Freie Universität Berlin

Bachelorarbeit

Entwurf und Implementierung einer Prozesszuordnungsstrategie für MPI-Prozesse in planungsbasierten Cluster-Verwaltungssystemen

von Felix Korthals
Matr.-Nr. 4661639, 6. FS

Gutachter:
Barry Linnert
Prof. Dr.-Ing. Jochen Schiller

betreut von:
Barry Linnert

22. September 2021

Inhaltsverzeichnis

1 Einleitung **3**

2 Scheduling in HPC-Systemen **8**

2.1 Warteschlangen-basiertes Scheduling 8

2.2 Plan-Based-Scheduling 10

3 Problemlage **12**

4 MPI **13**

4.1 Gruppen und Kommunikatoren 14

4.1.1 MPI Groups 14

4.1.2 MPI Communicators 15

4.2 Prozessverwaltung in MPI 15

4.2.1 MPI Prozess Initialisierung 15

4.2.2 MPI Dynamic Process Model 16

4.2.3 Prozesse in MPI 17

5 OpenMPI **18**

5.1 Programmstart 18

5.2 groups und communicators 21

5.3 Communicator-Initialisierung 23

6 Lösung **26**

6.1 Ansatz 26

6.2 Implementierung 27

6.3 Testaufbau 28

6.4 Identifikation des Knotens 29

7 Auswertung **31**

7.1 Randbedingungen 32

8 Fazit **34**

9 Ausblick **34**

Literatur **36**

Abbildungsverzeichnis **37**

1 Einleitung

Die aktuell leistungsstärksten x86-Prozessoren können bis zu 64 CPU-Cores besitzen (Beispiel: "AMD EPYC 7763" [1]). Dennoch gibt es Anwendungen, die auch auf Rechnern, die mit solchen Prozessoren und entsprechend großzügigen anderen Ressourcen, wie z. B. Speicher, etc. ausgestattet sind, eine Laufzeit haben, die für den Anwendungsfall inakzeptabel lang ist. Reichen die Rechenressourcen eines einzelnen Rechners nicht mehr aus, müssen mehrere Rechner zusammengeschlossen werden, um gemeinsam das gestellte Problem in einer geringeren Zeit zu lösen, unter der Voraussetzung, dass sich das Problem auch entsprechend parallelisieren lässt. Ein solcher Rechnerverbund bzw. verteiltes System, zum Ausführen paralleler Anwendungen, wird als High Performance Computing System (HPC-System) bezeichnet. Ein HPC-System kann dabei sowohl homogen, bestehend aus Rechnern mit identischen Komponenten und Eigenschaften (z. B. Typ und Anzahl der CPUs, des Speichers, etc.), als auch heterogen, bestehend aus unterschiedlichen Rechnern, sein. Die leistungsstärksten aktuellen HPC-Systeme haben mehrere Millionen Prozessoren und mehrere Petabyte Hauptspeicher (z. B. der aktuelle Platz 1 der Top500, der Rangliste der leistungsstärksten Supercomputer: "Supercomputer Fugaku" mit 7.630.848 Prozessoren und 5.087.232 GB Hauptspeicher [18]). Aber auch für ein solches System finden sich Anwendungen, die eine Laufzeit von mehreren Tagen, Wochen oder gar Monaten aufweisen.

Abbildung 1: Beispieldarstellung für ein HPC-System: "Summit" [11]

Es gibt viele Bereiche, sowohl in der Wissenschaft, als auch mittlerweile in der Wirtschaft, die HPC-Systeme nutzen und auslasten. Insbesondere die Simulation physikalischer Modelle ist oft ein Anwendungsgegenstand (wie beispielsweise im Forschungsbereich der Hochenergiephysik am CERN (*Conseil européen pour la recherche nucléaire*), wo HPC-Systeme unter anderem zur Simulation der Detektoren genutzt werden [3, S. 15, Abs. 3]), da sich viele reale Probleme nicht analytisch lösen lassen, im Gegensatz z. B. zum bekannten "Zweikörperproblem" (die Simulation der Bewegung zweier Körper, die miteinander wechselwirken, z.B. die Gravitation zwischen Erde und Sonne). Bereits die Simulation der Wechselwirkungen dreier, oder mehr, Körper miteinander, was analog als Dreikörperproblem bzw. allgemeiner als Mehrkörperproblem bezeichnet wird, kann bislang nur numerisch, also mit Hilfe bestimmter Approximationsalgorithmen, gelöst bzw. genähert werden.
Zur Veranschaulichung, wie man ein solches Problem mit einem HPC-System lösen könnte, wird im Folgenden ein naiver Beispielalgorithmus für eine parallele Berechnung des Mehrkörperproblems beschrieben: Man nimmt eine Startkonfiguration von einer bestimmten Menge von, zur Vereinfachung, Punktmassen an, jeweils bestehend aus ihrer Masse und ihren Koordinaten im Raum. Dann weist man jedem zugehörigen Prozess auf den Prozessoren der am HPC-System teilnehmenden Computer, welche als Knoten bezeichnet werden, eine bestimmte Menge der Massen in der Startkonfiguration zu und jeder Prozess berechnet die Gravitation aller anderen Massen auf die jeweilig betrachtete Punktmasse und die daraus resultierende Kraft, die auf die Punktmasse einwirkt und die Beschleunigung, die daraus resultiert. Die resultierenden Bewegungen aller Punktmassen werden schlussendlich an alle teilnehmenden Prozesse übermittelt und anhand des Ergebnisses der nächste Schritt berechnet.
Der Unterschied dazwischen, wie man ein solches Problem mit einem HPC-System, im Gegensatz zu einem einzelnen Rechner, lösen würden, liegt vor allem in der notwendigen Kommunikation, z. B. der Zwischenergebnisse jedes Iterationsschrittes, an die Prozesse auf den verschiedenen Knoten des HPC-Systems. In welcher Form diese Kommunikation konkret abläuft, bedingt allerdings das durchgeführte Programm, welches auch speziell für die parallele Ausführung auf einem HPC-System entwickelt sein muss: Anstatt der regelmäßigen Kommunikation von Zwischenschritten, kann ein Programm kann auch voneinander unabhängige Prozesse erschaffen, dessen Ergebnisse erst zum Schluss zusammengeführt werden, wie es beispielsweise bei einer Monte-Carlo-Simulation der Fall wäre. Damit hängt es auch direkt vom Programm ab, wie groß der Kommunikationsaufwand im HPC-System zur Laufzeit sein wird.
Ein einzelner Rechner kann zwar auch komplexe Probleme lösen, ist aber in seiner Leistungsfähigkeit beschränkt, da er nicht beliebig erweiterbar ist, woraus lange Laufzeiten für die Lösung groß dimensionierter Probleme resultieren können. Möchte man größere Probleme lösen und dennoch eine hinreichend kurze Laufzeit erreichen (vorausgesetzt man hat die Ausbaukapazitä-

ten eines einzelnen Rechners ausgeschöpft), muss man mehrere Rechner mit einem Netzwerk verbinden, um das Problem, wenn möglich, auf die Rechner zu verteilen und damit die Laufzeit zu verkürzen. Die Kommunikation zwischen verschiedenen physischen Rechnern über ein konventionelles Netzwerk bietet aber natürlich signifikant geringere Übertragungsraten und deutlich höhere Latenzen, als die interne Kommunikation auf einem einzelnen Rechner. In einem HPC-System werden die Knoten daher üblicherweise mit einem besonders leistungsfähigen Netzwerk miteinander verbunden, welches einen hohen Datendurchsatz und geringe Kommunikationslatenzen realisieren soll. Damit sind die Latenzen und Übertragungsraten zwischen den Knoten zwar dennoch nicht auf dem gleichen Niveau wie innerhalb eines einzelnen Knotens, aber im Gegenzug ist die Leistungsfähigkeit des Gesamtsystems vielfach höher, weshalb mit einem HPC-System Probleme gelöst werden können, deren Größe (z. B. die Auflösung einer Simulation) vielfach über dem liegt, was ein einzelner Rechner, in hinreichend kurzer Zeit, berechnen könnte. Dazu kommt, dass, abhängig vom Programm, der größere Kommunikationsaufwand mit wachsender Problemgröße meist eine geringere Rolle spielt und damit die Effizienz steigt (siehe "Gustafson's Law" [9, Abs. 4]).

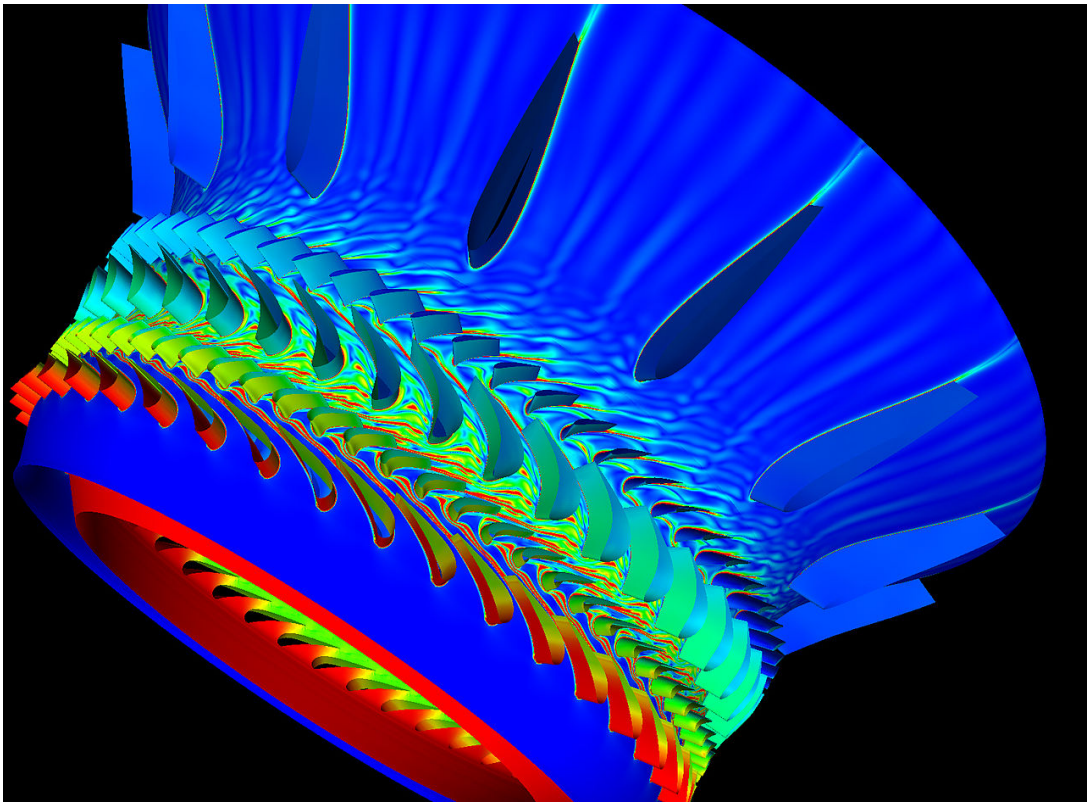

Abbildung 2: Darstellung der Simulation einer Turbine mit nach statischem Druck eingefärbten Blättern, [21]

Auch in der Forschung, den Ingenieurswissenschaften und der Wirtschaft sind Simulationen mittlerweile ein wichtiges Hilfsmittel. Experimente, wie beispielsweise Crashtests, sind sowohl sehr teuer, als auch zum Teil nicht einfach umsetzbar. Hier ist der Einsatz von Simulationen, die mit Hilfe eines HPC-Systems in einer relativ kurzen Zeit durchgeführt werden können, meist eine kostengünstige und schnelle Alternative. So können Strömungen von Luft durch Flugzeugturbinen (siehe Abb. 2) oder um Tragflächen oder von Treibstoff in einem Verbrennungsmotor simuliert werden, was eine Optimierung hinsichtlich beispielsweise des Luftwiderstands, Auftriebs, Treibstoffverbrauchs oder der Geräuschentwicklung ermöglicht. [2, S. 5, Kp. 2.1.3]

Abbildung 3: Darstellung des horizontalen Dreieckgitters des globalen ICON-Modells des DWD mit einer Verfeinerung des Gitters bzw. erhöhter Auflösung über Europa, [16, S. i]

Auch aktuelle Wettervorhersagen, und insbesondere Simulationen für die Klima-Forschung, profitieren von HPC-Systemen. Beispielsweise setzt der Deutsche Wetterdienst (DWD) laut [19] zwei Linux HPC-Systeme für die Berechnung lokaler und globaler Wettervorhersagemodelle ein. Im ICON (Icosahedral Nonhydrostatic) Modell des DWD, welches ein globales Wettervorhersagemodell ist, wird die Erdkugel mit einem Dreiecksgitter in Abschnitte eingeteilt, die einer Maschenweite von 13 km entsprechen. Für die gesamte Erdkugel sind das 2.949.120 Dreiecksflächen, in denen die Atmosphäre zwischen Boden und 75 km Höhe in weitere 90 Schichten unterteilt wird, woraus sich insgesamt 265 Millionen Gitterpunkte ergeben (siehe Abb. 3), die jeweils ihre eigenen skalaren Eigenschaften, wie Temperatur, Luftdruck und Luftfeuchtigkeit,

haben. Für alle diese Gitterpunkte werden horizontale und vertikale Transportvorgänge simuliert, inklusive dem Einfluss durch Prozesse, wie Strahlung, Turbulenzen, Wolkenbildung und Niederschlag [20, Abs. 7]. Ein Durchlauf dieses Modells auf dem HPC-System des DWD, für eine Prognose der nächsten sieben Tage, dauert etwa acht Minuten und erzeugt eine Datenmenge von ca. 900 GB. Ein solcher Modelllauf wird mehrmals pro Tag anhand von neu zu Verfügung stehenden Beobachtungsdaten (den tatsächlich gemessenen Werte der Stationen im Messnetz) erstellt, weshalb die Programmlaufzeit hinreichend kurz sein muss, da noch einige weitere Berechnungen auf dem HPC-System durchgeführt werden müssen, beispielsweise für Modelle, die den Europäischen Kontinent oder auch nur Mitteleuropa und Deutschland, mit größerer Auflösung als das Globalmodell, abdecken. [20, Abs. 8, 9]

2 Scheduling in HPC-Systemen

Um die benötigten Ergebnisse eines Programmlaufs möglichst schnell zu erhalten ist es immernoch oft der Fall, dass eine Organisation oder auch nur eine Abteilung einer Organisation (sowohl wirtschaftlicher, als auch akademischer Natur) ihr eigenes HPC-System betreibt. Dies resultiert jedoch in relativ hohen Kosten für die Anschaffung und den Betrieb der Maschine, da entsprechend ausgebildetes Personal unterhalten werden muss. Wenn allerdings nur eine Abteilung exklusiven Zugang auf ihr HPC-System besitzt, so kann es dazu kommen, dass das System nicht vollständig ausgelastet wird, wenn zu einem Zeitpunkt nicht genug Aufgaben anstehen, die berechnet werden sollen. Damit kann das Betreiben eines solchen Systems unwirtschaftlich werden. Um das Verhältnis zwischen Kosten und Nutzen eines Systems möglichst hoch zu halten, muss ein HPC-System bestmöglich ausgelastet werden, andernfalls hätte man das System möglicherweise auch vorab kleiner dimensionieren können und damit wiederum geringere Anschaffungs- und Betriebskosten erzeugt. Um eine bestmögliche Auslastung zu garantieren, ist eine gemeinsame Nutzung des HPC-Systems durch mehrere Parteien oft die einzige Möglichkeit.

2.1 Warteschlangen-basiertes Scheduling

Aufgrund der oben genannten Problematik sind solche HPC-Systeme oft darauf ausgelegt mehr als nur ein bestimmtes Programm gleichzeitig durchzuführen, sofern ansonsten nicht alle Kapazitäten des Systems voll ausgeschöpft werden würden. Um festzulegen, welches Programm wann und mit welchen Eingabedaten und Parametern ausgeführt werden soll, bedarf es eines standardisierten Ablaufes, der eine Reihenfolge der Anwendungen bzw. einen Ablaufplan bestimmt. Den Prozess der Erstellung dieses Ablaufplans bezeichnet man als das Scheduling und das System, welches für die Erstellung verantwortlich ist, als den Scheduler. In Folge wird die Einheit von einem Programm und seinen Eingabedaten und -parametern als Job bezeichnet. Ein Scheduling kann auf unterschiedliche Art und Weise implementiert sein, aber im einfachsten Fall ist es eine First-In-First-Out (FIFO) Warteschlange, in der die Jobs stehen, die nacheinander auf dem HPC-System ausgeführt werden sollen. Wenn das HPC-System aber von einem Programm nicht vollständig ausgelastet wird, besteht die Möglichkeit noch weitere Jobs gleichzeitig durchzuführen, unter der Voraussetzung, dass die dafür benötigten zusätzlichen Ressourcen nicht belegt sind. Dafür muss der Scheduler mindestens die Menge an verfügbaren Ressourcen, wie Prozessoren und Hauptspeicher kennen und außerdem wissen, welche Anforderungen an diese Ressourcen durch die ausstehenden Jobs in der Warteschlange gestellt werden. Mit diesen Informationen kann der Scheduler entscheiden, ob weitere Jobs, parallel zu den bereits laufenden, gestartet werden können oder ob noch gewartet werden

muss, da die erforderlichen Ressourcen noch blockiert sind. Um einen solchen Scheduler zu optimieren, kann man auch Jobs, die eigentlich in der Warteschlange weiter hinten stehen, zur Ausführung bringen, sollte es die Ressourcenbelegung zulassen und die Laufzeit nicht länger sein, als das längste bereits laufende Programm, vorausgesetzt, dass für keinen Job höherer Priorität in der Warteschlange die freien Ressourcen hinreichend für eine Ausführung wären. Diese Optimierung wird auch als Backfilling bezeichnet. Dieser Warteschlangen-basierte Ansatz, in Verbindung mit Backfilling und zumeist zusätzlichen Regeln zur Sicherung der Gleichberechtigung zwischen verschiedenen Nutzern oder Anwendungen, ist die Art und Weise, wie das Scheduling auch in modernen HPC-Systemen abläuft.
Diese relativ rudimentäre Lösung bringt allerdings Nachteile mit sich: Werden die Ressourcen so exklusiv vergeben, kann das darin resultieren, dass nicht alle Teile eines vergebenen Knotens maximal ausgenutzt werden, wie es für eine gute Effizienz erforderlich wäre. Abhängig davon, was das derzeit laufende Programm in einem bestimmten Moment ausführt, kann es beispielsweise sein, dass die Auslastung durch den Datenverkehr über das Netzwerk begrenzt ist: Wartet beispielsweise ein Prozess auf die Antwort eines anderen Prozesses, da er von dessen Ergebnissen abhängig ist, wird zu diesem Zeitpunkt nur ein geringer Anteil der Rechenressourcen tatsächlich genutzt. Gleiches gilt für relativ langwierige Zugriffe auf Festplattenspeicher oder Ähnliches. Auch andere Ressourcen, wie beispielsweise eine GPU (graphics processing unit), könnten durch eine solche feste Zuteilung möglicherweise weniger effizient genutzt werden. Im Optimalfall könnte ein Scheduler entsprechende Situationen erkennen und die freien Ressourcen einem anderen Prozess zur Verfügung stellen, der diese nutzen könnte und so die Antwortzeit der Programme auf einem HPC-System, abhängig vom konkreten Anwendungsfall, signifikant verringern. Die Schwierigkeit in der Umsetzung eines solchen, verbesserten Systems liegt darin, dass die Scheduler Informationen über alle Programme des HPC-Systems zur Laufzeit bräuchten. Der Scheduler muss wissen wie viele der ihm zugeteilten Ressourcen (z. B. CPUs, Hauptspeicher, Grafikkarten, etc.) die teilnehmenden Prozesse aktuell nutzen, um flexibel auf entsprechende Szenarien reagieren zu können.
Dennoch muss insbesondere in einem gemeinsam genutzten HPC-System berücksichtigt werden, dass bestimmte, von teilnehmenden Nutzern gestellte, Bedingungen (z. B. die zugewiesenen Ressourcen oder der Platz in der Warteschlange) so gut wie möglich eingehalten werden. Zu diesen Bedingungen könnten zukünftig auch Deadlines (bezüglich des Antwortzeitpunkts) gehören. Dies lässt sich mit einem Warteschlangen-basierten Ansatz allerdings nur schwer umsetzen, da zu einem bestimmten Zeitpunkt nicht klar ist, wann ein Programm tatsächlich zur Ausführung kommt, da aufgrund von Backfilling oder Mechanismen zur Wahrung der Gleichberechtigung (sodass nicht ein einzelner Nutzer oder eine einzelne Anwendung das System über lange Zeit exklusiv belegen kann) der Zeitpunkt der Ausführung und damit die gesamte

Antwortzeit nicht genau vorhergesagt werden kann.
Im Optimalfall würde man außerdem die Möglichkeit bereitstellen, bei Lastspitzen das HPC-System um externe Ressourcen zu erweitern, insofern das einen Vorteil für den konkreten Anwendungsfall bringt. Hängt eine Anwendung beispielsweise stark von der Netzwerklatenz ab, da viel Kommunikation zwischen den Knoten nötigt ist, würde das bestehende System möglicherweise verlangsamt werden, sofern man einen Teil der Aufgaben auf externe Ressourcen auslagert, da diese vermutlich schlechter angebunden sind, als die Ressourcen innerhalb des HPC-Systems. Einen solchen Job würde man eher vollständig auf ein externes System auslagern, um die Latenzen innerhalb des Jobs gering zu halten und gleichzeitig dennoch die Lastspitze auszugleichen. Damit ein hypothetisches, optimales System solche Entscheidungen treffen könnte, müssten ihm entsprechende Informationen über die betrachtete Anwendung, entweder vor oder zur Laufzeit, zur Verfügung gestellt werden.
Die Idee eines verteilten Systems, innerhalb dem Ressourcen geteilt und verteilt werden können, wurde bereits von Foster und Kesselman [7] unter der Bezeichnung "Grid-System" vorgeschlagen. Darin wurde das Konzept einer allgemeinen Infrastruktur beschrieben, die, analog zum elektrischen Stromnetz, ihre Rechenressourcen zur Verfügung stellt, die durch die Anwender transparent genutzt werden können. Für den Nutzer soll die mögliche Heterogenität des Systems soweit abstrahiert sein, dass der Nutzer selbst nichts über das physische (Teil-) System wissen muss, auf dem die Anwendung ausgeführt werden soll.

2.2 Plan-Based-Scheduling

Zu dem Ansatz eine Warteschlange zu benutzen, sodass Prozesse die zuerst eingereiht wurden, auch normalerweise zuerst ausgeführt werden (unter Berücksichtigung der Ausnahmen, die z. B. durch Backfilling entstehen), gibt es aber insbesondere eine Alternative, dessen Umsetzung aber noch ein aktueller Forschungsgegenstand ist: Ein planungsbasiertes Scheduling. Dabei würden die Informationen genutzt werden, die aus der derzeitigen Laufzeit sowie aus historischen Laufzeitdaten hervorgehen, wie z. B. hinsichtlich der Ressourcenbelegung, o. Ä.. In Kombination mit den Anforderungen der Nutzer, z. B. bezüglich benötigte Ressourcen oder einzuhaltende Deadlines (die maximal erwartete Antwortzeit), wird ein ein Plan erstellt, der den, anforderungsbedingt, "besten" Kompromiss darstellt, zwischen den verschiedenen Nutzeranforderungen und dem was das System zur Verfügung stellen kann.
Bezüglich des Verhandlungsprozesses wird in [8, S. 2, Abs. 2] das Prinzip eines sogenannten "Virtual Resource Manager" (VRM) beschrieben, der für die Verwaltung und Verteilung der Ressourcen in einem Grid-System dient. Auf jede Anfrage eines Klienten an das Grid-System, bzgl. der Ausführung einer Anwendung, folgt eine Verhandlung mit dem VRM, basierend auf der Anfrage und dem aktuellen Zustand des Systems, inklusive des aktuellen Ausführungs-

plans. Das Ergebnis dieses Prozesses ist ein Vertrag, welcher in [8, S. 2, Abs. 1] als "Service Level Agreement" (SLA) bezeichnet wird. Die Anfrage muss sowohl die Anwendung und ihre Eingabedaten und Parameter, als auch Vorhersagemodelle und historische Daten über den Programmablauf und die voraussichtliche Ressourcenbelegung zur Verfügung stellen, die es dem VRM ermöglichen die Anfrage in den vorhandenen Scheduling-Plan zu integrieren.
Damit können im Optimalfall gewisse Garantien bezüglich der Antwortzeiten durch das System zugesichert werden. Diese Garantien sind vor allem für die wirtschaftliche Nutzung von HPC-Systemen interessant, da die Einhaltung von Deadlines, z. B. für die Funktionstüchtigkeit von Produkten der HPC-System-Nutzer, eine wichtige Rolle spielen kann. Die Einhaltung dieser Garantien hängt allerdings stark davon ab, wie gut der planungsbasierte Scheduler informiert ist: Je genauer die Laufzeit eines Programms und die benötigten Ressourcen bekannt sind, desto besser kann ein solcher Scheduler arbeiten und desto geringer ist auch der entstehende Fehler in der Vorhersage der Antwortzeit. Weichen die gegebenen Informationen zu stark von dem ab, was tatsächlich auf dem System passiert, könnte ein solches System nur eingeschränkt funktionieren und die Garantien würden nicht eingehalten werden können.

3 Problemlage

Aktuelle HPC-Verwaltungssysteme haben allerdings nur sehr eingeschränkte Informationen über den konkreten Zustand aller beteiligten Prozesse. So beispielsweise bei der Verwaltung von Prozessen: Derzeit stellt das Verwaltungssystem dem Betriebssystem die Anforderungen für die Erstellung der benötigten Prozesse zu, welches in Folge die Prozesse seinerseits startet. Um welche Prozesse, insbesondere deren Identifikatoren, es sich dabei aber genau handelt, darüber besitzt das Verwaltungssystem keine Informationen. Dies führt dazu, dass der konkrete Zustand der beteiligten Prozesse eines Jobs zur Laufzeit nicht ersichtlich ist und das Verwaltungssystem demnach auch nicht entscheiden kann, ob einzelne Prozesse z. B. ihren Zeitplan (in einem System mit einem zukünftigen Planungsbasierten Scheduler, siehe Kapitel 2.2) erfüllen.
Des Weiteren kann nicht gewährleistet werden, dass das Verwaltungssystem festlegen kann, welcher Prozess auf welchem Prozessor eines Knotens ausgeführt werden soll, da das Betriebssystem des Knotens die finale Entscheidung der Zuordnung trifft. Insbesondere ist dies in einem heterogenen System wichtig, da so festgelegt werden kann, dass ein bestimmter Prozess beispielsweise auf einem leistungsfähigeren Prozessor ausgeführt werden soll.
In dieser Arbeit soll im Folgenden eine potentielle Lösung, für das Problem der Zuordnung und Festlegung der Prozesse auf bestimmte Knoten und Prozessoren in einem HPC-System, vorgestellt werden, die für eine Middleware nach der MPI-Spezifikation prototypisch für ein Linux-System (aufgrund der Verbreitung im HPC-Bereich mit einem Anteil von 100% in den 500 leistungsstärksten HPC-Systemen "Top500", siehe [17, TOP500 Release: June 2021, Category: Operating system family]) implementiert werden soll. Daher wird im Folgenden zunächst auf die MPI-Spezifikation im Kontext der hier beschriebenen Problemlage näher eingegangen.

4 MPI

Die Umsetzung eines nicht-trivialen, parallelen Programms benötigt ein Programmiermodell, welches festlegt, auf welche Art die teilnehmenden Prozesse miteinander kooperieren sollen. Wenn sich alle Prozesse des parallelen Programms auf dem selben Rechner befinden, kann dies z. B. mit einem gemeinsam genutzten Speicher (was auf verschiedenste Art implementiert sein kann), erreicht werden, über den die Prozesse unter anderem auch implizit kommunizieren können. In einem verteilten System aus mehreren verschiedenen Rechnern ist es aber üblicher (auch wenn es Lösungen für verteilten, gemeinsamen Speicher gibt) auf ein paralleles Programmiermodell zu setzen, welches auf der expliziten Übermittlung von Nachrichten zwischen den Prozessen basiert.
Um den Unterschied zwischen den beiden Programmiermodellen zu verdeutlichen, ist das Standardbeispiel für ein paralleles Programm nützlich: Eine Implementierung der Matrizenmultiplikation. Hierbei kann das Zusammenrechnen der Zeilen und Spalten der Matrix relativ einfach auf mehrere Prozesse verteilt werden. In einem parallelen Programm, welches auf gemeinsamem Speicher basiert, würden alle teilnehmenden Prozesse Anteile der Rechnung zugewiesen bekommen und ihre Ergebnisse an bestimmte, ihnen zugewiesene Stellen in der gemeinsam manipulierbaren Ergebnismatrix schreiben. Eine explizite Kommunikation zwischen Prozessen passiert hier nicht. In einem nachrichten-basierten, parallelen Programm wiederum würde der Prozess des Zusammenfügens der Einzelresultate z. B. so aussehen, dass ein Prozess die einzelnen Ergebnisse als explizite Nachrichten von den anderen Prozessen bekommt und diese dann zusammenfügt, um die Ergebnismatrix zu erhalten.
MPI ("Message-Passing Interface") ist die Spezifikation für die Schnittstelle einer Software-Bibliothek für Nachrichten-Übermittlung in Programmen [5, S. 1, Abs. 1, 3]. Es ist als Standard hauptsächlich darauf ausgerichtet, ein nachrichten-basiertes Programmiermodell für parallele Programme zu spezifizieren. Ein Beispiel eines solchen Programms wäre eine Implementierung des in Kapitel 1 (Abs. 3) beschriebenen Algorithmus zur numerischen Berechnung des Mehrkörperproblems. Dabei beinhaltet MPI auch die Anforderung, dass seine Implementierungen heterogene Systeme unterstützen sollen. Nicht zuletzt daher eignet es sich insbesondere gut für die Nutzung in verteilten Systemen und wird auch in gängigen HPC-Systemen verwendet, wie z. B. in den Systemen des DWD [19] (siehe Kapitel 1, Abs. 6) oder des CERN [4, Abs. 1]. Aufgrund dieser Eignung und Verbreitung wurde MPI als Spezifikation für die Implementierung gewählt, für die der Prototyp dieser Arbeit entwickelt werden soll.
Allgemein spezifiziert MPI einige grundlegende Aufgaben, die eine Implementierung mit sich bringen muss: Die Gruppierung von Prozessen, die Identifikation einzelner Prozesse und Prozessgruppen, damit eine Koordination, insbesondere von parallel laufenden Prozessen möglich ist, sowie auch die Initialisierung der Prozesse im Kontext von MPI. Die Gruppierung und

Identifikation macht es im Fall der oben beschriebenen, nachrichten-basierten Implementierung der Matrizenmultiplikation relativ einfach möglich, dass ein ganz bestimmter Prozess von der Gruppe der anderen Prozesse alle ihre Zwischenergebnisse empfangen und zusammensetzen kann.
Konkret bietet eine Middleware nach MPI-Spezifikation sowohl Punkt-zu-Punkt- Kommunikation, mit Befehlen zum Senden (`MPI_Send`) und Empfangen (`MPI_Recv`) von Nachrichten, als auch kollektive Kommunikation, wie beispielsweise, wenn ein Prozess einer Gruppe von Prozessen, z. B. mit `MPI_BCAST` etwas senden will (One-To-All), alle Prozesse z. B. ihre Resultate an einen Prozess mit `MPI_GATHER` weitergeben wollen (All-To-One) oder alle Prozesse allen anderen bestimmte Informationen zukommen lassen sollen (All-To-All), wofür man beispielsweise `MPI_ALLGATHER` nutzen könnte [5, S. 193, Kp. 6.2.2]. Des Weiteren in MPI spezifiziert sind die Konzepte von Gruppen ("Groups") und Kommunikatoren ("Communicators"), Prozess-Topologien, die Verwaltung der Umgebung, Prozesserstellung und -verwaltung, kollektive Operationen, externe Schnittstellen, Ein- und Ausgabe ("I/O") und Weiteres [5, S. 2, Abs 1].

4.1 Gruppen und Kommunikatoren

Damit die Entwicklung paralleler Programme unterstützt werden kann, bietet MPI eine Reihe von Datenstrukturen und Abstraktionen, um einen konsistenten Umgang mit unterschiedlichen parallelen Systemen zu ermöglichen. Zur Lösung des in Kapitel 3 vorgestellten Problems ist es dabei vor allem von Relevanz die Prozessverwaltung, sowie zwei grundlegende Konzepte von MPI zu verstehen: "Groups" und "Communicators".

4.1.1 MPI Groups

Um kollektive Operationen über mehreren Prozesse durchführen zu können, benötigt man das Konzept einer Gruppe von Prozessen, was von MPI als "Group" bezeichnet wird [5, S. 311, Kp. 7]. Genauer handelt es sich bei einer Group um eine geordnete Menge von Prozess-Kennungen, anhand derer man einzelne Prozesse eindeutig identifizieren kann (woraus diese Kennung genau besteht obliegt der Implementierung). In dieser Group ist jedem Prozess ein ganzzahliger Rang (im Folgenden, nach MPI-Spezifikation, als "Rank" bezeichnet) zugeordnet. Die Zuordnung dieser Ranks zu den Prozessen ist, von Null beginnend, fortlaufend, womit der höchste Rank in einer Group mit einer Anzahl von N Prozessen der Rank N-1 ist. Dieses Spezifikationsdetail ist in Folge noch wichtig, weil es die Implementierung einer Lösung, bezüglich der möglichen Manipulation von Ranks, einschränkt, da der Rank jedes Prozesses somit nicht kleiner als Null oder größer als N-1 sein darf, ohne die MPI-Spezifikation zu verletzen. [5,

S. 311, Kp. 7.2.1]

4.1.2 MPI Communicators

Kommunikation in MPI geschieht immer im Kontext eines bestimmten sogenannten "Communicator", der den Geltungsbereich, also die Menge an teilnehmenden Prozessen, einer Punkt-zu-Punkt oder kollektiven Operation definiert und damit auch für die maschinen-unabhängige Adressierung der Prozesse durch die Ranks verantwortlich ist. Ein Communicator beinhaltet immer die teilnehmenden Prozesse in Form einer Group. Nachrichten zwischen den Prozessen werden anhand ihres Ranks innerhalb der Group zugestellt. Ein solcher Communicator der innerhalb einer Group vermittelt, wird als "Intra-Communicator" bezeichnet. [5, S. 311, Kp. 7.2.3]
Im Gegensatz dazu unterstützt MPI auch Kommunikation zwischen zwei verschiedenen, disjunkten Groups. Für diesen Zweck sind "Inter-Communicators" spezifiziert, die jeweils genau eine "local Group" und eine "remote Group" aufweisen, zwischen denen Kommunikation ermöglicht werden soll. Ob sich ein Prozess in der "local Group" oder der "remote Group" befindet, ist immer relativ zum derzeit betrachteten Prozess: Die eigene Group eines Prozesses ist die "local Group" und die, relativ zum betrachteten Prozess, andere Group ist die "remote Group". [5, S. 328, Kp. 7.4.2, Abs. 2]

4.2 Prozessverwaltung in MPI

Auch wenn sich MPI hauptsächlich mit der Kommunikation und weniger mit Ressourcenverwaltung beschäftigt, ist es doch notwendig gewisse Schnittstellen für das Prozessmanagement und insbesondere die Initialisierung von Prozessen zu Verfügung zu stellen, ohne jedoch zu große Einschränkungen bezüglich der Ausführungsumgebung zu machen, um eine maximale Kompatibilität zu verschiedenen Systemen zu gewährleisten. [5, S. 487, Kp. 11.1, Abs. 1-2]

4.2.1 MPI Prozess Initialisierung

Bevor es möglich ist alle Funktionalitäten zu nutzen, die MPI einem Programm während der Laufzeit zur Verfügung stellt, muss ein solches Programm entsprechende Initialisierungen für bestimmte Teile von MPI durchführen. MPI bietet dafür zwei verschiedene Modelle zur Initialisierung von Prozessen an: Das "World Model" und das "Sessions Model" [5, S. 487, Kp. 11.1, Abs. 3]. Im "World Model" wird unter anderem der Communicator MPI_COMM_WORLD erstellt und die initial verfügbare Menge an Prozessen wird der Group dieses MPI_COMM_WORLD-Communicator hinzugefügt [5, S. 312, Kp. 7.2.4, Abs. 1].

Das (und viele weitere Initialisierungen von notwendigen MPI-Strukturen) geschieht durch die Ausführung der Routine MPI_INIT (siehe Listing 1, Z. 3) oder MPI_INIT_THREAD. Sobald MPI_INIT oder MPI_INIT_THREAD erfolgreich durchgeführt worden sind und solange MPI_FINALIZE (zum schlussendlich Aufräumen vor Programmende, siehe Listing 1, Z. 8) noch nicht aufgerufen wurde, ist MPI_COMM_WORLD valide und benutzbar. Bei einer Nutzung des "Sessions Model" andererseits werden zwar initial Prozesse erzeugt, aber hier ist die Applikation selbst für die Erstellung und Verwaltung der Groups und Communicators zuständig und die MPI_COMM_WORLD ist kein valider Communicator, solange er nicht von der Applikation (im Gegensatz zu MPI, im Fall des "World Model") selbst erzeugt wird. [5, S. 487, Kp. 11.1, Abs. 3]

```
1 int main(int argc, char *argv[])
2 {
3   MPI_Init(&argc, &argv);
4
5   /* Programmablauf */
6
7   MPI_Finalize();
8   return 0;
9 }
```

Listing 1: "Skelett" eines MPI-Beispielsprogramms unter Nutzung des "World Model" von MPI

Im weiteren Verlauf der Arbeit wird implizit immer die Initialisierung mit dem "World Model" betrachtet, sollte es nicht explizit anders beschrieben sein.

4.2.2 MPI Dynamic Process Model

Mit beiden Initialisierungsmodellen ist es jedoch auch möglich weitere Prozesse während der Laufzeit zu erstellen [5, S. 487, Kp. 11.1, Abs. 4]. Diese Fähigkeit wird von MPI durch das "Dynamic Process Model" spezifiziert, welches, unter anderem, auch für den Aufbau der Kommunikation zwischen neu erstellten Prozessen und der laufenden MPI-Applikation verantwortlich ist. Zur dynamischen Erzeugung neuer Prozesse dienen die Routinen MPI_COMM_SPAWN und MPI_COMM_SPAWN_MULTIPLE. Die neu erzeugten Prozesse werden in Folge als Kind-Prozesse und die erzeugenden Prozesse (die einen der beiden MPI_COMM_SPAWN-Aufrufe durchgeführt haben) als Eltern-Prozesse bezeichnet. Der Unterschied zwischen den beiden Routinen liegt hauptsächlich darin, dass MPI_COMM_SPAWN_MULTIPLE Prozesse anhand verschiedener Binärdateien oder der gleichen Binärdatei mit verschiedenen Argumenten gleichzeitig erzeugt, während MPI_COMM_SPAWN nur zu genau einer ausführbaren

Binärdatei, und einem Satz von Argumenten, einen oder mehrere neue Prozesse erstellt. [5, S. 521, Kp. 11.7.1, Abs. 2]
Beide Routinen erzeugen die Kind-Prozesse allerdings nicht in eine bestehende MPI_COMM_WORLD hinein, sondern erzeugen jeweils eine eigene MPI_COMM_WORLD pro Aufruf der Routine, die separat von der bestehenden MPI_COMM_WORLD der Eltern-Prozesse ist. Das bedeutet beispielsweise, dass, wenn bei einem Aufruf von MPI_COMM_SPAWN nur ein Prozess erzeugt wird bzw. werden soll, dieser als einziger in seiner eigenen MPI_COMM_WORLD ist und damit den Rank Null trägt. Die Kommunikation zwischen den Kind- und den Eltern-Prozessen wird über einen Inter-Communicator (siehe Kapitel 4.1.2) realisiert, den der Prozess-Erzeugungs-Aufruf zurückgibt. Die beiden Groups des Inter-Communicators sind einerseits die Group der Kind-Prozesse und andererseits die Group des entsprechenden Eltern-Prozesses, wobei sich, aus Sicht der Eltern-Prozesse, in der "remote group" die Kind-Prozesse und in der "local group" die Eltern-Prozesse befinden. Da es sich bei den Kind-Prozessen auch um MPI-Applikationen handelt (in dem hier betrachteten Fall, auch wenn damit grundsätzlich auch nicht-MPI-Prozesse gestartet werden können), rufen auch diese die MPI_INIT-Routine auf, infolgedessen die Kommunikation mit ihren Eltern-Prozessen stattfinden kann. [5, S. 523-24, Kp. 11.8.2, Abs. 2, unten]

4.2.3 Prozesse in MPI

Ein Prozess wird in MPI repräsentiert durch das Tupel aus der Group und dem Rank, den der Prozess in dieser Group hat [5, S. 522, Kp. 11.8.1, Abs. 1]. Dieses Tupel (Group, Rank) identifiziert einen Prozess eindeutig, während sich aus einem Prozess nicht eindeutig auf das Tupel (Group, Rank) schließen lässt, da ein Prozess gleichzeitig zu mehreren, verschiedenen Groups gehören kann.
Der Rank eines Prozesses ist also immer relativ zu der Group in der man ihn betrachtet.

5 OpenMPI

Da es sich bei MPI um eine Spezifikation handelt, ist es notwendig, für die konkrete Lösung, bzw. für die Umsetzung eines Prototypen, eine Implementierung dieser Schnittstellen-Spezifikation heranzuziehen. Dafür wurde in dieser Arbeit "OpenMPI" verwendet. Hierbei handelt es sich um eine quell-offene, frei verfügbare und vollständige Implementierung der MPI-Version 3.1 [13, Abs. 2], die sich in aktiver Entwicklung befindet (siehe [15]). Dadurch, dass OpenMPI diese MPI-Spezifikation vollständig implementiert, wird auch die Anforderung erfüllt, dass die gewählte MPI-Implementierung das dynamische Erzeugen von Prozessen während der Laufzeit unterstützen soll, was bereits seit MPI-2.0 der Fall ist [6]).

5.1 Programmstart

Ein OpenMPI-Programm startet man üblicherweise mit dem Befehl `mpirun [Optionen] <Programm> [<Argumente>]` [14, Kp. 2]. Dabei lassen sich einige Einstellungen für den Ablauf des Programms, insbesondere bezüglich der genutzten Prozessoren und MPI-Knoten, treffen, die bereits einen Teil des in Kapitel 3 beschriebenen Problems lösen. Mittels der Option `-np X` kann man eine Anzahl von `X` identischen Prozessen starten, die das gegebene `<Programm>` unter Verwendung der `<Argumente>` ausführen. Ohne weitere Angaben, auf die im Folgenden eingegangen werden soll, werden die Prozesse auf dem lokalen MPI-Knoten ausgeführt und nach dem Round-Robin-Prinzip auf die vorhandenen *CPU-Slots* verteilt. Unter dem Begriff *CPU-Slots* können zwei verschiedene Konzepte verstanden werden: Standardmäßig ist darunter die Anzahl an CPU-Kernen zu verstehen, die der HPC-Knoten besitzt. Sofern allerdings die Option `--use-hwthread-cpus` genutzt wird [14, Kp. 3, Abs. 3], wird die Anzahl an Hardware-Threads bzw. logischen Prozessoren der CPUs des aktuellen HPC-Knotens genutzt, was dazu führen kann, dass mehr CPU-Slots verfügbar und damit mehr Prozesse gleichzeitig ausführbar sind (falls die CPU das unterstützt, z. B. durch die "Hyperthreading"-Technik bei Intel CPUs). Dies muss aber, abhängig von der Anwendung, nicht unbedingt in einer Verringerung der Laufzeit resultieren und kann schlimmstenfalls die Laufzeit sogar verlängern, insbesondere, wenn die Threads untereinander kommunizieren müssen, aber nicht echt gleichzeitig durchgeführt werden können (bedingt durch die Implementierung der Hardware-Threads im Gegensatz zu der Nutzung von CPU-Kernen).
Das gilt allerdings nur dann, wenn die OpenMPI-Anwendung nicht mit einem Ressourcen-Verwaltungs-System, wie z. B. SLURM (Simple Linux Utility for Resource Management), gestartet wurde. Ist das der Fall, bekommt die OpenMPI-Anwendung die Informationen über die nutzbaren Ressourcen von dem Ressourcen-Verwaltungs-System. Damit ist es möglich, dass die zugewiesenen Ressourcen nicht den Ressourcen entsprechen, die das System tatsäch-

lich aufweist: Es können der OpenMPI-Anwendung sowohl weniger Slots, als auch mehr Slots zugewiesen werden, als der HPC-Knoten tatsächlich besitzt, wobei Letzteres als "Overprovisioning" bezeichnet wird. [14, Kp. 4] Insbesondere die Nutzung einer größeren Anzahl von Slots, als CPU-Kerne verfügbar sind, wirft potentiell ähnliche Probleme auf, wie sie bei der Nutzung von Hardware-Threads anstatt von CPU-Kernen auftreten (siehe oben).
Ein aktuelles Verwaltungssystem macht es für den Nutzer üblicherweise transparent, welche anderen Prozesse bzw. Jobs noch auf dem HPC-System laufen und auf welchem Knoten und welchem Prozessor das Programm schlussendlich ausgeführt wird. Der Nutzer übergibt nur den Job der ausgeführt werden soll, in dem unspezifische (insbesondere nicht auf bestimmte Prozessoren festgelegte) Bedingungen bezüglich der benötigten Ressourcen gestellt werden, und das Verwaltungssystem verteilt die Prozesse des Jobs auf die freien Ressourcen bzw. spezifischen Knoten und Prozessoren, insofern es die Belegung zu diesem Zeitpunkt zulässt.
Möchte man für eine Anwendung aber definieren, auf welchem spezifischen HPC-Knoten sie durchgeführt werden soll, z. B. wenn ein besonders leistungsstarker Knoten genutzt werden soll, erreicht man dies aktuell üblicherweise nur durch manuelle Konfiguration. Das kann man sowohl mittels der Option `-host <HPC-Knoten>` definieren, oder mit Hilfe eines sogenannten Hostfiles, welches mit der Option `-hostfile <myhostfile>` festlegbar ist. In diesem Hostfile (eine Textdatei) werden die Knoten des HPC-Systems beschrieben, auf denen die OpenMPI-Anwendung gestartet werden kann und außerdem, wie viele Slots auf den einzelnen HPC-Knoten belegt werden können. [14, Kp. 9]
In Listing 2 ist beispielhaft der Inhalt eines solchen Hostfiles dargestellt, welches drei Knoten aa, bb und cc beschreibt, von denen jeweils vier Slots genutzt werden können und von denen die Knoten aa und bb vor Überbelegung (Overprovisioning) durch den Parameter `max_slots` geschützt sind.

```
1 aa slots=4 max_slots=4
2 bb max_slots=4
3 cc slots=4
```

Listing 2: Beispiel für ein *Hostfile* [14, Kp. 9]

Nutzt man nun die Option `-np` um die Anzahl an Prozessen anzugeben, werden die Prozesse auf die im Hostfile angegebenen Knoten von oben nach unten verteilt. Das bedeutet, dass die HPC-Knoten, die um Hostfile unten stehen, nicht genutzt werden, wenn weniger Prozesse angegeben wurden als in Hostfile insgesamt, über alle Knoten, Slots verfügbar sind. [14, Kp. 9]
Um die Verteilung der Prozesse auf die vorhandenen Knoten und Slots genauer steuern zu können, lassen sich auch "Zuweisungs-Politiken" (mapping policies) definieren: Anstatt der standardmäßigen Verteilung pro Slot ist es mittels der Option `--map-by <Einheit>` auch

möglich unter anderem nach Knoten (`node`) oder Prozessor(-Sockel) (`socket`) als gewählte Einheit zu verteilen, sodass entsprechende HPC-Knoten oder Prozessoren gleichmäßig ausgelastet werden. Wie unten im Kontext der Option `--bind-to` noch genauer erläutert wird, handelt es sich hierbei aber tatsächlich nur um die Zuweisung auf die Knoten und nicht auf die spezifischen Hardware-Ressourcen der Knoten (z. B. Prozessoren).
Nach der Phase der Zuweisung der Prozesse auf die Knoten (Mapping) werden die Ranks (bzgl. der MPI_COMM_WORLD) an die Prozesse verwiesen (Ranking). Im Folgenden wird mit dem Begriff Rank immer der Rank in der MPI_COMM_WORLD beschrieben, außer es ist explizit anders festgelegt. Auch dieser Vorgang lässt sich bis zu einem gewissen Grad beeinflussen: Dafür ist die `--rank-by <Einheit>` -Option gedacht, bei der die <Einheit> wieder z. B. Knoten (`node`), Prozessor(-Sockel) (`socket`), CPU-Kern (`core`) und Weitere sein können, über die die Ranks gleichmäßig, nach dem Round-Robin-Prinzip, über die gewählte Einheit verteilt werden. Bei einem `--rank-by core` würden beispielsweise alle CPU-Kerne aller Knoten nacheinander durchgegangen und aufsteigend mit einem Rank versehen werden. [14, Kp. 12, Abs. 8]
Nachdem die Ranks zugewiesen wurden, erfolgt das "Binding", bei dem die Prozesse tatsächlich auf bestimmte Hardware-Ressourcen festgelegt werden, was laut [12, Kp. 19, Abs. 3] durch das `--map-by` noch nicht passiert, da dort nur die Zuweisung auf die Knoten festgelegt wird. Stattdessen hat das Betriebssystem des HPC-Knotens vorerst die Verfügungsgewalt über die Verteilung der Prozesse auf die Hardware-Ressourcen des entsprechenden Knotens. Da das Betriebssystem aber keinerlei Informationen über die Anwendung besitzt und nur dazu angewiesen wird eine bestimmte Anzahl an Prozessen zu erzeugen, kann das laut [14, Kp. 12, Abs. 8] zu einer nicht-optimalen Verteilung der Prozesse führen. Für die tatsächliche Ressourcen-genaue Zuweisung der Prozesse dient die Option `--bind-to <Einheit>` , durch welche entschieden werden kann, Prozesse beispielsweise an bestimmte CPU-Kerne zu binden. Auch lassen sich weitere Festlegungen treffen, wie z. B. dass die Prozesse sich ihren Cache bestenfalls nicht teilen sollen, indem man mit der Option `--bind-to l3cache` festlegt, dass jeder Prozess an genau die Prozessoren gebunden sein soll, die sich einen L3-Cache teilen. [14, Kp. 12, Abs. 9]
Alternativ kann man für eine individuelle Zuweisung der Prozesse auch ein sogenanntes Rankfile nutzen. Hierbei handelt es sich um eine Textdatei, in der die initiale Zuweisung der einzelnen Prozesse auf die HPC-Knoten und Prozessoren anhand ihrer Ranks in der MPI_COMM_WORLD beschrieben wird. In Listing 3 ist beispielhaft der Inhalt eines Rankfiles aus der OpenMPI Dokumentation [14, Kp. 13] dargestellt, welches festlegt, dass der Prozess mit Rank null auf den Kernen null bis zwei des Prozessors auf Sockel eins auf dem Knoten aa, dass der Prozess mit Rank eins auf den Kernen eins und zwei des Prozessors auf Sockel

null auf dem Knoten bb und dass der Prozess mit Rank zwei auf den Kernen eins und zwei (unabhängig vom Sockel) des Knotens aa laufen soll.

```
1 rank 0=aa slot=1:0-2
2 rank 1=bb slot=0:0,1
3 rank 2=cc slot=1-2
```

Listing 3: Beispiel für ein Rankfile [14, Kp. 13]

Diese Zuweisung von bestimmten Prozessen an bestimmte Knoten und deren spezifische Ressourcen löst bereits einen Teil der Anforderungen, die aus dem beschriebenen Problem (siehe Kapitel 3) hervorgehen. Mit einem Rankfile lassen sich allerdings nur Prozesse zuweisen, die auch initial gestartet wurden. Es lässt sich damit demnach nicht beeinflussen, wo ein dynamisch gestarteter Prozess erzeugt werden soll, womit die Problemstellung mit diesem Ansatz nicht vollständig gelöst werden kann.

5.2 groups und communicators

Um die Identifikation (und in Folge dessen möglicherweise auch die Zuweisung) von beliebigen, sowohl initial, als auch dynamisch, erzeugten Prozessen möglich zu machen, ist es relevant zu verstehen, wie MPI Groups und Communicators in OpenMPI implementiert sind, da sie die Prozessmengen abstrahieren und demnach intern die Prozesse eindeutig unterscheiden können müssen, womit sich das Zuordnungsproblem (siehe Kapitel 3) lösen lässt.
Der für diese Arbeit relevante Teil der konkreten Implementierung der MPI-Spezifikation einer Group durch OpenMPI ist in Listing 4 abgebildet.

```
84 struct ompi_group_t {
85     opal_object_t super;     /**< base class */
86     int grp_proc_count;      /**< number of processes in group */
87     int grp_my_rank;         /**< rank in group */
88     int grp_f_to_c_index;    /**< index in Fortran <-> C translation array
       */
89     struct ompi_proc_t **grp_proc_pointers;
90                              /**< list of pointers to ompi_proc_t
       structures
91                                   for each process in the group */
```

Listing 4: ompi/group/group.h [15]

Besonders relevant für diese technische Umsetzung des in Kapitel 4.1.1 beschriebenen Konzepts der MPI Group, sind die Felder `grp_proc_count` , `grp_my_rank` und `**grp_proc_pointers` : `grp_proc_count` beschreibt die Anzahl an Prozessen in der Group, `grp_my_rank` beinhal-

tet den Rank des aktuellen Prozesses innerhalb der Group und `**grp_proc_pointers` ist eine Liste von Zeigern auf interne Datenstrukturen (`ompi_proc_t`), die jeweils einen Prozess repräsentieren und Informationen über diesen beinhalten. Die Struktur `ompi_proc_t` hat ein Feld des Typs `opal_proc_t` , was wiederum ein Feld des Typs `opal_process_name_t` enthält. Diese Struktur `opal_process_name_t` ist für diese Arbeit besonders von Bedeutung, da sie aus zwei weiteren Feldern besteht: "jobid" und "vpid". Das sind zwei nicht-negative, ganzzahlige (uint32_t) Identifikatoren, die laut [15, ompi/runtime/ompi_rte.h, Z. 32] in Kombination für einen Prozess (bzgl. des genutzten HPC-Knotens zu einem bestimmten Zeitpunkt) einzigartig sind und ihn damit eindeutig identifizieren.
Das löst das Identifikationsproblem, was in Folge von dem dynamischen Erschaffen neuer Prozesse auftrat (siehe Kapitel 4.2.2): Mit jedem Aufruf der Routinen MPI_COMM_SPAWN oder MPI_COMM_SPAWN_MULTIPLE wird eine neue MPI_COMM_WORLD erstellt und die initial einzigartigen Ranks können mehrfach auftreten (siehe Kapitel 4.2.1). Damit war es notwendig einen anderen Identifikator für den Prozess zu finden als nur den Rank, da dieser so nicht mehr eindeutig war. Dieser Identifikator ist die "jobid", die nach [15, ompi/runtime/ompi_rte.h, Z. 33] einzigartig für jede MPI_COMM_WORLD sein muss und damit eine Unterscheidung von Prozessen gleichen Ranks, in unterschiedlichen MPI_COMM_WORLD, möglich macht.
Das Konzept des MPI Communicator setzt OpenMPI um, wie es in Listing 5, auf das für diese Arbeit Wesentliche reduziert, gezeigt ist. Das Feld `c_my_rank` ist hier der Rank des aktuell betrachteten Prozesses in dem Communicator, der `c_name` ist der Name des Communicators also z. B. MPI_COMM_WORLD und die beiden Felder `*c_local_group` und `*c_remote_group` sind die "local Group" und "remote Group" (siehe Kapitel 4.1.2), die beide nötig sind, sollte es sich um einen Inter-Communicator handeln, der zwischen zwei verschiedenen Groups vermitteln soll. Demnach werden MPI Intra-Communicator und Inter-Communicator in OpenMPI durch die gleiche Datenstruktur (`ompi_communicator_t`) repräsentiert.

```
156 struct ompi_communicator_t {
    ...
160     char  c_name[MPI_MAX_OBJECT_NAME];
161     uint32_t              c_contextid;
162     int                     c_my_rank;
    ...
174     ompi_group_t        *c_local_group;
175     ompi_group_t       *c_remote_group;
```

Listing 5: ompi/communicator/communicator.h [15]

5.3 Communicator-Initialisierung

Um zu verstehen, wie die Teile von MPI_INIT (siehe Kapitel 4.2.1) in der OpenMPI-Implementierung funktionieren, die die Initialisierung von relevanten Strukturen, wie Groups und Communicators durchführen, betrachten wir nun die Implementierung der entsprechenden Stellen in OpenMPI genauer:

```
88 int ompi_comm_init(void)
89 {
...
112     size = ompi_process_info.num_procs;
113     group->grp_proc_pointers = (ompi_proc_t **) calloc (size, sizeof (
    ompi_proc_t *));
114     group->grp_proc_count = size;

116     for (size_t i = 0 ; i < size ; ++i) {
117         opal_process_name_t name = {.vpid = i, .jobid = OMPI_PROC_MY_NAME
    ->jobid};
118         /* look for existing ompi_proc_t that matches this name */
119         group->grp_proc_pointers[i] = (ompi_proc_t *) ompi_proc_lookup (
    name);
...
    }
...
130     ompi_set_group_rank(group, ompi_proc_local());
...
135     ompi_mpi_comm_world.comm.c_my_rank      = group->grp_my_rank;
136     ompi_mpi_comm_world.comm.c_local_group  = group;
...
}
```

Listing 6: ompi/communicator/comm_init.c [15]

In Listing 6 ist der Teil Abschnitt des Quelltextes gezeigt, der bei Aufruf von MPI_INIT die Ranks der Prozesse in der Group der MPI_COMM_WORLD festlegt. Wichtig für das Verständnis hierbei ist, dass jeder einzelne Prozess die MPI_INIT-Routine selbst aufruft und dieser Abschnitt für ein Programm demnach insgesamt so oft aufgerufen wird, wie es teilnehmende Prozesse gibt. Die Perspektive ist also immer aus Sicht desjenigen Prozesses gewählt, der die Initialisierungs-Routine gerade für sich durchführt. Die Festlegung des Ranks z. B. bezieht sich somit auch nur auf den aktuellen Prozess, auch wenn jeder Prozess natürlich auch die Informationen, die er zu den anderen Prozessen hat, mit initialisiert. In Zeile 112 wird die Gesamtanzahl der Prozesse ausgelesen, die zum selben OMPI-Job gehören (unterschieden

durch die "jobid", siehe Kapitel 5.2) wie der aktuelle Prozess (siehe [15, ompi/runtime/ompi_rte.h, Z. 95]). Dann wird ab Zeile 116 eine Schleife durchlaufen, die, beginnend mit Null bis zur ausgelesenen Gesamtanzahl, über die Indizes `i` der einzelnen Prozesse iteriert. In Zeile 117 wird nun mit Hilfe von `OMPI_PROC_MY_NAME` (ein Zeiger auf eine Datenstruktur des Typs `opal_process_name_t`, die den aktuellen Prozess beschreibt) die "jobid" des aktuellen Prozesses ausgelesen, um dann in Kombination mit dem Index `i` als "vpid" die `opal_process_name_t`-Struktur zu bilden, die den Prozess identifiziert. Anhand dieser `opal_process_name_t`-Struktur lässt sich nun mit der Routine `ompi_proc_lookup` die Datenstruktur finden, die den Prozess in der Group repräsentiert und das Ergebnis wird in der Liste `grp_proc_pointers` der Group abgelegt (Zeile 119). Schlussendlich wird mit `ompi_set_group_rank` ab Zeile 130 noch für jeden Prozess in der Group das Feld `grp_my_rank` entsprechend gesetzt und die nun fertig initialisierte Group in dem MPI_COMM_WORLD-Communicator referenziert.

Der Grund warum die notwendigen Änderungen nicht in der Routine `ompi_set_group_rank` durchgeführt werden können, ist, dass damit ausschließlich der Rank bzw. das Feld `grp_my_rank` in der Group-Datenstruktur geändert werden würde. Was diesen Rank im Feld `grp_my_rank` aber direkt bedingt, ist der dazugehörige Index in der Liste `grp_proc_pointers`, in welchem ein Zeiger auf den entsprechenden Prozess abgelegt sein sollte. Ändert man nun das Feld `grp_my_rank`, ohne dabei auch die Prozess-Namen-Zeiger in `grp_proc_pointers` entsprechend zu ändern, so kommt es in anderen Funktionen zu Fehlern, die mit dem Rank direkt auf das Prozess-Namen-Zeiger-Array zugreifen, wie beispielsweise in Zeile 356 des folgenden Abschnitts (Listing 7):

```
345 static inline struct ompi_proc_t *ompi_group_dense_lookup (ompi_group_t *
    group, const int peer_id, const bool allocate)
346 {
...
350     if (peer_id >= group->grp_proc_count) {
351         opal_output(0, "ompi_group_dense_lookup: invalid peer index (%d)",
    peer_id);
...
    }
...
356     proc = group->grp_proc_pointers[peer_id];
357 }
```

Listing 7: ompi/group/group.h [15]

Daher musste stattdessen `grp_proc_pointers`, in dem Moment wo es initialisiert wird, geändert werden, da damit auch die `grp_my_rank`-Felder der einzelnen Prozesse korrekt

gefüllt werden und keine Synchronisierung im Nachhinein notwendig ist (siehe Kapitel 6.2).

6 Lösung

6.1 Ansatz

In der Beschreibung der Problemlage (Kp. 3) werden zwei Teilprobleme vorgestellt, die es eine bessere Kontrolle der Prozesse in einem HPC-System möglich machen: Einerseits die Identifikation von Prozessen eines Jobs und andererseits die Festlegung von diesen Prozessen an bestimmte Ressourcen (insbesondere HPC-Knoten und deren Prozessoren).
Ein Lösungsansatz für die Identifikation wurde für OpenMPI bereits in Kapitel 5.2 (Abs. 4) vorgestellt, nach dem die eigene "jobid" und "vpid" durch jeden Prozess selbst auslesbar ist und daher auch kommuniziert werden kann. Diese Kommunikation kann auf verschiedene Art und Weise implementiert sein.
Bezüglich der Festlegung von Prozessen eines Jobs auf bestimmte Knoten und Prozessoren wurde in Kapitel 5.1 bereits festgestellt, dass die von OpenMPI zur Verfügung gestellten Konfigurationsmöglichkeiten durch Host- und Rankfiles zwar den Teil der Anforderungen abdecken könnten, der die initial erstellten Prozesse betrifft, aber keine Möglichkeiten bieten, dynamisch erstellte Prozesse zu kontrollieren. Daher wurde für den hier vorgestellten Lösungsansatz die Annahme getroffen, dass sich diese Zuweisung durch eine Manipulation der Prozess-Ranks beeinflussen lässt. Aus dieser Annahme folgt, dass das Tauschen von Ranks zweier Prozesse auch einen Tausch der Prozess-"Orte" bzw. der CPU-Slots (siehe Kp. 5.1, Abs. 1) der Prozessoren auf den Knoten bedingt. Für diesen Ansatz die Ranks zu manipulieren, wurde in Kapitel 5.3 die Initialisierung der OpenMPI-Communicators untersucht, welche unter anderem die Zuweisung der Ranks an die Prozesse beinhaltet. Des Weiteren muss berücksichtigt werden, dass die Ranks nicht beliebig auf Prozesse zugewiesen, sondern nur zwischen Prozessen eines Jobs getauscht werden können, aufgrund der in Kapitel 4.1.1 beschriebenen Einschränkungen, die die MPI-Spezifikation vorgibt. Ob die Annahme, mit einer Änderung des Ranks auch die Prozess-Ressourcen-Zuweisung kontrollieren zu können, korrekt ist, ist Gegenstand weiterer Untersuchungen.
Beide Ansätze benötigen einen Kommunikationskanal zum HPC-Verwaltungssystem, was in diesem Ansatz über ein Programm realisiert wird, welches auf jedem HPC-Knoten läuft und mit dem jeder gestartete OpenMPI-Prozess eine Verbindung aufbaut. Dieses Programm, welches in Folge als "Agent" bezeichnet wird, ist die Schnittstelle zwischen allen OpenMPI-Prozessen auf einem Knoten und dem HPC-Verwaltungssystem. Der Agent dient einerseits dazu das HPC-Verwaltungssystem über die Identität (den ursprüngliche Rank, den zugehörigen Job, etc.) der teilnehmenden Prozesse jedes Jobs zu informieren, und andererseits um den Prozessen die ihnen vom HPC-Verwaltungssystem zugewiesenen Ranks zu übermitteln, sodass jeder Prozess den für ihn bestimmten Rank annehmen kann.

6.2 Implementierung

Für die Implementierung einer Lösung wird zunächst ein Kommunikationskanal, in Form des in Kapitel 6.1 (Abs. 4) beschriebenen Agenten, benötigt. Dafür werden in der Lösung, die diese Arbeit vorstellt, Unix-Domain-Sockets genutzt, da diese in jedem Linux-System verfügbar und einfach zu benutzen sind und lokale Kommunikation für den Zweck der Kommunikation zwischen den OpenMPI-Prozessen und dem Agenten auf dem einem gemeinsamen Knoten ausreichend ist. Würde man den Agenten, im Gegensatz zum Testaufbau dieser Arbeit, der aus nur einem Knoten besteht, auf einem tatsächlichen HPC-System nutzen, welches aus mehreren Knoten besteht, müsste auf jedem der Knoten eine Instanz des Agenten laufen.
Der Agent bekommt von jedem OpenMPI-Prozess des entsprechenden Knotens bei der jeweiligen Initialisierung die nötigen Informationen über diesen Prozess, insbesondere die (Linux) PID, den Rank (in seiner eigenen MPI_COMM_WORLD) und die "jobid" zur Unterscheidung der Ranks (siehe Kp. 5.3). Außerdem wird die Anzahl der Prozesse in der lokalen MPI_COMM_WORLD des Prozesses übergeben, da diese notwendig ist, damit bei der Zuordnung klar ist, welche Ranks verfügbar sind, da die Ranks zwischen 0 und der Anzahl der Prozesse, nach MPI-Spezifikation, kontinuierlich sind (siehe Kapitel 4.1.1).
Da es sich hierbei um einen Prototyp handelt, übernimmt der Agent in diesem Fall auch die Verteilung der Ranks auf die Prozesse, da dies den Aufbau erheblich vereinfacht und dennoch die Machbarkeit hinreichend aufzeigt. Insbesondere ist hierbei zu berücksichtigen, dass die Vergabe der Ranks innerhalb der Initialisierung von OpenMPI nicht in beliebiger Reihenfolge vonstattengehen kann. Sonst könnte es dazu kommen, dass z. B. der Prozess mit dem Rank null, der sich damit auf der nullten Stelle in der Liste von Prozessen `grp_proc_pointers` in der Group befindet, aufgerufen wird, obwohl der entsprechende Prozess-Zeiger noch nicht in das Array eingetragen ist (siehe Kp. 5.3, Abs. 3).
Um dennoch die Zuweisung beliebig permutieren zu können und gleichzeitig aber die Reihenfolge der Initialisierung beibehalten zu können, muss der Agent bereits vor der Zuweisung über alle verfügbaren Prozesse aufgeklärt werden und diese auch schon im Vorhinein zuordnen. Das geschieht in den Zeilen 117-120 von Listing 8, was analog zum späteren Ablauf der Zuweisung ist, bei der erst in Zeile 122 tatsächlich die Ranks festgelegt werden. Die, im Gegensatz zu Listing 6, geänderten Zeilen sind die Zeilen 116 bis 120 und 124, sowie 125. Zum Speichern der Zuordnung zwischen dem Tupel (Rank, "jobid") wird eine Hashtabelle auf Seiten des Agenten genutzt. Schlussendlich geschieht die jeweilige Zuordnung durch eine Abfrage an den Agenten für jeden Prozess einzeln, der das entsprechende Tupel für den Prozess in seiner Hashtabelle sucht und den korrigierten Rank jeweils zurückschickt, was OpenMPI wiederum, in Zeile 127, einträgt.

```
88 int ompi_comm_init(void)
89 {
...
114     group->grp_proc_count = size;

116     // transfer the infos about the current processes to the agent
117     for (size_t i = 0 ; i < size ; ++i) {
118         opal_process_name_t name = {.vpid = i, .jobid = OMPI_PROC_MY_NAME
    ->jobid};
119         getProcessAgentRank(name.jobid, name.vpid, size);
120     }

122     for (size_t i = 0 ; i < size ; ++i) {
123         opal_process_name_t name = {.vpid = i, .jobid = OMPI_PROC_MY_NAME
    ->jobid};
124         /* get desired rank from agent */
125         name.vpid = getProcessAgentRank(name.jobid, name.vpid, size);
126         /* look for existing ompi_proc_t that matches this name */
127         group->grp_proc_pointers[i] = (ompi_proc_t *) ompi_proc_lookup (
    name);
...
    }
...
}
```

Listing 8: ompi/communicator/comm_init.c

Damit ist das Tauschen der Ranks innerhalb einer MPI_COMM_WORLD nun durchführbar. Um zu zeigen, dass eine beliebige Rank-Reihenfolge möglich ist, werden im Agenten des Prototyps die Ranks absteigend, anstatt standardmäßig aufsteigend, vergeben. Damit ist gezeigt, dass die Implementierung auch dann funktioniert, wenn der erste Prozess nicht den Rank Null zugewiesen bekommt (was ansonsten, wie oben beschrieben, immer zu einem Fehler führen würde) und es besonders einfach und vor allem übersichtlich zu implementieren ist, da sich der Agent so nur den Index merken muss, mit dem er den Rank des nächsten Prozesses festlegen wird.

6.3 Testaufbau

Zum Prüfen der Funktionstüchtigkeit des Prototypen war es notwendig einige kurze Testprogramme zu erstellen, die die verschiedenen Fälle abdecken, mit denen der Prototyp umgehen können soll. Dabei musste nur geprüft werden, dass die durch den Agenten vergebenen Ranks

auch korrekt sind, also im validen Bereich liegen und konsistent sind. Wenn das nicht der Fall wäre, also z. B. Ranks in einer MPI_COMM_WORLD doppelt oder nicht vergeben werden würden, dann würde das zur Laufzeit meist schon bei der Initialisierung zu Fehlern führen, wie es unter anderem in Kapitel 5.3 erläutert ist. Bei den Fällen, die das Testen abdecken soll, handelt es sich um die verschiedenen Arten der Prozesserzeugung mit MPI, wie sie im Kapitel 4.2 näher erläutert sind: Die initiale Erzeugung bei Programmstart und die dynamische Erzeugung mittels `MPI_Comm_spawn` und `MPI_Comm_spawn_multiple` . Insbesondere bei der dynamischen Prozesserzeugung musste sichergestellt werden, dass Ranks der verschiedenen MPI_COMM_WORLDs vom Agenten unterschieden werden können: Wenn beispielsweise mehrfach der Befehl `MPI_Comm_spawn` einen einzelnen neuen Prozess erzeugen sollte, sodass jeder seine eigene MPI_COMM_WORLD hat, sollte jeder neue Prozess auch wieder den Rank null besitzen (siehe Kp. 4.2.2). Anfangs wurde für die Tests nur ein triviales "Hello World"-Programm genutzt, welches die Informationen über den aktuellen Prozess auf die Konsole ausgab, die dem Agenten zugesendet werden sollten und welches die verschiedenen Fälle zur Prozesserzeugung implementierte. Schlussendlich wurde auch mit Hilfe einer N-Body-Simulations-Implementierung, wie sie auch in Kapitel 1 beschrieben ist, geprüft, dass der Prototyp nicht nur für den trivialen Fall eines "Hello World"-Programms funktioniert, sondern auch für komplexere Anwendungen, in der Prozesse miteinander kommunizieren müssen.

```
Spawned - PID: 677733, vpid: 1, jobID: 1033895937, size: 2
Found Process: 1033895937:1 -> 0
Send Rank: 0
```

Listing 9: Beispielausgabe des Agenten (bzw. Servers) bzgl. einer Anfrage durch einen OpenMPI-Prozess

```
Received from server: 0
Set Rank: 0
Hello, world, I am 0 of 2, PID: 677733
```

Listing 10: Beispielausgabe des "Hello World"-OpenMPI-Programms (bzw. Clients) eines einzelnen Prozesses

6.4 Identifikation des Knotens

Da zum Zeitpunkt der Erstellung der Arbeit nicht die Ressourcen zur Verfügung standen, um den Prototypen auf einem verteilten System zu testen (welches einem echten HPC-System näher kommen würde), ist die Identifizierung des Knotens im HPC-System als theoretische Überlegung verfasst und ist nicht in der Implementierung des Prototypen berücksichtigt:

Ist ein Prozess auf einem Knoten eindeutig identifiziert, stellt sich noch die Frage der Identifikation des Knotens selbst im HPC-System, damit auch ein zentrales Verwaltungssystem des HPC-Systems jeden Prozess eindeutig zuordnen kann. Dieses Problem lässt sich relativ einfach lösen: Da jeder Agent (wie er in Kapitel 6.2 beschrieben ist) weiß, auf welchem Knoten er sich befindet bzw. auf Informationen über den Knoten Zugriff hat, kann dieser bei Kommunikation mit dem zentralen Verwaltungssystem immer angeben in welchem Kontext (bzgl. welchem Knoten) seine Angaben gegeben werden. Als Identifikator eines Knotens könnte man System-Informationen nutzen, wie sie beispielsweise mit dem Befehl `uname` auf einem Linux-System auslesbar sind, insofern diese in Kombination eindeutig für jeden Knoten sind. Alternativ könnte auch die Netzwerkadresse im Netzwerk des HPC-Systems genutzt werden, die auch eindeutig für jeden Knoten sein muss. Die Kombination aus einem solchen Identifikator für den Knoten und der Identifikation der Prozesse durch den Prototypen, sollte hinreichend sein für eine systemweite, eindeutige Identifikation von Prozessen.
Für eine Umsetzung dieser Überlegung müsste der Prototyp noch um eine Schnittstelle erweitert werden, die mit einem solchen zentralen Verwaltungssystem, direkt oder indirekt, kommuniziert.

7 Auswertung

Um das Ergebnis dieser Arbeit bewerten zu können, ist es erst notwendig zu analysieren, welche der Anforderungen bereits durch OpenMPI selbst gelöst werden. Grundsätzlich bietet OpenMPI, mit den in Kapitel 5.1 beschriebenen Hostfiles und Rankfiles, die Möglichkeit Prozessen mit bestimmten Ranks bestimmte Hardware-Ressourcen zuzuteilen. Ob diese spezifische Zuteilung, wie sie in den beschriebenen Rankfiles definiert ist, auch immer garantiert ist, ist Gegenstand weiterer Untersuchungen.
Allerdings sind die Zuweisungen der Rankfiles ausschließlich auf initial erzeugte Prozesse anwendbar, auch, da diese immer relativ zur initialen MPI_COMM_WORLD sind. Möchte man auch entsprechend granulare Kontrolle über die Zuweisung der Hardware-Ressourcen von dynamisch erzeugten Prozessen, auf die in Kapitel 4.2.2 für MPI näher eingegangen wird, haben, bietet OpenMPI für diesen Anwendungsfall keine Lösung an. Insofern bietet die in dieser Arbeit vorgestellte Lösung zumindest auch eine Kontrolle über Ranks von dynamisch erzeugten Prozessen zur Laufzeit und damit potentiell einen Mehrwert gegenüber der unmodifizierten OpenMPI-Implementierung.
Zusätzlich werden dem Agenten bzw. dem Verwaltungssystem auch weitere Daten über die gestarteten Prozesse übergeben, wie z. B. die PID (im lokalen Linux Betriebssystem), die für das Verwaltungssystem wichtig ist, um Prozesse als OpenMPI-Prozesse zu identifizieren und das Laufzeitverhalten dieser Prozesse zu beobachten oder sie auch gegebenenfalls automatisch anhalten zu können, sollten sich die Prozesse außerhalb ihrer Anforderungen bewegen, indem sie beispielsweise ihre maximale Laufzeit überschreiten. Des Weiteren sind die Laufzeitinformationen auch relevant für eine Modellierung des Programmverhaltens im Nachhinein. Wie in Kapitel 2.2 beschrieben, benötigt ein Planungsbasierter Scheduler Vorhersagemodelle und historische Daten einer Anwendung, um das Laufzeitverhalten und die Ressourcenbelegung der Anwendung vorhersagen zu können. Diese Modelle setzen sich zumeist aus früheren Programmläufen des gleichen Programms mit ähnlichen Eingaben und Parametern zusammen und je präziser die Informationen über den Programmverlauf sind, desto besser ist das Modell und damit die Vorhersage des Planungsbasierten Schedulers, was wiederum eine effizientere Verteilung der Anwendungen auf dem entsprechenden HPC-System bewirkt.
Das Permutieren der Zuordnung von Prozess zu Rank funktioniert zumindest auf einem lokalen Rechner für den Testaufbau aus Kapitel 6.3 mit der prototypischen, modifizierten OpenMPI-Version. Die Vertauschung von Ranks ist deswegen interessant, weil die Last einer Anwendung meist nicht gleichverteilt über alle Ranks ist. Das heißt, dass der Rechenaufwand für jeden Prozess einer Anwendung von seinem Rank abhängig sein kann. Hat das Verwaltungssystem ein Vorhersagemodell über den Verlauf der Anwendung, kann es entscheiden welcher Knoten und Prozessor die höhere Last tragen soll, was insbesondere dann von Relevanz ist, wenn es

sich um ein heterogenes HPC-System handelt, welches nicht aus vielen identischen Knoten besteht, sondern aus Knoten mit unterschiedlichen Eigenschaften und Ressourcen, aus denen verschiedene Rechenleistungen resultieren. So könnte das Verwaltungssystem, auch zur Laufzeit, entscheiden, dass ein Prozess mit voraussichtlich besonders hoher Last auch auf einem Prozessor mit besonders hoher Leistung ausgeführt werden soll. Inwiefern die Änderung des Ranks Auswirkungen auf die Zuweisung der Ressourcen, also auch auf den Ort der Erzeugung des Prozesses, hat, ist allerdings noch nicht geklärt. Die Überprüfung des Prototyps an einem realen HPC-System ist somit auch Gegenstand weiterer Untersuchungen, welche aufgrund fehlender Ressourcen noch nicht durchgeführt werden konnten.

7.1 Randbedingungen

Unter anderem aufgrund der Natur eines prototypischen Programms und aufgrund bestimmter Vorgaben durch beispielsweise die MPI-Spezifikation, ist die hier vorgestellte Lösung zwar hinreichend bezüglich der Anforderung, bringt aber einige Randbedingungen mit sich, über die man sich für eine Umsetzung dieses Prototypen für ein Produktivsystem Gedanken machen müsste.
Um der MPI-Spezifikation zu genügen (siehe Kapitel 4.1.1), war es nötig die Vergabe der Ranks durch das Verwaltungssystem bzw. den Agenten einzuschränken: So können Ranks nur innerhalb des gleichen OMPI-Jobs (bzw. der gleichen MPI_COMM_WORLD) geändert werden und der zu vergebende Rank selbst kann auch nicht beliebig gewählt werden, sondern muss innerhalb des Bereichs von null bis $N - 1$, für eine Anzahl von N Prozessen in der gleichen MPI_COMM_WORLD, liegen. Es ist also keine freie Vergabe sondern mehr eine Vertauschung der Ranks zwischen den Prozessen innerhalb eines OMPI-Jobs.
Des Weiteren beinhaltet die Implementierung, auch aufgrund des Prototypen-Status, keine Sicherheitsmechanismen, die für einen tatsächlichen Anwendungsfall vermutlich notwendig wären: Es ist beispielsweise grundsätzlich möglich durch den Agenten einen Rank innerhalb einer MPI_COMM_WORLD doppelt, gar nicht oder außerhalb der erlaubten Grenzen zu vergeben. Da es innerhalb der Implementierung auf Seiten von OpenMPI nicht abgefangen wird und OpenMPI keine Eingaben außerhalb der Spezifikation von MPI erwartet (die normalerweise auch nicht vom Nutzer so weitreichend beeinflusst werden können), würden solche Konfigurationen bereits bei der Initialisierung zu schwerwiegenden Fehler und einem schlussendlichen Programmabbruch führen.
Desweiteren ist die Implementierung nicht modular und wurde als modifizierte Variante einer aktuellen OpenMPI-Version [15] umgesetzt. Jedes Update von OpenMPI erfordert somit möglicherweise eine Anpassung der Implementierung, oder könnte schlimmstenfalls auch dazu führen, dass der gewählte Lösungsansatz nicht mehr anwendbar ist, sollte sich hinreichend

viel an dem betroffenen Stellen im Quelltext ändern, die für den hier vorgestellten Prototypen modifiziert wurden. Wenn es in OpenMPI an einer Stelle, beispielsweise in der Initialisierung, dazu kommen sollte, dass anstatt dem geänderten Rank die "vpid" eines Prozesses genutzt wird (was außerhalb dieses Prototyps durchaus valide ist), dann müsste an jeder solchen Stelle eine Anfrage zu Korrektur an den Agenten geschickt werden, um sicherzustellen, dass auch der richtige Rank angesprochen wird. Eine solche problematische Nutzung ohne eine Korrektur würde zu Inkonsistenzen zwischen den Ranks und den angesprochenen Prozessen führen, was vielfältige und auch schwerwiegende Folgen haben kann.
Da sich diese Arbeit, wie in Kapitel 4.2.1 festgelegt, explizit nur auf MPI unter der Verwendung des Initialisierungsmodells "World Model" bezieht, ist dieser Prototyp nicht kompatibel mit dem alternativen Initialisierungsmodell "Session Mode", da es keine Definition einer MPI_COMM_WORLD garantiert.
Da die Kommunikation zwischen den einzelnen, teilnehmenden Prozessen und dem Agenten nur bei der Initialisierung von Prozessen, also einmal pro Prozess, passiert, sollte die Effizienz der Implementierung eine untergeordnete Rolle spielen. Sollte aber ein Programm sehr oft neue Prozesse erschaffen, könnte die Antwortzeit des Agenten und die Umsetzung der Kommunikation dennoch von Nachteil sein: So stellt jeder Prozess die Anfragen an den Agenten einzeln. Da jeder Prozess die korrigierten Ranks für jeden anderen Prozess mit anfragt, ergibt sich die Laufzeit $O(n^2)$ für n Prozesse im gleichen OMPI-Job (bzw. der gleichen MPI_COMM_WORLD), da der Agent die Anfragen auch nur sequenziell beantwortet. Effizienter wäre es vermutlich, wenn die gespeicherten "vpid"-Werte in den darunterliegenden Hashtabellen modifiziert werden würden, unter der Voraussetzung, dass diese Hashtabellen nur einmal festgelegt und dann die beinhalteten Informationen an alle Prozesse verteilt werden. Das wäre aber ein noch tieferer Eingriff in das OpenMPI-System und würde ein weitreichendes Verständnis über die Konsequenzen einer solchen Änderung erfordern. Des Weiteren wäre es wahrscheinlich außerdem möglich den Agenten zu parallelisieren, damit die Anfragen an die korrigierten Ranks gleichzeitig von mehreren Prozessen getätigt werden können, auch wenn der Teil für die initiale Eintragung vermutlich weiterhin sequenziell ablaufen müsste, was aber von der Implementierung des Verwaltungssystems abhängig ist, welches die Ranks zuweist.

8 Fazit

Die Anforderung der eindeutigen Identifizierung von MPI-Prozessen auf MPI-Knoten mit einem Linux-Betriebssystem und der Zuweisung von Ranks an die MPI-Prozesse ist durch den Prototypen implementiert und funktioniert mindestens für die in Kapitel 6.3 beschriebenen Testfälle, die sowohl die initiale, als auch die dynamische Erzeugung von Prozessen abdecken. Ein Teil dieser Anforderungen, im Hinblick auf die Zuweisung, ließ sich auch bereits mit Hilfe eines Rankfiles (siehe Kapitel 5.1) lösen, aber insbesondere dynamisch erzeugte Prozesse lassen sich damit nicht kontrollieren.

9 Ausblick

Zunächst ist eine Überprüfung des Prototyps an einem realen HPC-System Gegenstand weiterer Untersuchungen, da der Prototyp bis jetzt noch ausschließlich auf einem lokalen OpenMPI-Knoten durchgeführt wurde. Hinsichtlich einer Umsetzung des Prototypen wurden in Kapitel 7.1 bereits einige mögliche Optimierungen der bestehenden Lösung angesprochen, die mindestens geändert werden müssten, um sie in einem potentiellen Produktivsystem einsetzen zu können. Auch müssten die theoretischen Überlegungen aus Kapitel 6.4 in die Implementierung mit einfließen, um eine Kontrolle durch ein zentrales Verwaltungssystem möglich zu machen.
Des Weiteren ist das Thema des Rankings von Prozessen in OpenMPI mit dieser Arbeit nicht erschöpfend behandelt: Beispielsweise ist es noch offen, inwieweit die Rankfiles die Zuteilung der Prozesse auf die Hardware-Ressourcen garantieren und die Änderung des Ranks, durch den hier vorgestellten Prototypen, Auswirkungen auf die Zuweisung der Prozesse auf die physischen Ressourcen des HPC-Systems hat.
Auf lange Sicht besteht die Aussicht, dass mit den gewonnen Laufzeitinformationen Vorhersagemodelle für den in Kapitel 2.2 beschriebenen Planungsbasierten Scheduler verbessert werden können, beispielsweise durch die Ergebnisse dieser Arbeit bezüglich der Prozesse, aber z. B. auch durch die Arbeit von Kovacs [10] bezüglich der Kommunikation und vielen weiteren und voraussichtlich noch kommenden Untersuchungen, die zum Verständnis des Programmablaufs und Ressourcenverbrauchs (insbesondere zur Laufzeit) in HPC-Systemen beitragen. Das System legt dann, mit Hilfe von Vorhersagemodellen und historischen Daten, die Konfiguration (z. B. den Startzeitpunkt der Anwendung, die konkrete Ressourcenbelegung, etc.) fest, die im Hinblick auf mögliche Deadlines, verfügbare Ressourcen und andere Anwendungen, voraussichtlich durchführbar ist. Damit könnten im Optimalfall jedem Nutzer Antwortzeiten für seine Anwendungen garantiert und somit die Arbeit mit HPC-Systemen schlussendlich weitaus

effizienter und gut planbar werden.

Literatur

[1] Inc Advanced Micro Devices. *AMD EPYC™ 7763*. 2021. URL: https://www.amd.com/de/products/cpu/amd-epyc-7763 (besucht am 30. 06. 2021).

[2] Dr. Roman Bansen et al. *Bedeutung von High Performance Computing im Unternehmen*. 2020. URL: https://www.bitkom.org/sites/default/files/2020-10/20201001_high_performance_computing_im_unternehmen.pdf (besucht am 30. 06. 2021).

[3] Johannes Albrecht u. a. „A Roadmap for HEP Software and Computing R&D for the 2020s“. In: *Computing and Software for Big Science* 3.1 (März 2019). ISSN: 2510-2044. DOI: 10.1007/s41781-018-0018-8. URL: http://dx.doi.org/10.1007/s41781-018-0018-8.

[4] CERN. *CERN Batch Service User Guide - Linux HPC - Overview*. o.D. URL: https://batchdocs.web.cern.ch/linuxhpc/index.html (besucht am 10. 09. 2021).

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface StandardVersion 4.0*. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (besucht am 30. 06. 2021).

[6] Message Passing Interface Forum. *Official Votes for MPI-2*. 1996. URL: https://www.mpi-forum.org/mpi-20/ (besucht am 30. 06. 2021).

[7] I. Foster und C. Kesselman. *The grid: Blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558-604-758.

[8] Kelvin Glaß. „Plan Based Thread Scheduling on HPC Nodes“. Magisterarb. Berlin, DE: Freie Universität Berlin, Institut für Informatik, März 2018.

[9] John L. Gustafson. „Reevaluating Amdahl's Law“. In: *Commun. ACM* 31.5 (Mai 1988), S. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: https://doi.org/10.1145/42411.42415.

[10] Fabian Kovacs. „Modelling Communication behaviour of Parallel Programs“. Magisterarb. Berlin, DE: Freie Universität Berlin, Institut für Informatik, Nov. 2018.

[11] OLCF at ORNL. *File:Summit Supercomputer 2018.jpg*. 2018-04-05. URL: https://commons.wikimedia.org/wiki/File:Summit_Supercomputer_2018.jpg (besucht am 19. 09. 2021).

[12] The Open MPI Project. *FAQ: General run-time tuning*. 2019. URL: https://www.open-mpi.org/faq/?category=tuning (besucht am 30. 06. 2021).

[13] The Open MPI Project. *MPI: A Message-Passing Interface StandardVersion 4.0*. 2021. URL: https://www.open-mpi.org/ (besucht am 30. 06. 2021).

[14] The Open MPI Project. *mpirun(1) man page (version 4.1.1)*. 2021. URL: https://www.open-mpi.org/doc/current/man1/mpirun.1.php (besucht am 30. 06. 2021).

[15] The Open MPI Project. *ompi*. https://github.com/open-mpi/ompi/tree/060f129a6fe963ae84dd0a4c5e51cc090eb95090. Apr. 2021.

[16] D. Reinert u. a. „DWD Database Reference for the Global and Regional ICON and ICON-EPS Forecasting System". In: *Research and Development at DWD* 2.1.5 (Sep. 2021). DOI: 10.5676/DWD_pub/nwv/icon_2.1.5. URL: https://www.dwd.de/DWD/forschung/nwv/fepub/icon_database_main.pdf (besucht am 19. 09. 2021).

[17] TOP500.org. *List Statistics*. 2021-06. URL: https://www.top500.org/statistics/list/ (besucht am 19. 09. 2021).

[18] TOP500.org. *Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D*. 2020. URL: https://www.top500.org/system/179807/ (besucht am 30. 06. 2021).

[19] Deutscher Wetterdienst. *Datenverarbeitung, DMRZ*. o.D. URL: https://www.dwd.de/DE/derdwd/it/_functions/Teasergroup/datenverarbeitung.html?nn=20256#doc391686bodyText3 (besucht am 30. 06. 2021).

[20] Deutscher Wetterdienst. *Globalmodell ICON*. o.D. URL: https://www.dwd.de/DE/forschung/wettervorhersage/num_modellierung/01_num_vorhersagemodelle/icon_beschreibung.html (besucht am 30. 06. 2021).

[21] Dale Van Zante und Jay Horowitz. *File:Jet engine blades simulation.jpg*. 2009-08-30. URL: https://commons.wikimedia.org/wiki/File:Summit_Supercomputer_2018.jpg (besucht am 19. 09. 2021).

Abbildungsverzeichnis

1 Beispieldarstellung für ein HPC-System: "Summit" [11] 3
2 Darstellung der Simulation einer Turbine mit nach statischem Druck eingefärbten Blättern, [21] 5
3 Darstellung des horizontalen Dreieckgitters des globalen ICON-Modells des DWD mit einer Verfeinerung des Gitters bzw. erhöhter Auflösung über Europa, [16, S. i] 6