiver Optimierungsprobleme. Die in COL beschriebenen Informationen werden mit den PSetOps verknüpft und dienen als Grundlage für Entscheidungen im dynamischen Ressourcenmanagement des Systems.

2.5.2 Funktionalität von MPI-DPP

MPI-DPP dient als Grundlage für die Implementierung eines dynamischen Ressourcenmanagements. Es ist als zusätzliche Schicht über MPI und PMIx integriert und

erweitert bestehende Frameworks wie Open-MPI, Open-PMIx und PRRTE.

Die Implementierung erweitert das MPI-Session-Interface, um PSetOps zu unterstützen. Dadurch können Prozesse dynamisch hinzugefügt, entfernt oder PSet Informationen gespeichert und abgefragt werden. In Kombination mit einem Ressourcen-Manager fragt die Anwendung über PSetOps die Erlaubnis für Änderungen an, welche der Ressourcen-Manager entweder genehmigt oder ablehnt. Läuft die Anwendung ohne Ressourcen-Manager, werden die Änderungen direkt ausgeführt, sofern dies möglich ist. In der vorliegenden Arbeit wird kein Ressourcen Manager für die Benchmarks verwendet, statt dessen wird die direkte Ausführung der PSetOps genutzt.

2.5.3 Operationen und Funktionen von MPI-DPP

MPI-DPP stellt Funktionen bereit, die die Erstellung, Verwaltung und Abfrage von PSets in der Anwendung ermöglichen:

- `MPI_Session_dyn_v2a_psetop`: Erstellt ein neues Prozess-Set (PSet) und fügt oder entfernt Prozesse.
- `MPI_Session_dyn_v2a_psetop_nb`: Nicht blockierende Variante von `MPI_Session_dyn_v2a_psetop`.
- `MPI_Session_dyn_v2a_query_psetop`: Gibt Informationen über das PSet und den Operanden zurück, sofern eine Änderung der Ressourcen erfolgt.
- `MPI_Session_dyn_v2a_query_psetop_nb`: Nicht blockierende Variante von `MPI_Session_dyn_v2a_query_psetop`.
- `MPI_Session_set_pset_data`: Setzt Daten für ein PSet.
- `MPI_Session_get_pset_data`: Ruft Daten für ein PSet ab.

Ein zentrales Merkmal von MPI-DPP ist, dass keine automatische Datenverteilung zwischen Prozessen erfolgt. Die korrekte Verteilung und Synchronisierung der Daten liegt vollständig in der Verantwortung der Entwickler. Da Abschnitt 3.2.3 die Verwendung von MPI-DPP anhand eines Beispiels detailliert erläutert, wird an dieser Stelle auf eine zusätzliche Darstellung verzichtet.

2.5.4 Limitierung von MPI-DPP

Derzeit ist MPI-DPP ein forschungsbasierter Prototyp und unterliegt daher noch einigen Einschränkungen. Im Gegensatz zu herkömmlichem MPI, das die Verwaltung einzelner Prozesse auf logischen Kernen ermöglicht, erlaubt DPP momentan nur das Hinzufügen oder Entfernen ganzer Knoten mit einer fest definierten Anzahl von Prozessen. Zum Beispiel kann eine Anwendung, die auf einem Knoten mit 8 Prozessen gestartet wurde, nur durch das Hinzufügen weiterer Knoten mit jeweils 8 Prozessen erweitert werden. Das gleiche gilt für das Entfernen von Knoten. Diese Einschränkung verringert die Flexibilität bei der dynamischen Ressourcenzuweisung.

3 Evaluation elastischer Anwendungen

In diesem Kapitel werden die Implementierungen der Benchmarks Pi für statische Arbeitslasten und UTS (Unbalanced Tree Search) für dynamische Arbeitslasten in den Programmiermodellen APGAS-GLB und MPI-DPP vorgestellt und evaluiert. Abschnitt 3.1 beschreibt die Benchmarks Pi und UTS und begründet deren Eignung zur Evaluierung dynamischer paralleler Programmiermodelle. Die detaillierte Darstellung der Implementierung dieser Benchmarks in APGAS-GLB und MPI-DPP erfolgt in Abschnitt 3.2. In Abschnitt 3.3 wird die Programmiererproduktivität der beiden Programmiermodelle verglichen. Schließlich werden in Abschnitt 3.4 die Ergebnisse der Benchmarks hinsichtlich Skalierbarkeit und Ressourcenanpassung analysiert.

3.1 Benchmarks

Für die folgenden Experimente wurden die Benchmarks Pi und UTS ausgewählt. Beide Benchmarks bieten eine solide Grundlage, um die Leistungsfähigkeit und Eigenschaften der getesteten Programmiermodelle zu evaluieren und miteinander zu vergleichen.

3.1.1 Pi

Der Pi-Benchmark verwendet die Monte-Carlo-Simulation [21] zur Approximation der Kreiszahl Pi. Hierbei werden zufällige Punkte innerhalb eines Quadrats generiert, und es wird ermittelt, wie viele dieser Punkte innerhalb eines eingeschriebenen Kreises liegen. Das Verhältnis der Punkte innerhalb des Kreises zur Gesamtanzahl der Punkte ermöglicht die Näherung von Pi.

Dieser Benchmark ist zur Evaluierung statischer paralleler Programmiermodelle geeignet, da:

- Hohe Rechenlast: Der Pi-Benchmark erzeugt beliebig viele rechenintensive Aufgabe, bei der keine Abhängigkeiten zwischen den Teilaufgaben bestehen.
- Minimale Kommunikation: Die Prozesse arbeiten unabhängig voneinander, was die Kommunikation auf ein Minimum reduziert.
- Simple Implementierung: Aufgrund der einfachen Struktur und geringen Komplexität des Benchmarks lässt er sich mit minimalem Aufwand implementieren. Dies macht ihn besonders geeignet, um parallele Programmiermodelle zu evaluieren, da der Fokus auf der Parallelisierbarkeit und nicht auf der Komplexität der Implementierung liegt. [22]

Der Pi-Benchmark ist daher hilfreich, um die Skalierbarkeit und Effizienz eines Programmiermodells bei rechenlastigen Aufgaben mit statischer Arbeitslast zu bewerten.

3.1.2 UTS

Der UTS-Benchmark (Unbalanced Tree Search) [23] simuliert die parallele Suche in einem unbalancierten Baum mit zufälliger Struktur. Sein Hauptziel ist es, die Effizienz paralleler Algorithmen bei der Verarbeitung dynamischer und ungleichmäßig verteilter Arbeitslasten zu bewerten.

Charakteristische Eigenschaften von UTS:

- Unbalancierte Arbeitslast: Die ungleichmäßige Verteilung der Baumknoten führt zu einer dynamischen Arbeitslast, die eine effiziente Lastverteilung zur Laufzeit erfordert.
- Hohe Kommunikationsanforderungen: Die parallele Verarbeitung der Knoten erfordert eine effiziente Koordination und Kommunikation zwischen den Prozessen, was zu einem hohen Kommunikationsaufkommen führt.
- Test der Elastizität: Der Benchmark prüft, wie gut ein Programmiermodell auf die unvorhersehbare Verteilung der Arbeitslast reagieren kann.

Der UTS-Benchmark eignet sich daher zur Bewertung der Leistungsfähigkeit eines Programmiermodells in realistischen Szenarien mit variabler Arbeitslast und hohen Kommunikationsanforderungen.

3.2 Implementierung der Benchmarks

Im Folgenden wird die Implementierung der Benchmarks Pi (3.2.1) und UTS (3.2.2) in APGAS-GLB und MPI-DPP, sowie den Teil der Ressourcenänderung (3.2.3) beschrieben. Dabei wird sich an der Implementierung von APGAS-GLB[1] orientiert, um eine Vergleichbarkeit der Benchmarks zu gewährleisten.

3.2.1 PI

APGAS-GLB: APGAS-GLB verteilt die initiale Gesamtanzahl der Iterationen n gleichmäßig auf m Tasks, die wiederum den verfügbaren Places zugewiesen werden. Jeder Task berechnet eigenständig die Anzahl der Punkte mithilfe der Monte-Carlo-Methode. Sobald ein Place alle ihm zugewiesenen Tasks abgeschlossen hat, beginnt er, Tasks von anderen Places zu stehlen (Work-Stealing). Dieser Prozess wird fortgesetzt, bis alle Tasks abgearbeitet sind.

Ein zentraler Vorteil von APGAS-GLB ist der Verzicht auf explizite Synchronisationspunkte. Da die Arbeit dynamisch verteilt und Ressourcenänderungen automatisch gehandhabt werden, erfolgt die Lastverteilung effizient und flexibel.

MPI-DPP: Bei MPI-DPP wird die initiale Gesamtanzahl der Iterationen n in benutzerdefinierte Abschnitte, die durch *total_iterations* bestimmt werden, unterteilt. Jeder dieser Abschnitte markiert einen möglichen Wechselpunkt zwischen Prozessen. Die Anzahl der Iterationen, die ein einzelner Prozess ausführt, ergibt sich aus der Division der Gesamtanzahl n durch *total_iterations* und anschließend durch die Anzahl der Prozesse.

Da die Berechnung von Pi eher für statische Arbeitslasten ausgelegt ist, erfolgt die Aufteilung der Iterationen ausschließlich statisch – ein Work-Stealing-Mechanismus wird hier nicht verwendet.

Der Pseudocode in Listing 3.1 zeigt die Implementierung des PI-Benchmarks mit MPI-DPP. Die *while*-Schleife (Zeile 3) fungiert als Synchronisationspunkt für Ressourcenänderungen. Die Funktion *distributeToProcesses()* (Zeile 4) verteilt die Arbeitslast auf die Prozesse, während *monteCarloIteration()* (Zeile 5) die Monte-Carlo-Methode ausführt.

Die Funktion *resource_change_needed()* (Zeile 6) überprüft, ob eine Anpassung der Ressourcen erforderlich ist, und *triggerRessourceChange()* (Zeile 7) setzt diese gegebenenfalls um. Die Schleife läuft so lange, bis alle Iterationen abgeschlossen sind.

[1] https://github.com/posnerj/PLM-APGAS-Applications/tree/master/src/examples

Anschließend werden die Ergebnisse an Prozess 0 zurückgesendet (Zeile 11), und die MPI-Session wird beendet (Zeile 12).

```
1  initMPI();
2
3  while (cur_iter < total_iterations) {
4      distributeToProcesses();
5      double pi = monteCarloIteration();
6      if(resource_change_needed()) {
7          triggerRessourceChange();
8      }
9      cur_iter++;
10 }
11 reduceToRoot();
12 finalizeMPI();
```

Listing 3.1: Pseudo Code Beispiel für MPI-DPP PI

3.2.2 UTS

APGAS-GLB: APGAS-GLB definiert jeden Knoten des Baums als eine Task, die von einem Place verarbeitet wird. Place 0 initialisiert den Baum, indem er den Wurzelknoten erzeugt und aufklappt, wodurch n Kindknoten entstehen, welche wiederum zu Tasks werden. Währenddessen beginnen die anderen Places, Tasks von Place 0 zu stehlen. Dadurch wird die Arbeit dynamisch auf die Places verteilt. Falls ein Place keine Tasks mehr zum Stehlen findet, wird er inaktiv und wartet auf neue Tasks. Dieser Ablauf wird fortgesetzt, bis alle Knoten des Baums vollständig bearbeitet sind.

MPI-DPP: In dieser Variante wurde ein grundlegendes Work-Stealing-Schema implementiert, um die Funktionsweise von APGAS-GLB nachzubilden. Ein statischer Ansatz, wie er bei der Berechnung von Pi eingesetzt wird, wäre hier ungeeignet, da UTS bei jeder Aufklappung eines Baumknotens eine unvorhersehbare Menge an neuer Arbeit erzeugt. Dies würde zu einer unausgeglichenen Lastverteilung führen. Um die Verteilung der Arbeit effizient zu steuern, wurde ein sogenannter Lifeline-Graph eingeführt, der die Kommunikation zwischen den Prozessen regelt. Dieser basiert auf der Struktur eines Hyperwürfels [16], wobei die Anzahl der Lifelines durch die Formel *log_2 (Anzahl der Prozesse)* bestimmt wird. Der Berechnungsprozess beginnt mit Prozess 0, der den Baum initialisiert, indem er den Wurzelknoten aufklappt.

Prozesse, die eine Lifeline zu Prozess 0 haben, beginnen daraufhin mit dem Stehlen von Tasks.

Kommt es zu einer Ressourcenanpassung, wird der Lifeline-Graph neu berechnet. Neue Prozesse integrieren sich in die Berechnung, indem sie aktiv Arbeit von anderen Prozessen stehlen. Sobald alle Knoten des Baums abgearbeitet sind, senden die Prozesse ihre Ergebnisse an Prozess 0. Das Pseudocode-Beispiel in Listing 3.2 zeigt die Implementierung des UTS-Benchmarks in MPI-DPP unter Verwendung des Work-Stealing-Schemas. Nach der Initialisierung von MPI (Zeile 1) und des Lifeline-Graphs (Zeile 2) beginnt die Verteilung und Verarbeitung der Arbeit.

Zunächst wird in einer Schleife (Zeile 3) überprüft, ob eine Terminierungsaufforderung von Prozess 0 vorliegt. Ist dies nicht der Fall, wechselt der Prozess in die nächste Schleife (Zeile 4), die so lange durchlaufen wird, wie noch Baumknoten zum Aufklappen vorhanden sind. Die Funktion *tree_nodes_expanding()* (Zeile 5) übernimmt das Aufklappen und Erstellen neuer Knoten, während gleichzeitig geprüft wird, ob Steal-Anfragen vorliegen. Falls ja, gibt der Prozess die Hälfte seiner Arbeit ab. Falls eine Ressourcenänderung erfolgen soll, wird dies ausschließlich von Prozess 0 geprüft (Zeile 6) und mit der Funktion *triggerRessourceChange()* (Zeile 7) durchgeführt.

Die Funktion *checkResourceChange()* (Zeile 9) überprüft, ob eine Anpassung der Ressourcen erforderlich ist, und führt diese gegebenenfalls durch. In der Stealing-Phase (Zeile 12-15) senden Prozesse Steal-Anfragen (Zeile 13) an ihre Lifeline-Partner. Falls ein Prozess noch Arbeit besitzt, aber keine Steal-Anfragen erhält, könnte es sein, dass seine Lifeline-Partner inaktiv sind. In diesem Fall werden sie mithilfe von *checkIdle()* (Zeile 17) reaktiviert. Hat ein Prozess keine Arbeit mehr und erhält auch keine neue, geht er in einen Inaktivitätszustand über und meldet dies Prozess 0. Falls notwendig, kann er später durch *goingIdleAndWaitForWakeUp()* (Zeile 20) wieder aktiviert werden. Sobald alle Knoten bearbeitet sind und Prozess 0 die Terminierung anfordert, senden die Prozesse ihre Ergebnisse an Prozess 0 (Zeile 22) und beenden sich.

```
1  initMPI();
2  initLifeLines();
3  while(!terminate) {
4      while(!nodes.isEmpty()) {
5          tree_nodes_expanding();
6          if(resource_change_needed()) {
7              triggerRessourceChange();
8          }
9          checkResourceChange();
10         // Stealing Phase
11         if(!nodes.size()) {
12             for(int i = 0; i < lifeLines.size(); i++) {
13                 if(stealing(lifeLines[i])) {
14                     break;
15             }
16         } else {
17             checkIdle();
18         }
19     }
20     goingIdleAndWaitForWakeUp();
21 }
22 reduceToRoot();
23 finalizeMPI();
```

Listing 3.2: Pseudo Code Beispiel für MPI-DPP UTS

3.2.3 Ressourcenänderungen

APGAS-GLB kann Ressourcenänderungsereignisse von außen empfangen und darauf reagieren. Dadurch entfällt für den Entwickler die Notwendigkeit, Ressourcenänderungen manuell zu implementieren, da das Programmiermodell dies automatisch übernimmt. Im Gegensatz dazu erfordert MPI-DPP eine explizite Implementierung sowohl der Ressourcenänderungen als auch der Verteilung der Arbeit bei einer steigenden oder sinkenden Anzahl von Prozessen. Dies erhöht die Komplexität des Codes und erfordert eine detaillierte Planung.

Listing 3.3 3.4 zeigt die Implementierung der Ressourcenänderung in MPI-DPP für UTS. Die Funktion *triggerResourceChange()* wird nur von Prozess 0 aufgerufen, um eine Ressourcenänderung einzuleiten. Dabei wird mithilfe von *MPI_Info_set()* (Zeile 9 und 11) festgelegt, um wie viele Knoten die Ressourcen wachsen oder schrumpfen sollen. Dies geschieht über die Schlüsselwörter *mpi_num_procs_add* für

das Hinzufügen und *mpi_num_procs_sub* für das Entfernen von Prozessen.

Die eigentliche Ressourcenänderung erfolgt durch die Funktion *MPI_Session_dyn_v2a_psetop()* (Zeile 18), die entweder die Operation *MPI_PSETOP_GROW* zum Erhöhen oder *MPI_PSETOP_SHRINK* zum Reduzieren der Prozessanzahl ausführt.

Die beteiligten Variablen sind:

- `local_input_pset`: Die aktuelle PSet-Gruppe, die die aktiven Prozesse enthält.
- `local_output_psets`: Die neue PSet-Gruppe nach der Ressourcenänderung.
- `local_noutput`: Die Größe von *local_output_psets*.

Mit *MPI_Session_set_pset_data()* (Zeile 26) werden die neuen PSet-Daten unter dem Namen des aktualisierten PSets gespeichert und veröffentlicht. Anschließend informiert *notifyResourceChange()* (Zeile 32) alle Prozesse über die durchgeführte Änderung und sendet diesen den Namen des neuen PSets.

Sobald alle Prozesse die Meldung erhalten haben, wird die Funktion *gettingNewPset()* ausgeführt. Diese überprüft, ob eine Neuzuweisung der Arbeit erforderlich ist. Dabei rufen alle Prozesse mithilfe von *MPI_Session_get_pset_info()* (Zeile 7 und 25) und *MPI_Session_get_pset_data()* (Zeile 20) die aktuellen PSet-Informationen ab. Anhand dieser Informationen wird bestimmt, ob ein Prozess terminiert wird. Zudem wird die Kommunikationsgruppe basierend auf dem neuen PSet-Namen neu erstellt (Zeile 31).

Mit dem abschließenden Aufruf von *MPI_Session_dyn_finalize_psetop()* (Zeile 34) wird das alte PSet gelöscht, wodurch die Ressourcenänderung als abgeschlossen gilt.

```
void triggerResourceChange(UTS_Strategy &nodes,int local_op) {
    defineLocalVariables();

    MPI_Info_create(&local_info);

    char* nprocs = itoa(recChange);

    if(local_op == MPI_PSETOP_GROW) {
       MPI_Info_set(local_info, "mpi_num_procs_add", nprocs);
    } else {
       MPI_Info_set(local_info, "mpi_num_procs_sub", nprocs);
    }
    free(nprocs);

    local_input_psets = (char **) malloc(1 * sizeof(char*));
    local_input_psets[0] = strdup(main_pset);
    local_noutput = 0;
    MPI_Session_dyn_v2a_psetop(session, &local_op,
       local_input_psets, 1, &local_output_psets, &local_noutput,
       local_info);

    MPI_Info_free(&local_info);

    if(MPI_PSETOP_NULL != local_op) {
        MPI_Info_create(&local_info);
        MPI_Info_set(local_info, "main_pset", local_output_psets
           [1]);

        MPI_Session_set_pset_data(session, local_output_psets[0],
           local_info);

        MPI_Info_free(&local_info);
    }
    freeArrays();
    op = local_op;
    notifyResourceChange();
}
```

Listing 3.3: Pseudo Code Beispiel für MPI-DPP triggerResourceChange(). Diese Funktion initiiert eine Ressourcenänderung und wird vom Hauptprozess aufgerufen.

```
void gettingNewPset(UTS_Strategy &nodes) {
    MPI_Info info;
    char boolean_string[16];
    int flag;

    bool terminate = false;
    MPI_Session_get_pset_info (session, delta_pset, &info);

    MPI_Info_get(info, "mpi_included", 6, boolean_string, &flag);
    if(0 == strcmp(boolean_string, "True")){
        terminate = true;
    }
    MPI_Info_get(info, "mpi_size", 6, boolean_string, &flag);
    int newSize = atoi(boolean_string);

    MPI_Info_free(&info);

    distributeWorkWhenShrinking();

    MPI_Session_get_pset_data (session, main_pset, delta_pset, (
       char **) &dict_key, 1, true, &info);
    MPI_Info_get(info, "main_pset", MPI_MAX_PSET_NAME_LEN,
       main_pset, &flag);

    MPI_Info_free(&info);

    MPI_Session_get_pset_info (session, main_pset, &info);

    MPI_Info_get(info, "mpi_primary", 6, boolean_string, &flag);
    primaryProcess = (0 == strcmp(boolean_string, "True"));
    MPI_Info_free(&info);

    rebuildNewComm();

    if(primaryProcess){
        MPI_Session_dyn_finalize_psetop(session, main_pset);
    }
    initCommForLifeLines();
}
```

Listing 3.4: Pseudo Code Beispiel für MPI-DPP gettingNewPset(). Diese Funktion wird von allen Prozessen aufgerufen und baut eine neue Kommunikation für das neue PSet auf, sowie eine Datenverteilung falls Prozesse in dem neuen PSet nicht vorhanden sind.

3.3 Evaluation der Programmierer Produktivität

Im folgenden Abschnitt wird die Programmierproduktivität der Implementierungen von Pi und UTS in den Programmiermodellen GLB und MPI-DPP verglichen. Die Analyse basiert auf den Metriken *Lines of Code (LOC)* und *Anzahl der Konstrukte*. Diese Metriken dienen als Indikatoren für den Entwicklungsaufwand, die Komplexität und die Wartbarkeit der Implementierungen.

3.3.1 Lines of Code (LOC)

Lines of Code (LOC) dienen als Indikator für den Entwicklungsaufwand, die Komplexität und die Wartbarkeit einer Anwendung. Eine geringe Anzahl von LOC deutet häufig auf eine effiziente und leichter wartbare Implementierung hin, während eine hohe Anzahl auf eine komplexere und potenziell schwerer wartbare Implementierung hindeuten kann. Dennoch ist die Aussagekraft dieser Metrik begrenzt, da sie stark von individuellen Programmierpraktiken und Stilvariationen abhängt. Nichtsdestotrotz bietet die LOC-Metrik eine grundlegende Orientierung zur Bewertung des Entwicklungsaufwands und Komplexitätsgrads von Anwendungen.

In Tabelle 3.1 ist die Anzahl der LOC für die Implementierungen von Pi und UTS in den Programmiermodellen APGAS-GLB und MPI-DPP dargestellt. Die Implementierung von Pi in MPI-DPP wurde, nicht wie in APGAS-GLB, ohne ein Work-Stealing-Schema entwickelt, bietet jedoch die Möglichkeit zur Anpassung von Ressourcen. Im Gegensatz dazu wurden die UTS-Implementierungen in beiden Programmiermodellen mit einem Work-Stealing-Schema und der Fähigkeit zur Ressourcenanpassung realisiert.

Die Tabelle verdeutlicht, dass die Implementierung von Pi in MPI-DPP mit 262 LOC deutlich mehr Codezeilen erfordert als in APGAS-GLB mit nur 98 LOC. Der Hauptgrund dafür ist, dass sowohl die Ressourcenanpassung als auch die Synchronisationspunkte für die Anpassung in MPI-DPP explizit implementiert werden müssen, während diese Mechanismen in APGAS-GLB bereits vom Programmiermodell bereitgestellt und automatisch ausgeführt werden.

Dasselbe Muster zeigt sich bei der Implementierung von UTS: Hier benötigt die MPI-DPP-Version mit 912 LOC ebenfalls erheblich mehr Code als die APGAS-GLB-Variante mit 262 LOC. Dies liegt daran, dass in MPI-DPP nicht nur die Ressourcenanpassungen, sondern auch das Work-Stealing-Schema manuell implementiert werden müssen, während APGAS-GLB diese Funktionen vollständig abstrahiert

und automatisiert bereitstellt.

Dieser Vergleich verdeutlicht, dass APGAS-GLB den Entwicklungsaufwand erheblich reduziert, indem es zentrale Mechanismen integriert. MPI-DPP hingegen bietet Entwicklern zwar mehr Flexibilität, erfordert aber gleichzeitig eine aufwendigere Implementierung grundlegender Funktionalitäten.

Benchmark	LOC	Einzigartige Konstr.	Insgesamt Konstr.	RÄ-Konstr.
GLB Pi	98	15	22	0
GLB UTS	225	14	19	0
MPI-DPP Pi	262	40	81	13
MPI-DPP UTS	912	50	197	12

Tabelle 3.1: Programmierer Produktivität: Vergleich der Implementierungen von Pi und UTS in GLB und MPI-DPP hinsichtlich *Lines of Code (LOC)*, *Einzigartige Konstrukte*, *Insgesamt Konstrukte* und *Ressourcenänderungskonstrukte*.

3.3.2 Anzahl Konstrukte (Library Calls)

Ein weiterer Indikator für die Programmierproduktivität ist die Anzahl der verwendeten Konstrukte (Library Calls) in einer Anwendung. Diese Metrik gibt Aufschluss darüber, welches Maß an Wissen über spezifische Funktionalitäten ein Entwickler besitzen muss, um eine Anwendung zu implementieren. Eine hohe Anzahl an Konstrukten weist häufig auf eine komplexere und stärker spezialisierte Implementierung hin, während eine geringere Anzahl auf eine allgemeinere schließen lässt.

In Tabelle 3.1 ist die Anzahl der Konstrukte für die Implementierungen von Pi und UTS in den Programmiermodellen APGAS-GLB und MPI-DPP dargestellt. Die Zählung basiert auf den spezifischen Konstrukten der GLB- und MPI-Bibliotheken.

Bei der Implementierung von Pi zeigt sich, dass die Anzahl der Konstrukte in MPI-DPP mehr als doppelt so hoch ist wie in GLB. Dieser Unterschied ist bei der Gesamtanzahl der Konstrukte noch deutlicher, mit fast viermal so vielen in MPI-DPP. Der Grund dafür liegt in den zusätzlichen Konstrukten, die in MPI-DPP für die Implementierung von Ressourcenänderungen und die Synchronisation dieser Anpassungen erforderlich sind.

Die Implementierung von UTS verdeutlicht diesen Trend noch stärker: Die Anzahl der einzigartigen Konstrukte in MPI-DPP ist 3,5-mal höher als in APGAS-GLB. Dies ist darauf zurückzuführen, dass in MPI-DPP sowohl die Ressourcenanpassungen als auch das Work-Stealing-Schema explizit implementiert werden müssen, während

diese Mechanismen in GLB vollständig durch das Programmiermodell abstrahiert und automatisiert bereitgestellt werden. Insgesamt ist die Anzahl der Konstrukte in MPI-DPP nahezu zehnmal so hoch wie in APGAS-GLB.

Diese Ergebnisse machen deutlich, dass die Implementierung von UTS in MPI-DPP eine erheblich spezifischere und komplexere Herangehensweise erfordert. Dies führt zu einem deutlich höheren Aufwand für den Entwickler im Vergleich zu APGAS-GLB, das durch seine abstrahierten und integrierten Mechanismen eine erhebliche Vereinfachung ermöglicht.

3.3.3 Ergebnisse

Die Analyse der LOC (3.3.1) und der Anzahl der Konstrukte (3.3.2) verdeutlicht, dass die Implementierung mit MPI-DPP einen erheblich höheren Entwicklungsaufwand erfordert als mit APGAS-GLB. Darüber hinaus ist der Lernaufwand bei MPI-DPP deutlich höher, da Entwickler ein fundiertes Verständnis sowohl der spezifischen Funktionalitäten und Konstrukte des MPI-Standards als auch der Erweiterung MPI-DPP benötigen.

Im Gegensatz dazu bietet APGAS-GLB eine abstrahierte und integrierte Lösung, die den Entwicklungsprozess erheblich vereinfacht und den Lernaufwand reduziert. Wie in Kapitel 2.3 beschrieben, ermöglicht APGAS-GLB durch seine Abstraktion eine Entwicklung ausschließlich mit Standard-Java, ohne dass sich Entwickler mit spezifischen Konstrukten für Ressourcenanpassungen oder Work-Stealing auseinandersetzen müssen. Dies macht GLB zu einer effizienteren und produktiveren Lösung für die Implementierung dynamischer und adaptiver Anwendungen.

Dennoch bietet MPI-DPP durch seinen expliziten Ansatz eine größere Flexibilität und Kontrolle über die Implementierung, was in bestimmten Anwendungsfällen von Vorteil sein kann. Die Wahl des Programmiermodells hängt daher maßgeblich von den spezifischen Anforderungen und Prioritäten des jeweiligen Projekts ab.

3.4 Experimente

Der folgende Abschnitt beschreibt die Experimente zur Evaluierung der dynamischen Anwendungen. Zunächst wird in Abschnitt 3.4.1 die Experimentierumgebung erläutert, gefolgt von einer detaillierten Beschreibung des Versuchsaufbaus in Abschnitt 3.4.2. Die Ergebnisse zu statischen und dynamischen Ressourcen werden in den

Abschnitten 3.4.3 und 3.4.4 präsentiert.

Darüber hinaus beleuchtet Abschnitt 3.4.5 die Perspektive des Ressourcen-Managers auf die Ressourcenanpassungen, während Abschnitt 3.4.6 die Limitationen der Experimente thematisiert. Abschließend werden in Abschnitt 3.4.7 die zentralen Erkenntnisse zusammengefasst.

3.4.1 Umgebung

Die Experimente wurden auf dem SuperMUC-NG Supercomputer [24] durchgeführt, der vom Leibniz-Rechenzentrum (LRZ) in Garching bei München betrieben wird. Dabei kamen bis zu 16 Knoten zum Einsatz, von denen jeder mit Intel Skylake Xeon Platinum 8174 CPUs, 48 Kernen und 96 GB RAM ausgestattet ist. Die Knoten sind über ein leistungsstarkes InfiniBand-Netzwerk miteinander verbunden.

Für die Experimente wurden die Programmiermodelle APGAS-GLB[2] und MPI-DPP[3] in ihren jeweils aktuellen Versionen verwendet. APGAS-GLB lief unter Java 21[4] und wurde mit einer Konfiguration von einem Prozess pro Knoten und 48 Worker-Threads pro Prozess betrieben. Im Gegensatz dazu startete MPI-DPP 48 Prozesse pro Knoten und nutzte ausschließlich MPI für Kommunikation und Parallelisierung.

3.4.2 Aufbau der Experimente

Für die statischen Ressourcen (siehe Abschnitt 3.4.3) wurden die Benchmarks Pi und UTS mit jeweils 1, 2, 4, 8 und 16 Knoten ausgeführt. Jeder Knoten führte dabei 48 Prozesse aus. Dabei wurde strong scaling verwendet, um die Skalierungseffizienz bei einer konstanten Problemgröße zu untersuchen.

Die Experimente mit dynamischen Ressourcen (siehe Abschnitt 3.4.4) wurden in die Szenarien shrink und grow unterteilt:

- Im shrink-Szenario wurden fünf Kofigurationen ausgeführt. Jede Konfiguration startete mit einer statischen Anzahl von Knoten und führte eine einzige Reduzierung der Knoten durch:
  - Die erste Konfiguration reduzierte die Knotenanzahl von 16 auf 8.
  - Die zweite Konfiguration reduzierte die Knotenanzahl von 8 auf 4.

[2] https://github.com/projectwagomu/lifelineglb.git
[3] https://gitlab.inria.fr/dynres/dyn-procs
[4] https://openjdk.org/projects/jdk/21/

  - Die dritte Konfiguration reduzierte die Knotenanzahl von 4 auf 2.
  - Die vierte Konfiguration reduzierte die Knotenanzahl von 2 auf 1.

- Im grow-Szenario wurde die Knotenanzahl entsprechend erhöht. Auch hier wurden fünf Konfigurationen ausgeführt:
  - Die erste Konfiguration erhöhte die Knotenanzahl von 1 auf 2.
  - Die zweite Konfiguration erhöhte die Knotenanzahl von 2 auf 4.
  - Die dritte Konfiguration erhöhte die Knotenanzahl von 4 auf 8.
  - Die vierte Konfiguration erhöhte die Knotenanzahl von 8 auf 16.

Bei den dynamischen Ressourcen wurde der Fokus auf die Messung der Zeit für die Ressourcenänderung gelegt, um die Reaktionszeit und Effizienz der Programmiermodelle bei Änderungen der Ressourcenanzahl zu bewerten.

Alle Experimente wurden jeweils fünfmal ausgeführt, sowohl für MPI-DPP als auch für APGAS-GLB, unter Verwendung der Benchmarks Pi und UTS. Aus den Ergebnissen wurde der Durchschnitt berechnet.

Keines der verwendeten Programmiermodelle nutzte einen Ressourcen-Manager. Stattdessen führte das Programm nach 10 Sekunden eine Ressourcenänderung durch.

Die Benchmarks wurden mit den folgenden Parametern ausgeführt:

- UTS: Geometrische Baumform, ein Branching-Faktor von 4 und ein Zufalls-Seed von 19.
- Statische Ressourcen:
  - Pi mit einer Iterationsanzahl von 2×10^{11}.
  - UTS mit einer Baumtiefe von 17 und 18.
- Dynamische Ressourcen:
  - Ausschließlich UTS mit einer Baumtiefe von 18.

3.4.3 Statische Ressourcen

Bei den statischen Ressourcen liegt der Fokus auf der Skalierbarkeit und dem Verhalten der Laufzeit bei unterschiedlichen Knotenanzahlen. Die Ergebnisse dieser Experimente sind in den Abbildungen 3.1, 3.2a und 3.2b dargestellt.

Pi: Die Ergebnisse zeigen, dass MPI-DPP bei einem Knoten mit einer Laufzeit von 204 Sekunden schneller ist als APGAS-GLB, das 408 Sekunden benötigt. Dieser Unterschied ist darauf zurückzuführen, dass MPI-DPP in C++ implementiert ist, während APGAS-GLB auf Java basiert. C++ ist bekanntlich performanter als Java, insbesondere bei rechenintensiven Anwendungen [25]. Desweiteren verwendet APGAS-GLB bei einem Knoten ein Work-Sharing-Schema, während MPI-DPP ohne dieses auskommt.

Trotzdem skaliert APGAS-GLB deutlich besser als MPI-DPP. Bei 16 Knoten erreicht APGAS-GLB einen nahezu linearen Speed-Up von 15,5, während MPI-DPP lediglich einen Speed-Up von 14,0 erzielt. Dieser Unterschied lässt sich auf das effiziente Work-Stealing-Schema von APGAS-GLB zurückführen, das eine gleichmäßige Verteilung der Arbeit ermöglicht und dadurch die Skalierung verbessert.

UTS: Die Ergebnisse bei UTS folgen einem ähnlichen Muster wie bei Pi. Auch hier zeigt sich bei einem Knoten ein deutlicher Performance-Unterschied zwischen Java und C++. Hierbei verwenden beide Programmiermodelle ein Work-Stealing-Schema.

Bei einer Baumtiefe von 17 zeigt APGAS-GLB eine deutlich bessere Skalierung mit einem Speed-Up von 14,1, im Vergleich zu einem Speed-Up von 8,9 bei MPI-DPP. Dieser Vorteil ist auf das effizientere Work-Stealing-Schema von APGAS-GLB zurückzuführen.

Bei einer Baumtiefe von 18 verringert sich der Unterschied im Speed-Up zwischen den beiden Programmiermodellen. Dies ist auf die höhere Arbeitslast zurückzuführen, die die Skalierung beeinflusst. APGAS-GLB erreicht hier einen Speed-Up von 13,5, während MPI-DPP einen Speed-Up von 12,5 erzielt.

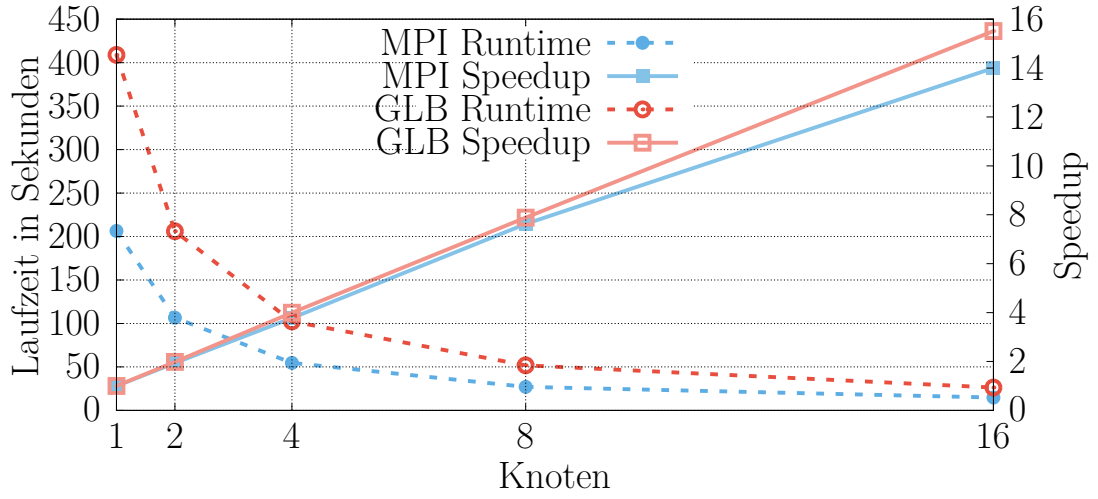
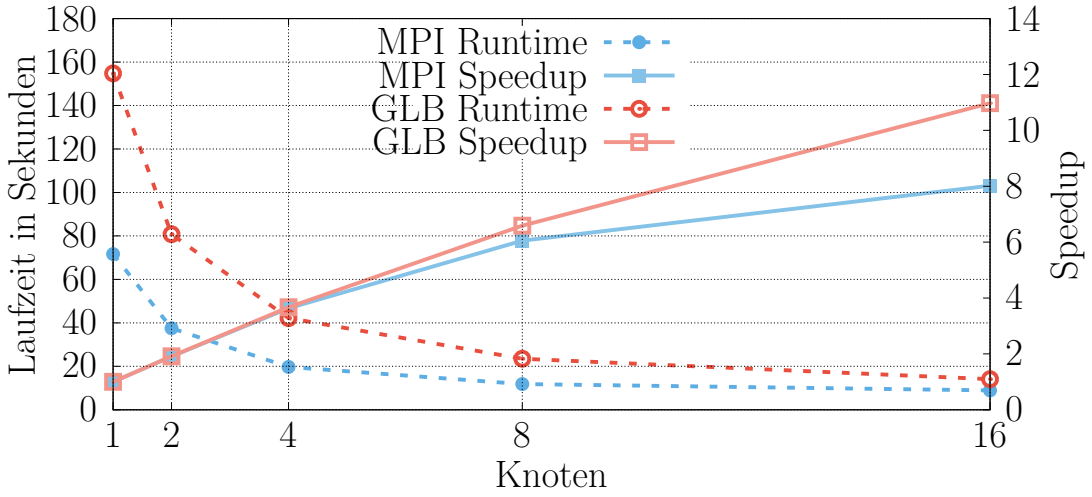
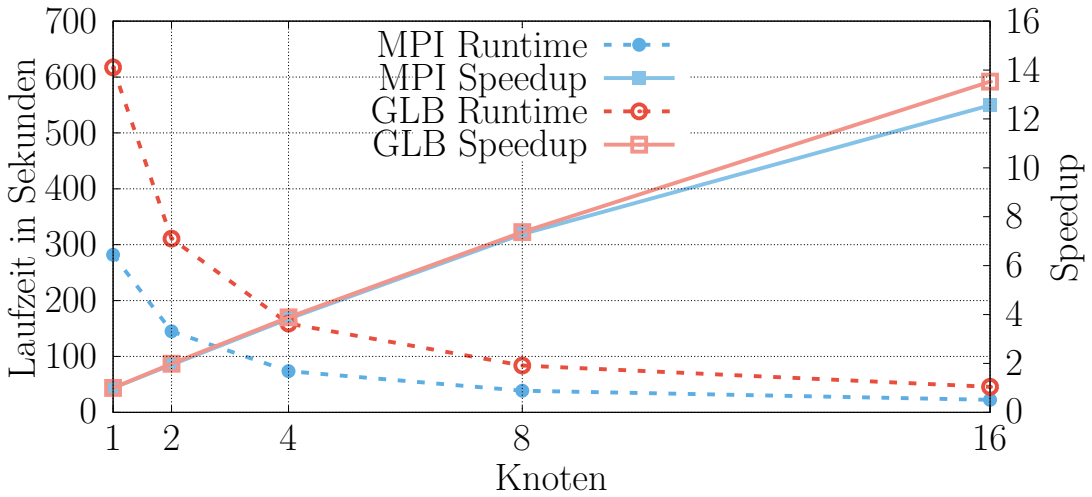


Abbildung 3.1: Statische Ressourcenkonfigurationen beim Strong Scaling für Pi mit einer Genauigkeit von 2×10^{11}. Die Problemgröße bleibt konstant, während die Anzahl der Knoten von 1 auf 16 skaliert wird. Die linke Y-Achse stellt die Laufzeit dar, während die rechte Y-Achse den Speedup relativ zur Ausführung mit einem einzelnen Knoten zeigt.



(a) UTS mit einer Tiefe von 17



(b) UTS mit einer Tiefe von 18

Abbildung 3.2: Statische Ressourcenkonfigurationen beim Strong Scaling für UTS mit Baumtiefen von 17 und 18. Die Problemgröße bleibt konstant, während die Anzahl der Knoten von 1 auf 16 skaliert wird. Die linke Y-Achse zeigt die Laufzeit, die rechte Y-Achse den Speedup relativ zur Ausführung auf einem einzelnen Knoten.

3.4.4 Dynamische Ressourcen

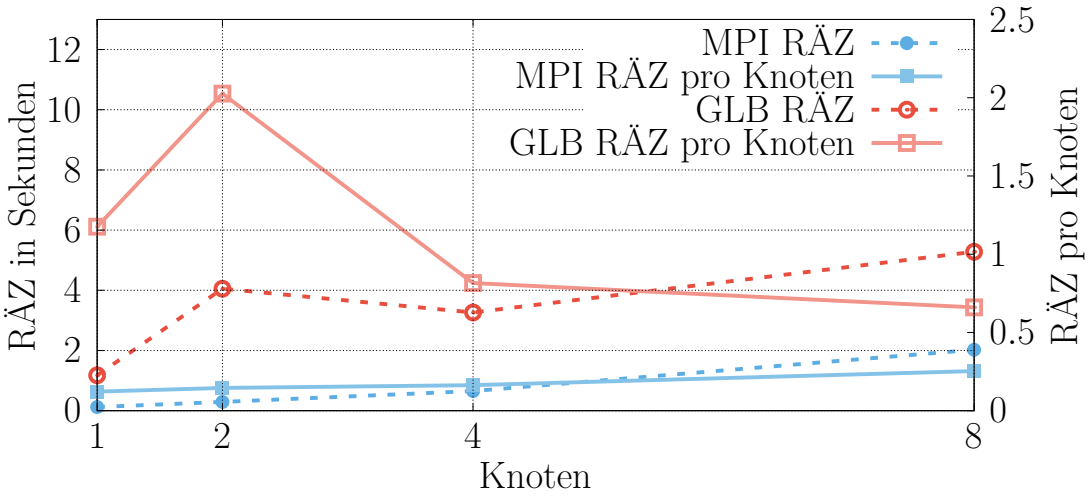
Bei den dynamischen Ressourcen liegt der Fokus auf der Reaktionszeit und der Effizienz der Programmiermodelle bei Änderungen der Ressourcenanzahl. Die Ergebnisse dieser Experimente sind in den Abbildungen 3.3a und 3.3b dargestellt und wurden aus Anwendungssicht gemessen. Diese Abbildungen zeigen die Ressourcenänderungzeiten, die für die Anpassung der Ressourcen in den Szenarien *shrink* und *grow* bei unterschiedlichen Knotenanzahlen benötigt wurde.

Shrink-Szenario: Die Ergebnisse zeigen, dass MPI-DPP Ressourcenänderungen bei einer Reduzierung der Knotenanzahl schneller durchführt als APGAS-GLB. Zwar nimmt die Zeit für die Ressourcenanpassung bei beiden Programmiermodelle mit steigender Knotenanzahl zu, jedoch bleibt sie bei MPI-DPP insgesamt geringer. Auffällig ist zudem ein Ausreißer bei APGAS-GLB: Die Anpassung von 4 auf 2 Knoten dauert länger als die von 8 auf 4 Knoten.

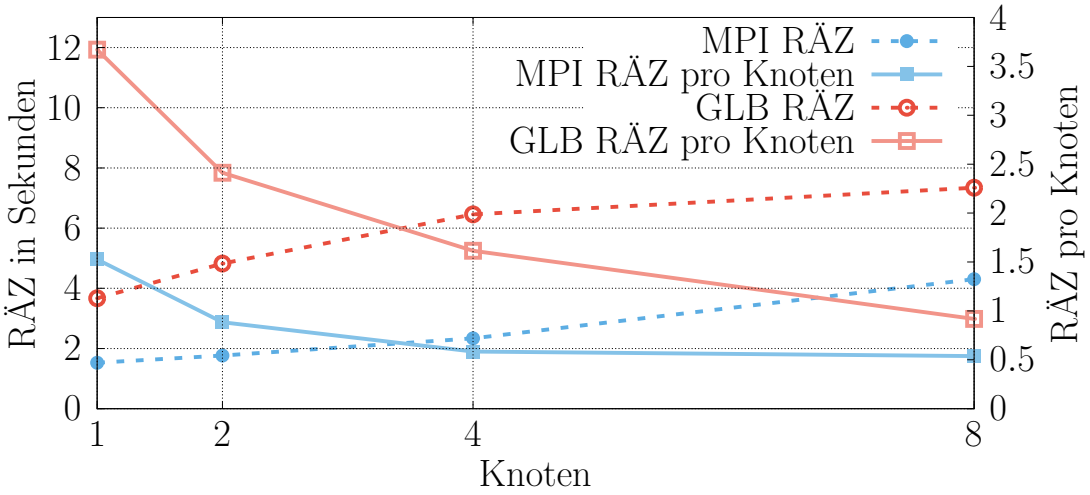
Die durchschnittliche Zeit für eine Ressourcenänderung von 16 auf 8 Knoten beträgt bei MPI-DPP 2 Sekunden, während APGAS-GLB 5,2 Sekunden benötigt. Ein wesentlicher Unterschied liegt in der Anzahl der zu terminierenden Prozesse: APGAS-GLB beendet lediglich 8 Prozesse, während MPI-DPP 8×48 Prozesse terminiert. Dieser zusätzliche Aufwand verdeutlicht den Unterschied in der Reaktionszeit noch stärker.

Grow-Szenario: Ein ähnliches Muster zeigt sich auch bei der Erhöhung der Knotenanzahl: MPI-DPP passt die Ressourcen schneller an als APGAS-GLB. Zwar nimmt die Dauer der Ressourcenänderung bei beiden Programmiermodelle mit steigender Knotenanzahl zu, doch erweist sich MPI-DPP insgesamt als effizienter.

Die durchschnittliche Zeit für eine Ressourcenänderung von 8 auf 16 Knoten beträgt bei MPI-DPP 4,3 Sekunden, wobei 8×48 Prozesse hinzugefügt werden. Im Vergleich dazu benötigt APGAS-GLB 7,3 Sekunden, um 8 Prozesse hinzuzufügen. Auch hier wird deutlich, dass MPI-DPP effizienter bei Ressourcenänderungen ist als APGAS-GLB.



(a) Shrinking: die Initialanzahl von 2 zu 16 Knoten wird halbiert.



(b) Growing: die Initialanzahl von 1 zu 8 Knoten wird verdoppelt.

Abbildung 3.3: Dynamische Ressourcenkonfigurationen für UTS (Tiefe 18). Im Szenario *Shrinking* wird die Knotenanzahl halbiert, während sie im *Growing*-Szenario verdoppelt wird. Bei GLB wird ein multithreaded Prozess pro Knoten gestartet/gestoppt, bei MPI-DPP 48 Prozesse pro Knoten. Die linke Y-Achse zeigt die gesamte Ressourcenänderungzeit (*RÄZ*), die rechte die *Ressourcenänderungzeit pro Knoten*.

3.4.5 Sicht des Ressourcen-Manager

Zusätzlich zu den Experimenten wurde bei MPI-DPP und APGAS-GLB die Zeit gemessen, die für die Initialisierung neuer Prozesse sowie für die Freigabe bestehender Prozesse benötigt wurde, sodass der Supercomputer diese erneut nutzen konnte. Die Messergebnisse sind in Tabelle 3.2 dokumentiert und zeigen die Dauer der Ressourcenänderung – vom Start der Anpassung bis zur vollständigen Initialisierung neuer Prozesse bzw. zur Freigabe bestehender Prozesse.

Diese Messungen liefern wertvolle Erkenntnisse darüber, wie lange der Ressourcen-Manager warten muss, bis die Knoten wieder voll einsatzbereit sind, und wie schnell Prozesse entfernt und anderweitig verwendet werden können.

Auffällig ist, dass MPI-DPP Ressourcenänderungen schneller durchführt als APGAS-GLB. Betrachtet man die Skalierung von 8 auf 16 bzw. von 16 auf 8 Knoten, zeigt sich eine Zeitdifferenz von 4,25 Sekunden beim Shrinking und 3,15 Sekunden beim Growing zwischen den beiden Programmiermodelle – mit einer steigenden Tendenz bei wachsender Knotenanzahl.

Ein weiteres Ergebnis ist, dass die Shrinking-Zeiten bei MPI-DPP schneller sind als in Abbildung 3.3a dargestellt. Der Grund dafür ist, dass Abbildung 3.3a die Anwendungssicht zeigt, bei der zusätzlich die Kommunikation neu aufgebaut werden muss, was die Anpassungszeit verlängert. Beim Growing hingegen sind die Unterschiede gering, da der Kommunikationsaufbau bereits Teil der Initialisierung neuer Prozesse ist.

Für APGAS-GLB bleiben die Werte unverändert, da sich die Anwendungssicht und die Ressourcen-Manager Sicht nicht unterscheiden. Dies ist auf das *MalleableHandler*-Interface zurückzuführen, das den Ressourcen-Manager direkt über Änderungen informiert.

Benchmark	1 Knoten	2 Knoten	4 Knoten	8 Knoten
APGAS-GLB UTS Shrink	1,17	4,05	3,26	5,28
APGAS-GLB UTS Grow	3,67	4,82	6,45	7,34
MPI-DPP UTS Shrink	0,19	0,19	0,56	1,03
MPI-DPP UTS Grow	1,44	1,69	2,21	4,19

Tabelle 3.2: Ressourcenänderungen: Durchschnittliche Zeit für Ressourcenänderungen aus der Sicht des Ressourcen-Managers in den Szenarien shrink und grow bei unterschiedlichen Knotenanzahlen. Die Shrink-Zeit beschreibt die Zeit, vom trigger Zeitpunkt bis zur Freigabe der Prozesse, sodass der Supercomputer sie wieder verwenden kann. Bei Grow wird die Zeit vom trigger Zeitpunkt bis zur vollständigen Initialisierung der neuen Prozesse gemessen.

3.4.6 Limitationen

Die Evaluierung der Programmiererproduktivität sowie die Experimente zu statischen und dynamischen Ressourcen weisen einige Limitationen auf. Für genauere Ergebnisse könnten z.B. zusätzliche Metriken, wie Speicherverbrauch und die Lernkurve der Entwickler einbezogen werden.

Hinsichtlich der Skalierung der Tests ergeben sich ebenfalls Einschränkungen: Die Experimente zu statischen Ressourcen wurden auf maximal 16 Knoten begrenzt, während bei dynamischen Ressourcen lediglich Tests mit maximal 8 zu 16 Knoten durchgeführt wurden. Darüber hinaus wurde pro Programm nur ein einzelner Ressourcenwechsel durchgeführt. Realitätsnähere Szenarien könnten durch mehrere Ressourcenwechsel abgebildet werden, was zusätzliche Erkenntnisse liefern würde.

Zudem ist DPP-MPI derzeit auf die Hinzufügung oder Entfernung ganzer Knoten beschränkt, wobei die Prozessanzahl auf jedem Knoten gleich sein muss. In der Praxis könnte dies jedoch zu Einschränkungen führen, da es oft nicht möglich ist, die Prozessanzahl auf allen Knoten konstant zu halten.

3.4.7 Ergebnisse

Für die oben durchgeführten Experimente wurden die Benchmarks Pi und UTS verwendet. Die Ergebnisse zeigen, dass APGAS-GLB bei statischen Ressourcen eine bessere Skalierbarkeit als MPI-DPP aufweist. Allerdings erweist sich MPI-DPP bei dynamischen Ressourcenänderungen als effizienter und erzielt insgesamt eine bessere Performance als APGAS-GLB.

Ein Vorteil von APGAS-GLB liegt in der geringeren Anzahl an Codezeilen (LOC) und Konstrukten im Vergleich zu MPI-DPP, was auf eine höhere Entwicklerproduktivität und eine bessere Wartbarkeit des Codes hinweist. MPI-DPP hingegen bietet mehr Flexibilität und Kontrolle über die Implementierung, was in bestimmten Anwendungsfällen vorteilhaft sein kann.

Eine fairere Vergleichsbasis wäre gegeben, wenn statt der aktuellen Konfiguration von MPI-DPP – mit $48 \times$ Anzahl der Knoten – nur ein Prozess pro Knoten gestartet würde, der dann mithilfe von MPI + OpenMP[5] [26] Multithreading nutzt. Dies würde den Vergleich gerechter machen, da die größere Anzahl an gestarteten Prozessen bei MPI-DPP einen höheren Overhead bei Ressourcenänderungen verursacht als APGAS-GLB.

Trotz dieser Unterschiede zeigen die Ergebnisse, dass APGAS-GLB eine äußerst effiziente Lösung für dynamische und adaptive Anwendungen darstellt. MPI-DPP überzeugt hingegen durch seine hohe Effizienz bei Ressourcenänderungen sowie seine insgesamt bessere Leistung. Eine Kombination beider Programmiermodelle – die Einfachheit von APGAS-GLB mit der Effizienz von MPI-DPP – könnte eine Lösung für die Entwicklung dynamischer und adaptiver Anwendungen darstellen.

[5] Bibliothek für abstrahiertes Multithreading in C++

4 Verwandte Arbeiten

Dynamisches Ressourcenmanagement für Supercomputer wird seit über 20 Jahren intensiv untersucht. Erste Ansätze basierten auf Checkpointing-Verfahren, bei denen der aktuelle Programmzustand gespeichert, das Programm beendet und anschließend mit neuen Ressourcen wieder gestartet wurde [27, 28, 29]. Allerdings erwies sich dieser Ansatz als ineffizient, da er durch den Overhead des Speicherns und Ladens belastet wird und nicht von allen dynamischen Ressourcenmodellen unterstützt wird.

Als Alternative wurden zahlreiche Konzepte entwickelt, darunter ReShape [5], FlexMPI [6], Elastic MPI [7], DMR API [8], DMRlib [9] und MPR [10]. Diese Ansätze verfolgen das Ziel, Ressourcenänderungen in Kombination mit MPI zu ermöglichen und im Supercomputer-Bereich zu etablieren. Sie umgehen die Notwendigkeit von Checkpoints und Programmneustarts, indem sie den globalen MPI-Kommunikationsraum dynamisch anpassen. Allerdings gestaltet sich dies aufgrund der statischen Natur des MPI-Weltmodells als schwierig, da dynamische Ressourcenänderungen nicht einfach durch den Entwickler erzwungen werden können [30, 31].

Zudem fehlt den genannten Ansätzen eine einheitliche, leicht zugängliche API, die Entwicklern eine einfache Nutzung dynamischer Ressourcen ermöglichen würde. Stattdessen sind viele dieser Lösungen speziell für bestimmte Anwendungen entwickelt worden und nicht allgemein einsetzbar, was ihre praktische Verwendbarkeit einschränkt.

Mit der Einführung von MPI 4.0 [4] im Juni 2021 wurde das MPI-Sessions-Modell [30] vorgestellt, das durch die Verbesserungen des PMIx-Standards [32] neue Möglichkeiten für ein flexibles und vielseitig einsetzbares dynamisches Ressourcenmanagement eröffnet. Aufbauend auf diesen Fortschritten wurden die Designprinzipien von DPP [11] entwickelt, aus denen schließlich MPI-DPP hervorging – mit dem Ziel, dynamische Ressourcenanpassungen in parallelen Programmiermodellen wie MPI zu ermöglichen und Entwicklern eine benutzerfreundliche Schnittstelle bereitzustellen.

Neben MPI existieren auch alternative parallele Programmiermodelle wie

Charm++/AMPI [33, 34] und APGAS-GLB [3, 35, 15], die ebenfalls dynamische Ressourcenanpassungen unterstützen – jedoch durch eine zusätzliche Abstraktionsebene. Diese Modelle wurden entwickelt, um die Produktivität der Entwickler zu steigern und die Anwendungsentwicklung zu vereinfachen, indem sie viele Details des dynamischen Ressourcenmanagements abstrahieren. Trotz dieser Vorteile sind sie jedoch noch nicht so weit verbreitet wie MPI und haben bislang nicht dieselbe Akzeptanz in der Supercomputer-Welt erreicht.

5 Fazit

In dieser Arbeit wurden zwei Ansätze für das dynamische Ressourcenmanagement untersucht: MPI-DPP und APGAS-GLB. Zur Evaluation wurden die Benchmarks Pi und UTS implementiert und getestet. Während die Implementierung in C++ mit MPI-DPP erfolgte, wurde Java für APGAS-GLB verwendet. Die Bewertung der Programmiererproduktivität erfolgte anhand der Anzahl der Codezeilen (LOC) und der Anzahl der Konstrukte (Library Calls). Zudem wurden die Experimente auf dem Supercomputer SuperMUC-NG sowohl mit statischen als auch mit dynamischen Ressourcen durchgeführt. Neben der Skalierbarkeit der Benchmarks wurde auch die Laufzeit der Ressourcenanpassung ohne dedizierten Ressourcen-Manager untersucht.

Die Ergebnisse zeigen, dass APGAS-GLB insgesamt weniger Codezeilen und Konstrukte benötigt als MPI-DPP. Beide Ansätze bieten eine gute Skalierbarkeit, wobei APGAS-GLB eine bessere Skalierung erreicht als MPI-DPP. Allerdings ist die Laufzeit der Ressourcenanpassung in MPI-DPP effizienter als in APGAS-GLB. Ein wesentlicher Vorteil von APGAS-GLB liegt in seinem geringeren Entwicklungsaufwand, da es das dynamische Ressourcenmanagement, sowie eine dynamische Lastverteilung abstrahiert und eine einfache API bereitstellt, die Entwicklern eine intuitive Nutzung ermöglicht. Im Gegensatz dazu erfordert MPI-DPP mehr manuellen Aufwand, da Entwickler sich mit den internen Details des Ressourcenmanagements auseinandersetzen müssen. Dies bietet jedoch auch größere Flexibilität und Kontrolle über das System.

Für zukünftige Arbeiten wäre es interessant, MPI-DPP in Kombination mit OpenMP zu untersuchen, um die Skalierbarkeit der Benchmarks weiter zu verbessern und Multi-Processing mit Multi-Threading zu verbinden. Dies könnte nicht nur die Laufzeit der Ressourcenanpassung optimieren, sondern auch neue Möglichkeiten für eine hybride Implementierung schaffen. Eine Kombination beider Programmiermodelle – die Einfachheit von APGAS-GLB mit der Effizienz von MPI-DPP – könnte eine Lösung darstellen, bei der je nach spezifischem Problem entweder MPI-DPP oder dann MPI-GLB gezielt eingesetzt werden kann.

Literaturverzeichnis

[1] Slurm Forum. *Slurm: Workload Manager*. 2025. URL: https://slurm.schedmd.com/documentation.html.

[2] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock und Devesh Tiwari. „Job characteristics on large-scale systems: long-term analysis, quantification, and implications“. In: *Supercomputing (SC)*. ACM, 2020. DOI: 10.1109/SC41405.2020.00088.

[3] Patrick Finnerty, Jonas Posner, Janek Bürger, Leo Takaoka und Takuma Kanzaki. „On the Performance of Malleable APGAS Programs and Batch Job Schedulers“. In: *Springer Nature Computer Science* (2024). DOI: 10.1007/s42979-024-02641-7.

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[5] Rajesh Sudarsana und Calvin J.Ribbens. „ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment“. In: *International Conference on Parallel Processing (ICPP)*. 2007. DOI: 10.1109/ICPP.2007.73.

[6] Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh und Jesús Carretero. „FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems“. In: *Euro-Par*. Springer, 2013. DOI: 10.1007/978-3-642-40047-6_16.

[7] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt und Hans-Joachim Bungartz. „Infrastructure and API Extensions for Elastic Execution of MPI Applications“. In: *EuroMPI*. ACM, 2016. DOI: 10.1145/2966884.2966917.

[8] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran und Antonio Journal Peña. „DMR API: Improving cluster productivity by turning applications into malleable“. In: *Parallel Computing* (2018). DOI: 10.1016/j.parco.2018.07.006.

[9] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí und Antonio Journal Peña. „DMRlib: Easy-Coding and Efficient Resource Management for Job Malleability“. In: *Transactions on Computers (TC)* (2020). DOI: 10.1109/TC.2020.3022933.

[10] Júnior Löff, Dalvan Griebler, Luiz Gustavo Fernandes und Walter Binder. „MPR: An MPI Framework for Distributed Self-adaptive Stream Processing“. In: *Euro-Par*. Springer, 2024. DOI: 10.1007/978-3-031-69583-4_28.

[11] Dominik Huber, Martin Schreiber, Martin Schulz, Howard Pritchard und Daniel Holmes. *Design Principles of Dynamic Resource Management for HPC Parallel Programming Models*. 2024. URL: https://arxiv.org/abs/2403.17107.

[12] Dominik Huber. *DPP demonstrator implementation*. 2023. URL: https://gitlab.inria.fr/dynres/dyn-procs.

[13] Olivier Tardieu. „The APGAS Library: Resilient Parallel and Distributed Programming in Java 8“. In: *SIGPLAN Workshop on X10*. ACM, 2015. DOI: 10.1145/2771774.2771780.

[14] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat und Mikio Takeuchi. „GLB: Lifeline-based Global Load Balancing Library in X10“. In: *Workshop on Parallel Programming for Analytics Applications (PPAA)*. ACM, 2014. DOI: 10.1145/2567634.2567639.

[15] Jonas Posner, Raoul Goebel und Patrick Finnerty. „Evolving APGAS Programs: Automatic and Transparent Resource Adjustments at Runtime“. In: *Workshop on Asynchronous Many-Task Systems and Applications (WAMTA)*. 2024. DOI: 10.1007/978-3-031-61763-8_15.

[16] Horng-Jyh Wu Frank Harary John P. Hayes. „A survey of the theory of hypercube graphs“. In: *Computers & Mathematics with Applications (Vol 15)* (2002). DOI: 10.1016/0898-1221(88)90213-1.

[17] Open-MPI Forum. *Open MPI v5.0.x*. 2025. URL: https://docs.open-mpi.org/en/v5.0.x/index.html.

[18] OpenPMIx Forum. *OpenPMIx*. 2025. URL: https://docs.openpmix.org/en/latest/index.html#.

[19] PMIx Administrative Steering Committee. *Process Management Interface for Exascale (PMIx) Standard 4.0*. 2020. URL: https://pmix.github.io/uploads/2020/12/pmix-standard-v4.0.pdf.

[20] PRRTE Forum. *PMIx Reference Runtime Environment (PRRTE)*. 2025. URL: https://docs.prrte.org/en/latest/.

[21] *Monte-Carlo-Simulation*. 2025. URL: https://de.wikipedia.org/wiki/Monte-Carlo-Simulation.

[22] *Embarrassingly parallel*. 2025. URL: https://en.wikipedia.org/wiki/Embarrassingly_parallel.

[23] Stephen Olivier, Journaln Huan, Journalnze Liu, Journaln Prins, Journalmes Dinan, P. Sadayappan und Chau-Wen Tseng. „UTS: An Unbalanced Tree Search Benchmark“. In: *Languages and Compilers for Parallel Computing*. Springer, 2006. DOI: 10.1007/978-3-540-72521-3_18.

[24] *SuperMUC-NG*. 2025. URL: https://doku.lrz.de/supermuc-ng-10745965.html.

[25] Joel S. Emer Bradley C. Kuszmaul Butler W. Lampson Daniel Sanchez Tao B. Schardl Charles E. Leiserson Neil C. Thompson*. „There's plenty of room at the Top: What will drive computer performance after Moore's law?“ In: *COMPUTER SCIENCE* (2020). DOI: 10.1126/science.aam9744.

[26] Open-MP. *OpenMP: The OpenMP API specification for parallel programming*. 2025. URL: https://www.openmp.org/.

[27] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski und Carlos Varela. „Dynamic Malleability in Iterative MPI Applications“. In: *International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, 2007. DOI: 10.1109/CCGRID.2007.45.

[28] Adam Moody, Greg Bronevetsky, Kathryn Mohror und Bronis R. de Supinski. „Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System“. In: *Supercomputing (SC)*. ACM, 2010. DOI: 10.1109/SC.2010.18.

[29] Sathish S. Vadhiyar und Jack J. Dongarra. „SRS—A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems“. In: *Parallel Processing Letters* (2003). DOI: 10.1142/S0129626403001288.

[30] Daniel Holmes, Kathryn Mohror, Ryan E. Grant, Anthony Skjellum, Martin Schulz, Wesley Bland und Jeffrey M. Squyres. „MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale“. In: *EuroMPI*. ACM, 2016. DOI: 10.1145/2966884.2966915.

[31] Atsushi Hori, Emmanuel Jeannot, George Bosilca, Takahiro Ogura, Balazs Gerofi, Jie Yin und Yutaka Ishikawa. „An international survey on MPI users“. In: *Parallel Computing* (2021). DOI: 10.1016/j.parco.2021.102853.

[32] Ralph H. Castain, Joshua Hursey, Aurelien Bouteiller und David Solt. „PMIx: Process management for exascale environments“. In: *Parallel Computing* (2018). DOI: 10.1016/j.parco.2018.08.002.

[33] Chao Huang, Orion Lawlor und L. V. Kalé. „Adaptive MPI“. In: *Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2004. DOI: 10.1007/978-3-540-24644-2_20.

[34] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta und Laxmikant V. Kale. „A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications“. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2015. DOI: 10.1109/IPDPS.2015.34.

[35] Jonas Posner und Claudia Fohry. „Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing“. In: *International Conference on Parallel Processing (ICPP) Workshop (P2S2)*. ACM, 2021. DOI: 10.1145/3458744.3473361.