UNIKASSEL
VERSITÄT

Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik

Ressourcenelastizität für das task-basierte parallele Programmiersystem APGAS

Bachelorarbeit

im Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik
der Universität Kassel

Eingereicht von: Raoul W. Goebel
Anschrift: Heideweg 44
34131 Kassel

Matrikelnummer: 30200760
E-Mail: ash@ashatar.org

Vorgelegt im: Fachgebiet Programmiersprachen/-methodik

Erstprüfer: Prof. Dr. Claudia Fohry
Zweitprüferin: Prof. Dr. Oliver Hohlfeld

Betreuer: Dr. Jonas Posner

Eingereicht am: 24. Januar 2024

Inhaltsverzeichnis

Abkürzungsverzeichnis

APGAS - Asynchronous Partitioned Global Address Space

CPU - Central Processing Unit

DMR - Dynamic Management of Resources

ESP - Effective System Performance

GLB - Global Load Balancing

HPC - High Performance Computing

JVM - Java Virtual Machine

MPI - Message Passing Interface

RMS - Ressource Management System

SLURM - Simple Linux Utility for Resource Management

SSH - Secure Shell

1 Einleitung

Berechnungsaufgaben aus diversen Bereichen haben einen hohen Rechenaufwand, zum Beispiel aus den Bereichen Quantenmechanik, Wetterberechnung sowie Simulationen aus dem Bereich der Physik und Chemie. Diese Berechnungsaufgaben werden oftmals per Supercomputer gelöst. Sie gehören dem Bereich des *High Performance Computing* (HPC) an, in welchem unter anderem *Cluster* eingesetzt werden [13]. Bei einem Cluster werden Daten zwischen den Computern (*Nodes*) des Clusters via Netzwerk übertragen. Die Nodes sind unabhängige Systeme mit eigenem Betriebssystem. Cluster sind extern per *Secure Shell* (SSH)-Verbindung erreichbar.Der Start eines Programms erfolgt nicht direkt auf den Nodes, sondern es läuft auf dem Cluster ein *Scheduler*, der *Jobs* annimmt. Auf dem genutzten Cluster ist dies der *Simple Linux Utility for Resource Management* (SLURM) Scheduler [6, 22]. Ein Job ist ein Auftrag an den Scheduler, ein zu berechnendes Programm zu starten, Jobs werden per *Batch-Script* mit den nötigen Programm- sowie Job-Parametern, wie beispielsweise der Node Anzahl, an den Scheduler übergeben (*submittet*) und dort in eine Warteschlange eingereiht. Der Scheduler startet einen Job, wenn ausreichend Nodes verfügbar sind. Falls nicht ausreichend Nodes verfügbar sind, bleibt der Job in der Warteschlange, bis ausreichend Nodes verfügbar sind [2, 5, 14, 26].
Der aktuelle Stand der Technik sind Scheduler, welche Jobs mit einer fixen Anzahl an Nodes starten und diese Anzahl bleibt zur Laufzeit gleich. Dieses Verhalten wird *Rigid* genannt. Bei rigid Jobs kann nach dem Start des Jobs die Anzahl der dem Job zugewiesenen Nodes nicht mehr angepasst werden. Dieser Umstand birgt zwei Probleme: (1) Es können trotz freier Nodes im Cluster und voller Auslastung der dem Job zugewiesenen Nodes keine weiteren Nodes zugebucht werden, um die Berechnung zu beschleunigen. (2) Es kann passieren, dass die reservierten Nodes im Verlauf der Berechnung nicht zu jeder Zeit ausgelastet sind, zum Beispiel bei einem nicht parallelisierbaren Programmabschnitt. Dennoch ist es nicht möglich die unausgelasteten Nodes freizugeben. Diese Probleme treten bei irregulärem *Workload*, d. h. wenn zu verschiedenen Abschnitten der Laufzeit die Berechnung verschieden viel Auslasung generiert, mit deutlich höherer Wahrscheinlichkeit auf, als bei regulärem Workload. Reservierte, aber kaum ausgelastete Nodes, sind nachteilig

für das gesamte Cluster. Die vollkommene Auslastung aller Nodes eines rigid Jobs, ohne Möglichkeit weitere Nodes zu reservieren, ist nachteilig für den betroffenen Job, da der Job mit mehr Nodes schneller abgeschlossen werden könnte. Dies ist auch nachteilig für die Auslastung des Clusters, da freie Ressourcen so ungenutzt bleiben können.
Damit ein Problem von vielen Nodes gleichzeitig berechnet werden kann, muss es in Teilaufgaben unterteilt werden. In einigen Programmiermodellen werden diese Teilaufgaben als *Tasks* bezeichnet. Die Unterteilung wird vom genutzten Programmiersystem vorgenommen. Eine Task besteht aus einer Operation und zugehörigen Daten, auf welchen die Operation ausgeführt wird. Es existieren diverse task-basierte parallele Programmiersysteme, zum Beispiel *X10* [1, 7], *Go*, *Threading Building BLocks* [25] und *Charm++* [23] für C++ sowie die *APGAS* (Asynchronous Partitioned Global Address Space) Bibliothek für Java [1, 37], welche in dieser Arbeit genutzt wird.
Der Begriff der Elastizität beschreibt in diesem Kontext, wie anpassungsfähig ein Job bzw. ein Scheduler ist. Es existieren verschiedene Grade von Elastizität, diese lauten *rigid*, *moldable*, *malleable* bzw. *evolving* [15]. Moldable Jobs werden mit einer gewünschten minimalen und maximalen Anzahl an Nodes submittet. Der Scheduler entscheidet dann mit welcher Node-Anzahl er den Job startet. Sobald der Job gestartet ist, bleibt die Node-Anzahl konstant. Malleable Jobs werden ebenfalls mit minimaler und maximaler Nodeanzahl submittet. Zudem kann die Nodeanzahl zur Laufzeit durch den Scheduler angepasst werden. Evolving Jobs sind weitestgehend vergleichbar mit malleable Jobs, sie unterscheiden sich von malleable Jobs dadurch, dass sie in der Lage sind in Zusammenarbeit mit dem Scheduler ihre Node-Anzahl selbstständig dynamisch zur Laufzeit anzupassen.
Um der Problematik von Leerlauf und Ressourcenmangel entgegenzuwirken, können bei task-basierter Parallelisierung zwei Methoden eingesetzt werden; die Lastverteilung sowie die evolving Funktionalität. Das Ziel der Lastverteilung ist eine ausgewogene Verteilung der Tasks auf den verfügbaren Nodes um den Leerlauf eines Nodes zu verhindern. Durch *Workstealing* werden die Tasks zwischen den Nodes eines Jobs verteilt [29]. Im Rahmen dieser Arbeit wird die *Lifeline-Based Global Load Balancing*-Bibliothek (GLB) [4] zur Lastverteilung und die *APGAS*-Bibliothek [3] für Java zur Parallelisierung genutzt. APGAS startet auf den Nodes *Places*, welche die Berechnungen ausführen. Aktuell unterstützen APGAS und GLB bereits *Malleability* [16].
Die Zielsetzung der vorliegenden Bachelorarbeit ist die Erweiterung von APGAS und GLB um die Evolving-Funktionalität. Durch die Evolving-Funktionalität ist es APGAS möglich die Anzahl der Places ohne manuellen Eingriff des Programmierers zur Laufzeit anzupassen.

Für diese Automatisierung wird die Last während der Laufzeit stetig festgestellt (Lastsammlung). Die Lastsammlung wird dabei von APGAS vorgenommen. Die CPU-basierte Lastsammlung ausschließlich in APGAS und die task-basierte Lastsammlung in Kooperation mit GLB. Die Daten der Lastsammlung werden innerhalb APGAS ausgewertet und es wird entschieden, ob neue Places gestartet oder Places heruntergefahren werden sollen (Lastauswertung).
Zur Evaluation der Evolving-Funktionalität wurde ein vorhandener dynamischer synthetischer Benchmark erweitert. Die erweiterte Variante des Benchmarks (`evotree`) erzeugt einen m-ären Baum, an einem Blatt der letzten Ebene wird dann ein Ast erzeugt, am Ende des Asts wird nochmals der selbe m-ären Baum wie zu Beginn erzeugt. Dabei ist jeder Knoten eine Task, welche zur Laufzeit generiert wird. Der Baum wird generiert, indem für eine definierte Zeit - inklusive einer festgelegten Varianz - jede Task eine Dummy-Berechnung durchführt. Diese Struktur führt, je nach gewählten Parametern, zu Beginn des Benchmarks zu Erhöhung der Place Anzahl bis der Ast erreicht ist. Während der Berechnung des Asts führt die Struktur zu Reduzierung der Place-Anzahl bis nur noch ein Place aktiv ist. Nach der Berechnung des Asts wieder zu Erhöhung der Place-Anzahl bis Programm beendet ist. Dies hebt den Effekt der Evolving-Funktionalität hervor.
Der `evotree` Benchmark wurde auf dem Cluster des Fachbereich 16 der Universität Kassel ausgeführt. Bei der Nutzung des Benchmarks wurde bei doppelt so vielen reservierten Nodes, wie Places zu Programmbeginn, ein *Speedup* bis zu 21% im Vergleich zu dem selben Programm ohne Evolving-Funktionalität gemessen. Beide implementierten Varianten der Lastsammlung schnitten in etwa gleich gut ab.

Die Arbeit ist wie folgt unterteilt:
In Kapitel 2 wird auf die nötigen Grundlagen eingegangen. Kapitel 3 stellt die wichtigsten Konzepte von APGAS und GLB vor. Kapitel 4 stellt die Implementierung der Evolving-Funktionalität vor. Kapitel 5 beinhaltet Informationen zur Messumgebung, die Beschreibung des Benchmarks und die Messergebnisse sowie deren Auswertung. Im Kapitel 6 werden zusammenführend verwandte Arbeiten vorgestellt. Kapitel 7 fasst die Ergebnisse der vorliegenden Arbeit zusammen.

2 Grundlagen

Im diesem Kapitel wird auf die Struktur von der Hardware bis zum Programm des Nutzers, sowie die Grundlagen von APGAS und GLB eingegangen.

2.1 Struktur

Die grundsätzliche Struktur von der Hardware bis zum Programm des APGAS Nutzers kann als Struktur aus verschiedene Ebenen betrachtet werden. Beginnend bei der Hardware-Ebene (dem Node), über die auf dem Node laufende *Java Virtual Machine* (JVM), der Software-Ebene. Auf dem Betriebssystem eines Nodes läuft eine JVM, die eine Java-Umgebung bereitstellt, um ein Java-Programm auszuführen. Die Java-Umgebung wird durch die APGAS-Bibliothek um Parallelisierung erweitert. Die GLB-Bibliothek erweitert diese wiederum um Lastverteilung. Der Programmierer eines parallelen Programms nutzt dann die Konstrukte von APGAS und GLB, um ein parallelisierbares Programm mit Lastverteilung zu programmieren (Nutzer-Ebene). Die beschriebenen Ebenen sind in Abbildung 2.1 grafisch zusammengefasst.

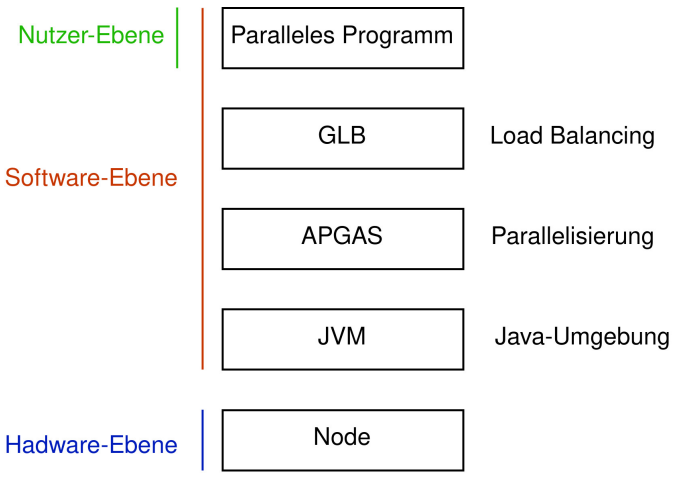

Abbildung 2.1: Ebenen von der Hardware bis zum parallelen Programm

2.2 APGAS

Die APGAS-Bibliothek für Java basiert auf dem Programmiermodell des asynchronen partitionierten globalen Adressraums (Asynchronous Partitioned Global Address Space - APGAS) [34]. In diesem Modell wird der Speicher unterschiedlicher Rechner eines Clusters als ein globaler Speicher angesehen. Da im Rahmen dieser Arbeit mit der Bibliothek APGAS für Java gearbeitet wurde, wird in dieser Arbeit die Abkürzung APGAS für die Java-Bibliothek und nicht für das Programmiermodell verwendet.
APGAS startet eine Berechnung immer auf Place 0, von dort aus werden weitere Berechnungen auf dem lokalen Place oder auf entfernten Places gestartet. Es ist möglich auf einem Node mehr als einen Place zu starten. Die Places können untereinander kommunizieren. Wenn Places verschiedener Nodes untereinander kommunizieren, geschieht dies über das Netzwerk. Durch die Asynchronizität können die Places verschiedene Abschnitte eines Programms berechnen, ohne auf den Abschluss der Berechnungen auf anderen Places warten zu müssen.
APGAS kann rigid und malleable gestartet werden. Durch die Erweiterung der Bibliothek in dieser Arbeit kann APGAS nun auch evolving gestartet werden. Wenn APGAS malleable oder evolving gestartet wird, können zur Laufzeit Places durch `shrink` heruntergefahren und durch `grow` neue Places gestartet werden.
Die Konstrukte der APGAS-Bibliothek ermöglichen es, Berechnungen auf dem lokalen Place, wie auch auf entfernten Places zu starten und diese zu synchronisieren. APGAS stellt dem Nutzer synchrone, wie auch asynchrone Konstrukte zur Parallelisierung bereit. Ein synchrones Konstrukt zur Parallelisierung ist `at`, es startet eine synchronen Task auf dem, beim Aufruf angegebenen Place. Asynchrone Konstrukte zur Parallelisierung sind: `asyncAt` sowie erweiterte Varianten `uncountedAsyncAt` und `immediateAsyncAt`. `asyncAt` startet eine asynchrone Berechnung auf dem beim Aufruf angegebenen Place. `uncountedAsyncAt` startet eine Berechnung auf dem beim Aufruf angegebenen Place, ohne den Abschluss der Berechnung zu verfolgen. `immediateAsyncAt` ist in seiner Funktionalität ähnlich wie `uncountedAsyncAt`, aber um sofort kurzfristige Berechnungen auf einem Place zu starten. Wenn auf den Abschluss der Bearbeitung asynchroner Aufrufe gewartet werden soll, müssen die Aufrufe innerhalb eines `finish`-Blocks gestartet werden. Eine Berechnung besteht aus einem Konstrukt der APGAS-Bibliothek, z. B. ein `asyncAt`, und einem Teil Quelltext des Nutzers.

Es folgt ein Beispiel eines einfachen parallelen Programms, zu sehen im Programmausdruck 2.1.

Programmausdruck 2.1: APGAS-Beispiel

```
1  class APGAS_Basics{
2      System.out.println("This is " + here());
3      at(place(1) - > System.out.println("This is " + here()););
4
5      finish (() -> {
6          for(Place place: places()){
7              asyncAt(() -> {
8                  System.out.println("Hey from " + here());
9              });
10         )
11     });
12     System.out.println("Basics finished.");
13 }
```

Dieses Programm-Beispiel gibt die Places aus, auf welchen die Ausgabe verarbeitet wird. Beginnend auf Place 0 mit der Ausgabe `This is place 0`, dann die Ausgabe `This is place 1` synchron auf Place 1, was bedeutet, dass dieser Abschnitt auf einem anderen Place aber dennoch sequenziell ausgeführt wird. Darauf folgt eine asynchrone Ausgabe `Hey from place 0`, `Hey from place 1` usw. auf allen aktuell existierenden Places. Diese ist umschlossen von einem `finish`, wodurch auf den Abschluss aller `async`-Aufrufe auf allen Places im `finish` gewartet wird, bevor `Basics finished.` ausgegeben werden kann. Dieses Beispiel könnte bei drei Places die in Programmausdruck 2.2 gezeigte Ausgabe liefern - wobei die Ausgaben von `Hey from place X` in zufälliger Reihenfolge auftreten können, da diese asynchron bearbeitet werden.

Programmausdruck 2.2: APGAS Beispiel Ausgabe

```
This is place 0
This is place 1
Hey from place 2
Hey from place 0
Hey from place 1
Basics finished.
```

2.3 GLB

Dieser Abschnitt erläutert den Aufbau und Ablauf sowie den Grund der Nutzung von GLB in der vorliegenden Arbeit. GLB erweitert die APGAS-Bibliothek um Lastverteilung. In dieser Arbeit wird zur Lastverteilung die *Malleable Lifeline-Based Global Load Balancing*-Biblothek [4] genutzt. Die Ablaufbeschreibungen beziehen sich direkt auf die genutzte Variante der Lastverteilung. Diese Variante der Lastverteilung unterstützt bereits `grow` und `shrink` Aufrufe von APGAS.

Die Teilaufgaben einer Berechnung, Tasks genannt, werden einem *Worker-Thread* eines Places zugewiesen (*static*) oder zur Laufzeit eines Threads dort generiert (*dynamic*). Bei irregulärem Workload kann ein Place bereits nach kurzer Zeit die Berechnung abgeschlossen haben, wohingegen ein anderer Place noch lange zu rechnen hat - in einem solchen Fall findet die Lastverteilung Anwendung.

Gute Beispiele hierfür sind die *Unbalanced Tree Search*, das *N Damen Problem* sowie die Matrizenmultiplikation [30].

GLB wird genutzt, um die Berechnung auf den zur Verfügung stehenden Places effizienter zu gestalten, dies geschieht durch Verteilung von Tasks auf Places ohne Tasks. Effizienz bedeutet in diesem Kontext eine ausgewogene Verteilung von Tasks auf die Places. Eine effiziente Verteilung hat folgende Vorteile: (1) Durch eine effizientere Lastverteilung wird die Berechnung eines Jobs schneller abgeschlossen, das ist von Vorteil für den Job selbst. (2) Sie führt zu einer kürzeren Laufzeit eines Jobs, was wiederum zu einer kürzeren Wartezeit anderer Jobs führt, was einen positiven Effekt für die Effizienz des gesamten Clusters bedeutet.

Abb. 2.2 zeigt den Prozess des Work Stealing anhand eines Nodes mit zwei Workern. Auf jedem Place gibt es eine, durch Startparameter, festgelegte Anzahl an *Workern*, welche die Tasks berechnen. Jeder Worker läuft in einem eigenen Thread. Jeder Worker hat einen *Bag*, in welchem die ihm zugeordneten, zu berechnenden Tasks gespeichert werden. Solange ein Worker lebt, entnimmt er seinem Bag jeweils eine Task und berechnet diese. Tasks, die auf Abarbeitung warten und noch keinem Worker zugeteilt sind, werden in Warteschlangen eingereiht. Es gibt zwei Typen von Warteschlangen, wobei jeder Place seine eigenen Warteschlangen hat: Der erste Typ Warteschlange, die *intra queue*, wird genutzt um die Workerbags zu befüllen, falls diese leer sind; der zweite Typ ist die *inter queue* zum Teilen von Arbeit zwischen Places.

Die Worker arbeiten zuerst alle verfügbaren lokalen Tasks ab. Falls keine lokalen Tasks mehr

verfügbar sind, wird versucht von anderen Places Tasks zu erhalten. Diese Interaktion wird *work stealing* genannt. Work stealing ist eine Lastverteilungs-Technik, bei welcher Tasks zwischen Places ausgetauscht werden. Beim work stealing wird der stehlende Place *Thief* und der zu bestehlende Place *Victim* genannt. In GLB wird kooperatives Work Stealing betrieben; beim kooperativen Work Stealing fragt der Thief das Victim nach Tasks an [28]. Bei GLB hat jeder Place sogenannte *Lifeline-Buddies*, diese werden beim Work Stealing genutzt. Es wird zuerst eine konfigurierte Anzahl an zufälligen Places als Victims gewählt. Falls von all diesen Places eine Ablehnung oder keine Rückmeldung in angemessener Zeit zurückgemeldet wird, wendet sich der Thief als nächstes an seine Lifeline-Buddies und fragt diese nach zu berechnenden Tasks. Erhält er auch von diesen keine Tasks, wechselt der Place für eine Zeit in einen *idle* Zustand. Bei einem erfolgreichen Steal hingegen erhält der Thief Tasks aus der inter queue des Victims.
Die Berechnung des Programms ist abgeschlossen, sobald keine Tasks mehr in den Warteschlangen sind und keine weiteren Tasks generiert werden.

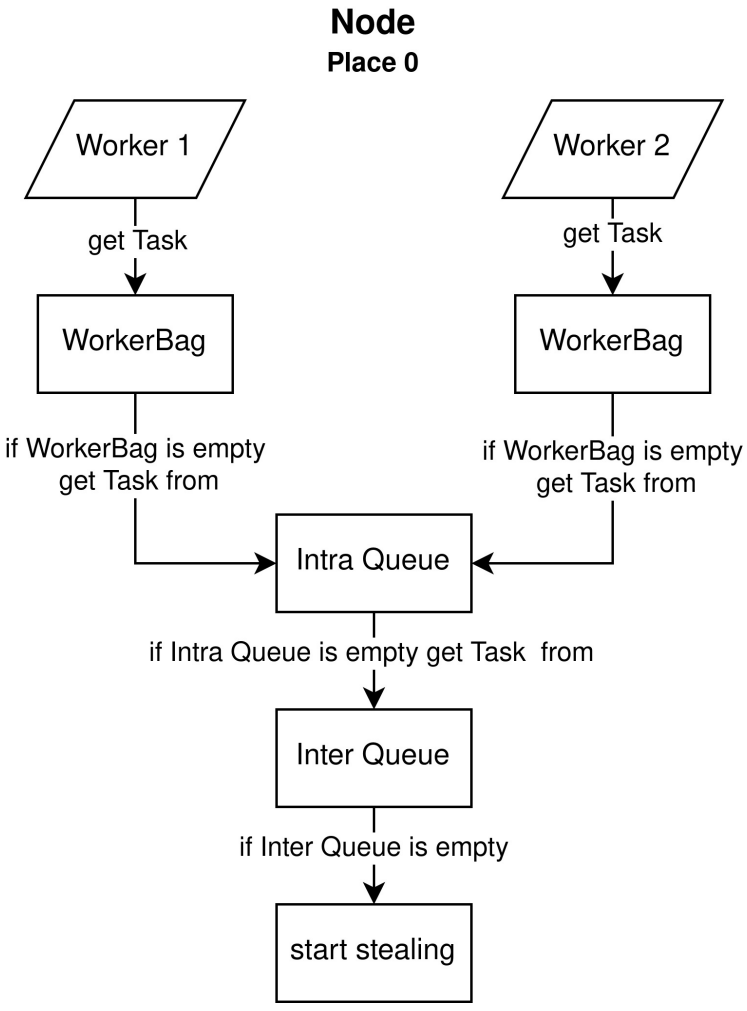

Abbildung 2.2: Task-Bearbeitung - Warteschlangen in GLB

3 Konzepte

Dieses Kapitel stellt die wichtigen Konzepte für die vorliegende Arbeit vor. Es enthält das Implementierungskonzept mit einer Übersicht der Evolving-Erweiterung.

Wenn ein Programm malleable gestartet wird, kann der Scheduler Änderungen der Place-Anzahl des Programms auslösen, welche dann in APGAS/GLB umgesetzt werden. Im Evolving-Modus nimmt APGAS anhand der konfigurierten Parameter diese Entscheidung selbst vor und setzt sie eigenständig um.
Um sinnvolle Zeitpunkte für einen `grow` oder `shrink` festzustellen, müssen diese anhand von inhärenten Faktoren des parallelen Programms des Nutzers oder anhand der Hardware der Nodes bzw. der softwareseitigen Faktoren von APGAS oder GLB festgestellt werden. Da die Arbeit sich mit dem Programmiersystem befasst und der Quelltext des parallelen Programms des Nutzers nicht bekannt ist, werden zum Feststellen der Zeitpunkte die hardwareseitigen Faktoren der Nodes sowie die softwareseitigen Faktoren von APGAS oder GLB genutzt. Es wurden zwei Varianten der Lastsammlung implementiert: Die erste Variante richtet sich nach der CPU-Last der Nodes, auf welchen das Programm ausgeführt wird - diese wird in APGAS ausgewertet. Die zweite Variante richtet sich nach dem Stand der Tasks von GLB.

Implementierungskonzept: Das Grundkonzept ist es, auszuwerten, wie hoch die Last aller Places zu einem bestimmten Zeitpunkt ist, um zu beurteilen, ob eine Erhöhung bzw. Reduzierung der Place-Anzahl zu diesem Zeitpunkt sinnvoll ist. Falls die Last aller Places hoch ist, soll die Anzahl der Places erhöht werden, um die Berechnung des Programms zu beschleunigen. Im Falle einer geringen Auslastung aller Nodes, soll der Node mit der geringsten Last freigegeben werden. Auch wenn nur ein einzelner Node über einen festgelegten Zeitraum eine geringe Auslastung hat, soll dieser heruntergefahren werden.
Ein weiteres Ziel ist es, eine Basis für automatisierte Ressourcenanpassung bereitzustellen, welche einfach um weitere Varianten der Lastsammlung erweitert werden kann.

Für die Evolving-Funktionalität ist der Verlauf der `grow`- und `shrink`-Operationen relevant, zu sehen in Abb. 3.1. Gelb markierte Schritte stammen von APGAS, grün markierte Schritte von GLB und rot markierte Schritte wurden durch die Erweiterung der vorliegenden Arbeit hinzugefügt. Die involvierten Places sind unterhalb des jeweiligen Schrittes angegeben. Wenn ein `grow` durch die Evolving-Funktionalität ausgelöst wird, wird, falls möglich, ein neuer Place gestartet. Dann informiert APGAS die alten Places über die Existenz des neuen Places. Woraufhin in GLB die *Lifelines* angepasst werden und Arbeit an den neuen Place vergeben wird. Sobald dies abgeschlossen ist, wird durch die Evolving-Funktionalität die Lastsammlung auf dem neuen Place gestartet. Wenn ein `shrink` durch die Evolving-Funktionalität ausgelöst wird, wird zuerst die Sammlung der Last auf diesem Place beendet. Dann passt GLB die Lifelines der verbleibenden Places an, wodurch keine Arbeit mehr an den Place gesendet wird. Daraufhin wird die verbleibende Arbeit des herunterzufahrenden Places auf die verbleibenden Places transferiert. APGAS informiert an dieser Stelle die verbleibenden Places über die anstehende Änderung der Place-Anzahl und fährt den Place herunter.

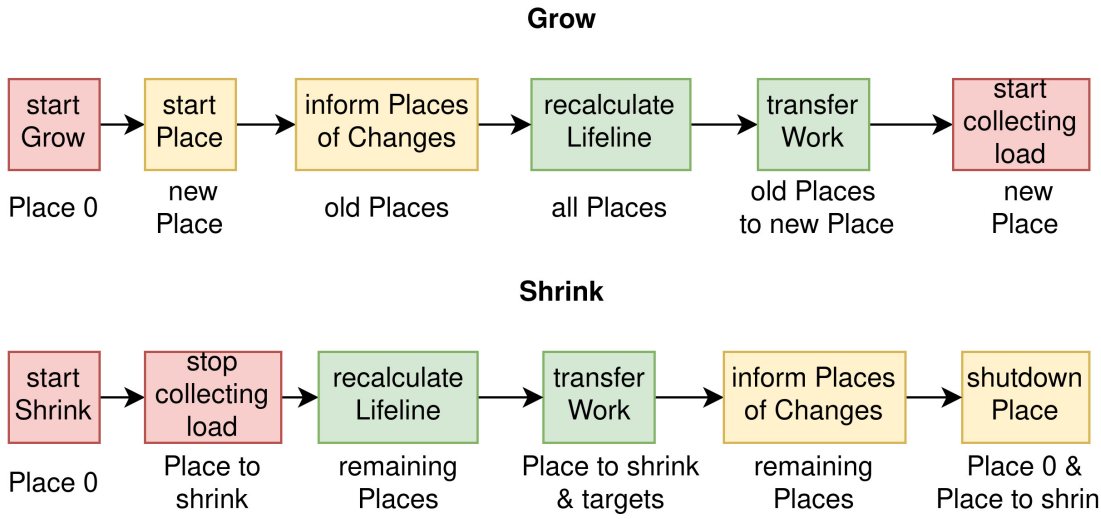

Abbildung 3.1: Schematischer Ablauf `grow`/`shrink`

Um dies umzusetzen wurden neue Klassen und ein Interface angelegt. Die Klasse `Evolving` stellt Methoden zur Auswertung und Sammlung von Last bereit. Das Interface `GetLoad` und die zugehörigen Klassen `GetCpuLoad` und `GetTaskLoad` werden zur Sammlung von Daten über die CPU- bzw. Task-Last der Places genutzt.
Die Entscheidung über `grow`- und `shrink`-Operationen findet in der Klasse `Evolving` statt, dazu werden die Methoden `loadEvaluation` und `calcMinAndAvgLoad` genutzt.

Das Flussdiagramm in Abb. 3.2 zeigt die Zusammenhänge der, für die Evolving-Funktionalität hinzugefügten, Methoden. Die Methode `evolve` ist für die Initiierung der Evolving-Funktionalität zuständig und wird nur auf Place 0 aufgerufen. Sie startet auf allen Places durch das `immediateAsyncAt` Konstrukt die Methode `startObtainPlaceLoad`, welche auf dem jeweiligen Place einen neuen Thread mit der Methode `obtainPlaceLoad` startet, der dort die Last des Places sammelt und die erhobenen Daten (Last) via `immediateAsyncAt` an Place 0 sendet. Durch die Verwendung von `immediateAsyncAt` ist der Abschnitt unabhängig vom `finish`-Block. Danach wird auf Place 0, in einem eigenen Thread die Methode `loadEvaluaion` zur Auswertung und Anpassung der Places gestartet. Diese wartet zu Beginn drei Sekunden (konfigurierbar) und ruft dann alle zwei Sekunden (konfigurierbar) die Methode `calcMinAndAvgLoad` auf, welche aus den aktuellen Werten der Last der Places den Place mit der geringsten Last sowie die durchschnittliche Last aller Places berechnet. Die Durchschnittliche Last wird genutzt, um zu entscheiden, ob ein `grow` oder `shrink` ausgeführt werden soll. Der Place mit der geringsten Last wird benötigt, um gezielt den am wenigsten ausgelasteten Place herunterzufahren. Die Auswertung auf Place 0 wird innerhalb einer Shutdown-Routine von APGAS gestoppt, kurz bevor das Programm beendet wird. Die Sammlung der Last wird gestoppt, kurz bevor der jeweilige Place heruntergefahren wird. In den folgenden Flussdiagrammen sind APGAS-Konstrukte blau hervorgehoben.

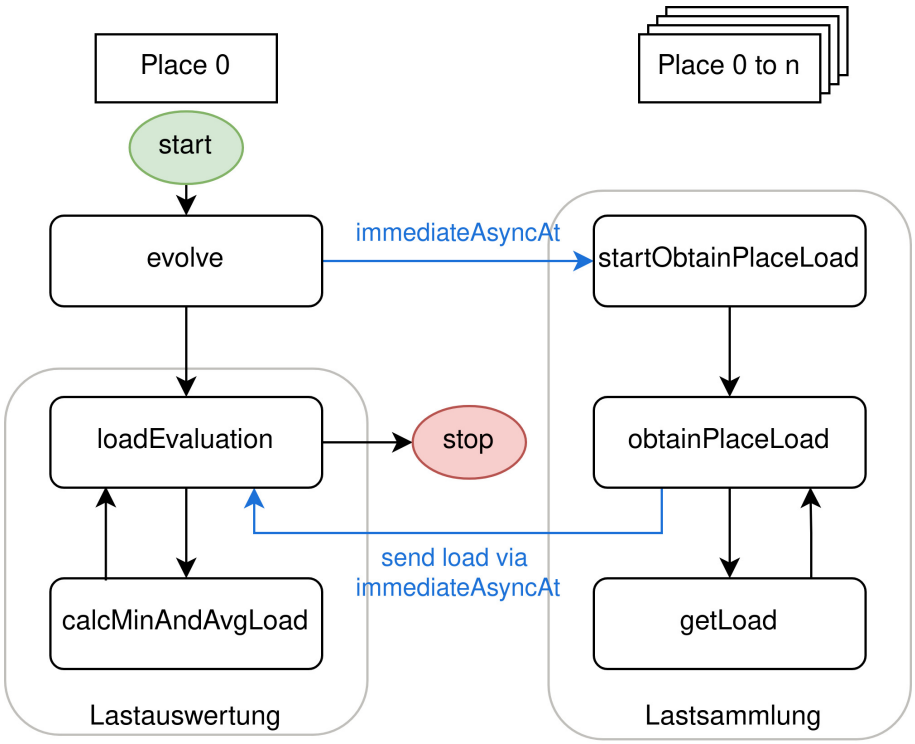

Abbildung 3.2: Flussdiagramm high-level Übersicht der Evolving-Erweiterung

4 Implementierung

Dieses Kapitel befasst sich mit der Implementierung der Ressourcenelastizität. Es ist unterteilt in Implementierung der Lastsammlung 4.1, wiederum unterteilt in CPU-basierte Lastsammlung 4.1.1 und task-basierte Lastsammlung 4.1.2. Es schließt mit der Lastauswertung 4.2.

4.1 Lastsammlung

Zuerst wird auf die Lastsammlung eingegangen, da diese für die Lastauswertung benötigt wird. Es wurde ein Interface `GetLoad` angelegt, von welchem Klassen erben, welche eine Variante der Lastsammlung implementieren. In der vorliegenden Arbeit wurden die Klassen `GetCpuLoad` und `GetTaskLoad` angelegt. Wobei `GetCpuLoad` eine APGAS-Klasse ist, welche Daten anhand der CPU-Last sammelt. `GetTaskLoad` ist eine GLB-Klasse, welche Daten anhand der Tasks sammelt. Programmierer, welche APGAS/GLB zur Parallelisierung nutzen, können eigene Klassen anlegen, welche vom Interface `GetLoad` erben, um eigene Varianten der Lastsammlung zu ermöglichen. Es wurde die Methode `obtainPlaceLoad` in der APGAS-Klasse `Evolving` angelegt, welche die Lastsammung initiiert, sie nutzt die `getLoad` Methode des Interfaces, welches je nach Konfiguration `GetCpuLoad` oder `GetTaskLoad` aufruft. Die Methode `obtainPlaceLoad` sammelt bis zum Herunterfahren eines Places die Last des Places und sendet diese zu Place 0. Die aktuellen Daten werden auf Place 0 in einer Hashmap für einen kurzen Zeitraum gespeichert, bis sie im nächsten Durchlauf überschrieben werden. Diese Hashmap ist per *Global Reference* für alle Places erreichbar. Dabei wird jedem Place anhand seiner ID ein fester Platz in der Hashmap zugewiesen.
Der schematische Ablauf der Methode ist in Abbildung 4.1 zu sehen. Die Lastsammlung und Lastauswertung ist während bereits eine `grow-` oder `shrink-` Operation ausgeführt wird, unnötig und wird daher innerhalb dieses Zeitfensters nicht aufgeführt. Die durchschnittliche Last ist durch die Anpassung der Place-Anzahl zum nächsten Auswertungszeitpunkt

voraussichtlich anders zum Zeitpunkt der letzten Auswertung. Zu Beginn wird anhand der Variable `waitingForPlaceChanges` geprüft, ob aktuell bereits auf eine Anpassung der Place-Anzahl gewartet wird. Ist dies der Fall wird geprüft, ob diese ungewöhnlich lange dauert und falls das zutrifft wird das Warten auf eine Anpassung zurückgesetzt. Durch das Rücksetzen wird erneut die Last auf den Places gesammelt und die Last auf Place 0 ausgewertet. Wird aktuell nicht bereits auf eine Anpassung der Place-Anzahl gewartet, wird die Methode `getLoad` aufgerufen, welche die Last des Places zu diesem Zeitpunkt zurückgibt. Je nachdem, welche Variante der Lastsammlung eingestellt ist, wird die Methode `getLoad` der Klasse `GetCpuLoad` oder der Klasse `GetTaskLoad` aufgerufen. Daraufhin sendet der Place per `immediateAsyncAt` seine PlaceID und die zugehörige Last via GlobalRef in eine Hashmap auf Place 0. In beiden Fällen wird danach für eine Sekunde gewartet bevor die nächste Lastsammlung gestartet wird.

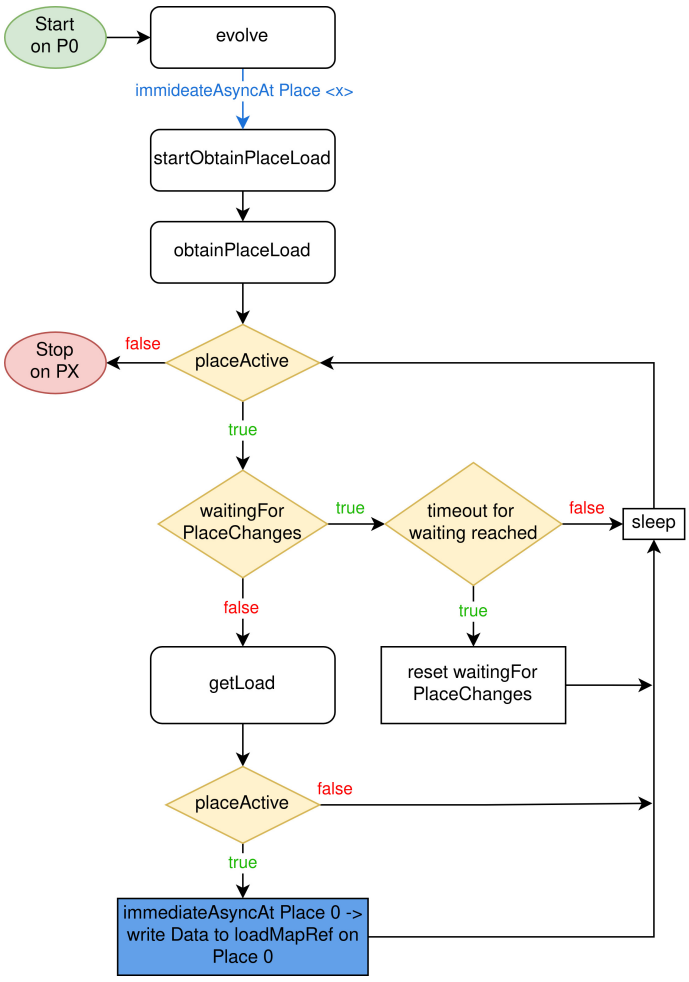

Abbildung 4.1: Flussdiagramm der Methode `obtainPlaceLoad`

4.1.1 CPU-basierte Lastsammlung

Die erste implementierte Variante der Lastsammlung sammelt Daten anhand der CPU-Auslastung. Dafür wurde in Zuge dieser Arbeit die APGAS Klasse `getCpuLoad` implementiert. Der Ablauf der CPU-basierten getLoad Methode ist dargestellt als Flussdiagramm in Abb. 4.2. Zu Beginn der Methode wird die aktuelle Zeit in `now` gespeichert, diese wird benötigt, um die Dauer niedriger Last des Places zu bestimmen. Um aus der JVM den Hardwarestatus des Nodes auszulesen wird die Java-Funktionalität der *OperatingSystemMXBean* genutzt. Dabei wird über eine Sekunde mit einem Zeitabstand von 50 Millisekunden 20 Mal die aktuelle CPU-Last ausgelesen, aus diesen Werten wird ein Mittel gebildet. Das Mittel wird genutzt, um kurze Tiefpunkte oder Spitzen in der Last zu vermeiden. Sobald ein Place eine Last von unter einem Prozent hat wird in `lowLoadSince` die aktuelle Zeit gespeichert. Diese wird in jedem Durchlauf mit der aktuellen Zeit aus `now` verglichen um herauszufinden, wie lange ein Place bereits kaum Last hat. Hat ein Place beim Auswerten 10 Sekunden lang eine Last von unter einem Prozent, wird eine Last von 0 zurückgegeben, da eine Last von unter einem Prozent wahrscheinlich nicht von einer APGAS-Berechnung stammt, sondern vom System des Nodes. Dies wird benötigt für den Fall, dass ein Place lange kaum ausgelastet ist, aber der Schnitt der Last aller Places nicht unter dem Schwellwert für niedrige Last sinkt. In diesem Fall würde kein Place heruntergefahren, obwohl zumindest ein Place einige Zeit kaum ausgelastet ist.

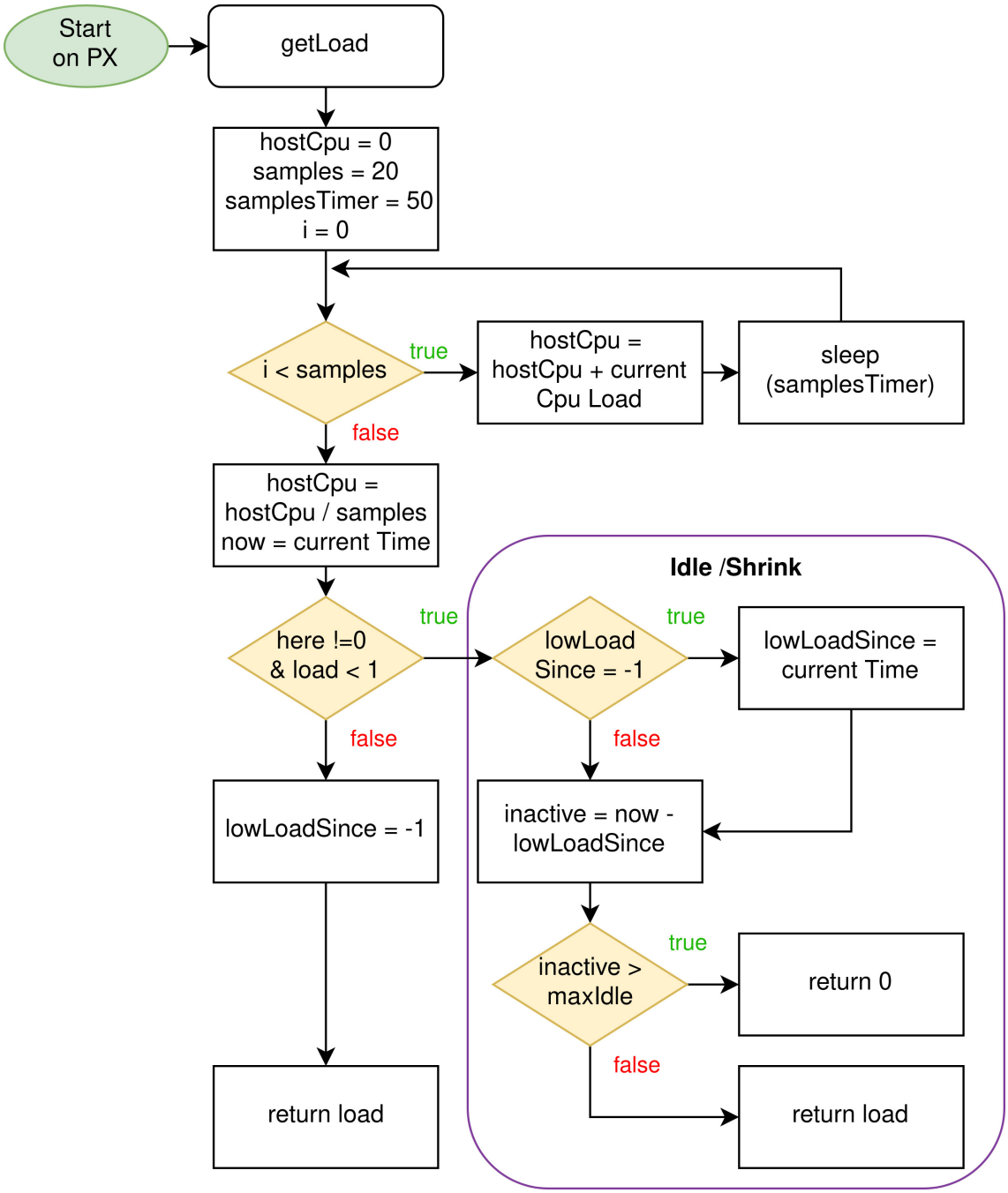

Abbildung 4.2: Flussdiagramm `getLoad`, CPU-basiert

4.1.2 Task-basierte Lastsammlung

Die zweite implementierte Variante der Lastsammlung wertet die Last anhand der Tasks der Places aus. Dafür wurde in Zuge dieser Arbeit die GLB-Klasse `getTaskLoad` implementiert. Der Ablauf der task-basierten `getLoad`-Methode ist als Flussdiagramm dargestellt in Abb. 4.3. Die task-basierte `getLoad`-Methode sammelt auf jedem Place zuerst den aktuellen Stand der Workerbags, die Inhalte der inter queue sowie intra queue sowie die aktuelle Zeit. Dann wird aus den Bags zusammen mit der inter und intra queue die gesamte Task-Anzahl des Place ermittelt. Die gesamte Task-Anzahl wird dann durch die Anzahl der Worker auf einem Place geteilt, um die Last pro Worker festzustellen. Folgend wird die aktuelle Zeit in `now` gespeichert, um sie später mit der Zeit ohne Tasks zu vergleichen.
Daraufhin wird geprüft ob die Task-Last so gering ist, dass ein `shrink` ausgeführt werden soll. Dazu wird geprüft ob der Place aktuell keine Tasks hat - falls dem so ist, wird geprüft ob dieser Zustand gerade erst eingetreten ist. Wenn dieser Zustand gerade erst eingetreten ist, dann wird in `noTasksSince` die aktuelle Zeit gespeichert. Danach wird die Zeit der Inaktivität des Places aus der Differenz von `now` und `noTasksSince` berechnet. Wenn nun die Inaktivität einen gesetzten Schwellwert überschreitet wird 0 als Last des Places zurückgegeben. Wodurch der Place bei der nächsten Auswertung zum Herunterfahren ausgewählt wird. Wenn ein Place eine Last von 0 hat wird unabhängig von der Durchschnittslast ein `shrink` auf diesen aufgerufen; wenn es mehrere Places mit der selben Minimallast gibt, wird aus diesen der Place mit der geringsten ID zum Herunterfahren gewählt. Falls der Place keine Tasks hat, aber der Schwellwert noch nicht überschritten ist, wird als Last 0.01 zurückgegeben, um den Schnitt minimal zu beeinflussen, aber den Place noch nicht herunter zu fahren. Wenn an dieser Stelle kein `return` mit einem Wert von 0.01 zurückgegeben würde, würde die Methode weiter laufen und `noTasksSince` auf -1 zurücksetzen sowie in der folgenden Auswertung 0 zurückgeben, was den Place für eine `shrink`-Operation freigeben würde.
Falls zu einem Auswertungszeitpunkt Tasks auf einem Place vorhanden sind, wird zuerst der Wert in `noTasksSince` auf -1 zurückgesetzt, da dieser nicht inaktiv ist. Daraufhin wird geprüft, ob die Anzahl der Tasks des Place kleiner als die Anzahl der Worker ist, was bedeutet, dass nicht für jeden Worker Arbeit vorhanden ist. In diesem Fall wird der Anteil von Tasks zu Workern zurückgegeben. Zum Beispiel würde bei 2 Tasks und 4 Workern der Wert 50 zurückgegeben werden. Ist die Anzahl der Tasks allerdings größer als die Anzahl

der Worker, ist der Place zu diesem Zeitpunkt voll ausgelastet und es wird der Wert 100 zurückgegeben.

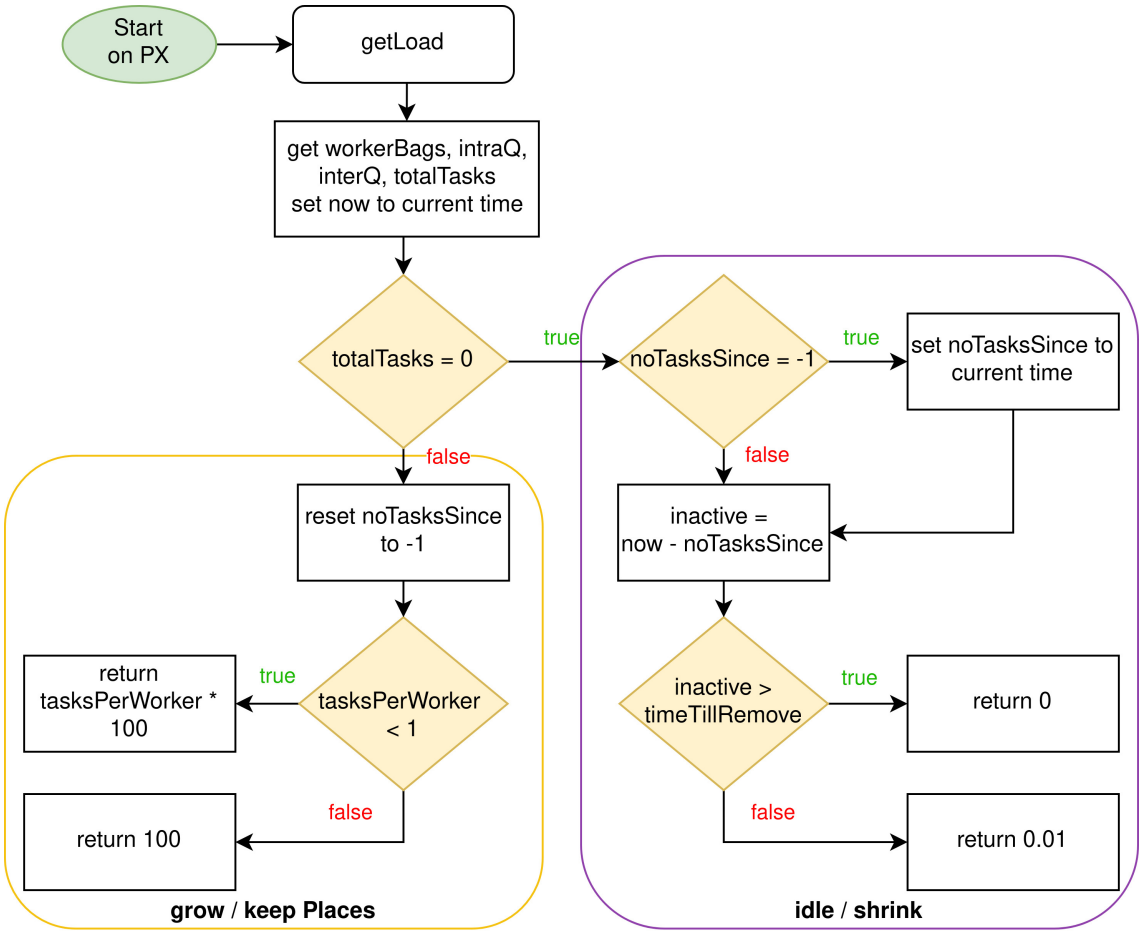

Abbildung 4.3: Flussdiagramm `getLoad`, task-basiert

4.2 Lastauswertung

Abbildung 4.4 zeigt den Ablauf der Methode `loadEvaluation` aus der APGAS-Klasse `Evolving`, diese wird nur auf Place 0 ausgeführt. Diese Methode nutzt die erhaltenen Werte der `obtainPlaceLoad`-Methode um zu beurteilen, ob ein neuer Place gestartet oder ob ein Place heruntergefahren werden soll. Zu Beginn der Methode wird ein neuer asynchroner Thread gestartet, welcher bis zum Programmende läuft, der Thread wartet zunächst für drei Sekunden, damit Daten zum Auswerten vorhanden sind. Solange Place 0 aktiv ist, wird alle zwei Sekunden die Last ausgewertet. In dieser Zeit wird von allen Places durch die `obtainPlaceLoad`-Methode die aktuelle Last des jeweiligen Places in der Hashmap auf Place 0 aktualisiert. Die Auswertung des aktuellen Durchlaufs der

`loadEvaluation` wird nur ausgeführt, wenn nicht bereits auf Änderungen der Place-Anzahl gewartet wird (`waitingForPlaceChanges=false`). Wenn nicht gewartet wird, wird die Methode `calcMinAndAvgLoad` aufgerufen, welche den Place mit der geringsten Last findet und dessen ID speichert sowie die durchschnittliche Last aller Places berechnet und diese zurück gibt.
Nun wird ausgewertet, ob die durchschnittliche Last über dem vom Nutzer gesetzten Wert für hohe Last liegt, wenn dem so ist folgt die `grow`-Prozedur. In der nun folgenden `grow`-Prozedur wird die Methode `getNextHost` genutzt um den Node mit der geringsten Place-Anzahl zu ermitteln, falls die Methode `null` zurück gibt wird der aktuelle Durchlauf übersprungen, da auf keinem Node weitere Places gestartet werden können. Wenn die Methode einen Node zurück gibt, wird dieser in einer Liste gespeichert. Daraufhin wird die aktuelle Zeit in `startedWaiting` gespeichert. `startedWaiting` wird in der Methode `obtainPlaceLoad` genutzt um zu prüfen, ob ein `grow` oder `shrink` ungewöhnlich lange dauert. Dann wird die Liste genutzt, um einen `grow` auf diesem Node auszuführen. Aktuell wird bei einem `grow` immer genau ein neuer Place gestartet, da nach einem `grow` die Gesamt-Last anders ausfällt. In beiden Fällen wird als letzter Schritt der `grow`-Prozedur die Liste der freien Nodes geleert.
Wenn die Bedingungen für einen `grow` nicht erfüllt sind, wird geprüft ob die Bedingungen für einen `shrink` erfüllt sind. Ein `shrink` findet dann statt, wenn die Last auf dem Place mit der geringsten Last genau null ist oder die durchschnittliche Last unter der vom Nutzer gesetzten Grenze liegt. Sobald ein `shrink` eingeleitet wird, wird auf dem zu herunterzufahrenden Place der Thread gestoppt, welcher die Last sammelt. Daraufhin wird die APGAS-Methode `evolvingShrink` mit dem herunterzufahrenden Place aufgerufen. Sobald diese abgeschlossen ist, wird die Liste der zu herunterzufahrenden Places geleert.
Vor jedem `shrink` und `grow` wird die Variable `waitingForPlaceChanges` auf `true` gesetzt, um während der Prozedur keine unnötige Sammlung und Auswertung der Last zu betreiben. Sobald die jeweilige Prozedur abgeschlossen oder das *Timeout* überschritten ist, wird die Variable wieder auf `false` gesetzt. Das Timeout wird in jedem Durchlauf der `obtainPlaceLoad` geprüft, solange `waitingForPlaceChanges true` ist.

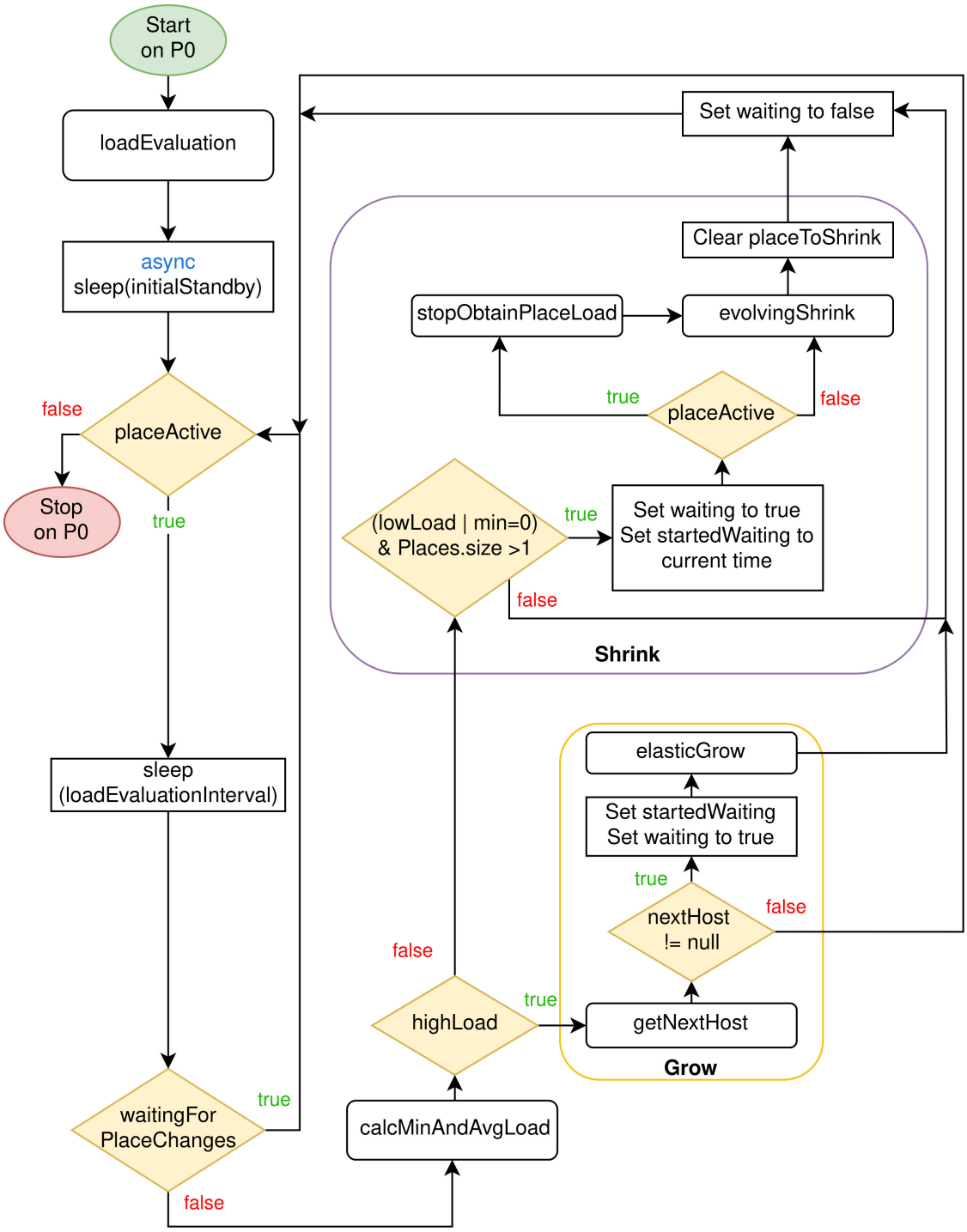

Abbildung 4.4: Flussdiagramm der Methode `loadEvaluation`

5 Performance-Bewertung

Dieses Kapitel behandelt die Performance-Auswirkungen der Evolving-Funktionalität im genutzten Benchmark. Es ist unterteilt in eine kurze Beschreibung der Experimentierumgebung 5.1 und darauffolgend eine Beschreibung des Benchmarks 5.2. Das Kapitel schließt mit den erzeugten Messergebnissen und deren Auswertungen 5.3.

5.1 Experimentierumgebung

Die Änderungen an APGAS bzw. GLB wurden auf dem Cluster des Fachbereichs 16 der Universität Kassel evaluiert. Das Cluster des Fachbereichs 16 besteht aus 12 Nodes. Die Nodes sind ausgestattet mit AMD Opteron 6276 CPUs mit 32 CPU-Kernen bei einer Taktrate von 2.3 GHz und 64 GB RAM.

5.2 Benchmark

Aufbau des Benchmarks:
Zur Auswertung der Evolving-Funktionalität wird ein synthetischer Benchmark genutzt, dieser Benchmark ist in Abb. 5.1 exemplarisch gezeigt. In Abb. 5.1 sind der Übersicht halber im ersten und dritten Teil pro Elternknoten genau zwei Kinder dargestellt und die Tiefe der Teilbäume sowie des Asts ist gering. Dies muss im Benchmark nicht so sein.
Der Basis-Benchmark existierte bereits im genutzten GLB. [4] Der Basis-Benchmark sucht einen geeigneten m-ären Baum der am ehesten zu den gewählten Benchmark-Parametern passt. Daraufhin wird der Baum mit der festgestellten Tiefe und Anzahl an Kindern pro Knoten dynamisch erzeugt. Es wurde die Option eingefügt, nicht nur den gefundenen m-ären Baum zu erzeugen, sondern einen Baum bestehend aus zwei gleichen spezifischen m-ären Teilbäumen, welche durch einen Ast verbunden sind. Dabei wird dynamisch ein irregulärer Baum generiert. Das bedeutet, dass Folgetasks erst nach der Berechnung einer

Task generiert werden. Die Knoten des Baums entsprechen dabei den Tasks. Der komplette Baum kann als ein Baum bestehend aus drei Teilen angesehen werden.
Im ersten Teil wird ein irregulärer m-ärer Teilbaum generiert. Während der Generierung des ersten Teils wird der Knoten ganz rechts auf der letzten Ebene festgestellt. Dieser Knoten wird im zweiten Teil genutzt um an diesem einen Ast zu generieren, welcher nach der Berechnung einer Task nur genau eine neue Task generiert. Somit wird ein nicht parallelisierbarer Engpass erzeugt. Am letzten Knoten des Asts wird erneut der selbe Teilbaum, wie im ersten Teil, generiert.

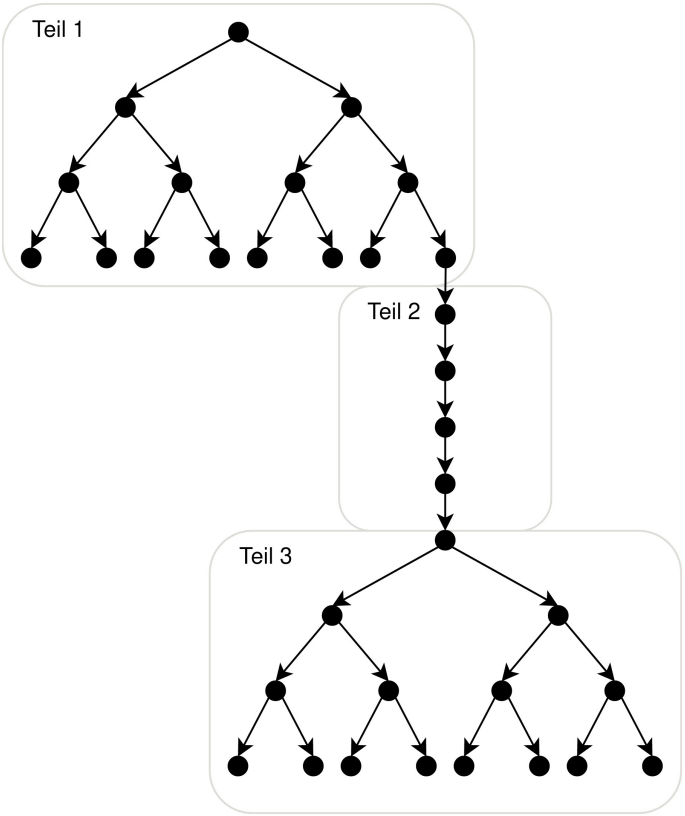

Abbildung 5.1: Synthetischer Baum, unterteilt in drei Teile, exemplarisch dargestellt mit zwei Kindern pro Elternteil innerhalb der Teile 1 und 3 mit einer Teilbaumtiefe von 4 sowie einer Astlänge von 4 im zweiten Teil.

Die Task-Anzahl des ersten sowie dritten Teils ergibt sich durch $Teilbaum = Anzahl\ der\ Worker \cdot Tasks\ pro\ Worker$ und die gesamte Anzahl der Tasks eines Jobs durch $Task{-}Anzahl = Teilbaum \cdot 2 + Astlänge$. Die Laufzeit des Benchmarks beträgt somit $Laufzeit = Dauer\ eines\ Tasks \cdot Varianz \cdot Task{-}Anzahl$. Die Parameter für den Benchmark sind die Folgenden:

- *Synth* gibt die Benchmark-Variante an, diese kann der originale Benchmark sein oder die erweiterte Variante `evotree` - für die Messungen auf `evotree` gesetzt.

- *branch* gibt die Länge des Asts an - für die Messungen auf 5000 gesetzt.

- *dynamic* true setzt die Erzeugung der Tasks des Baums auf dynamisch d. h. eine neue Task wird dann generiert, sobald eine vorhergehende Task berechnet wurde - für die Messungen auf `true` gesetzt.

- g ist die Dauer einer Task in Millisekunden - für die Messungen auf 100000 gesetzt.

- u gibt die maximale Varianz der Task-Dauer in Prozent an - für die Messungen auf 1000000 gesetzt.

- t sind die pro Worker erzeugten Tasks für die Generierung des m-ären Teilbaums - für die Messungen auf 20 gesetzt.

Die Laufzeit des Asts ist so konfiguriert, dass diese doppelt so lang ist, wie die Laufzeit des Teilbaums aus Teil 1, diese ist auf die Tasks des Asts aufgeteilt. Mit den genutzten Parametern ergibt sich eine Laufzeit von 100 Sekunden für den Teilbaum in Teil 1 und 3 sowie 200 Sekunden für den Ast, daraus ergibt sich eine Gesamtlaufzeit eines Programmlaufs von 400 Sekunden, bei Berechnung mit einem Node (sequenziell). Somit ist nur die Hälfte eines Laufs parallelisierbar. Bei den Messungen wird *weak scaling* [19] genutzt, das bedeutet, dass mit mehr genutzten Ressourcen auch der Workload (Taskanzahl) ansteigt.

Zielsetzung des Benchmarks: Um die Evolving-Funktionalität zu evaluieren müssen `grow-` und `shrink`-Operationen ausgelöst werden. Im Zeitfenster der Ast-Generierung ist nur noch der Place, welcher den Ast erzeugt ausgelastet. Alle anderen Places sind während dieser Zeit inaktiv. Die Lastverteilung durch GLB hat im nicht parallelisierbaren Teil des Baums keinen Effekt. Sobald der Ast erreicht ist, wäre es sinnvoll alle Places, außer dem den Ast generierenden Place herunter zu fahren. Wenn der Job rigid gestartet worden wäre, würden an dieser Stelle einige Places und somit auch Nodes inaktiv laufen, bis der Teilbaum von Teil 3 erreicht wird. Im Evolving-Modus werden an dieser Stelle durch die Evolving-Funktionalität, ohne manuellen Eingriff, alle inaktiven Places heruntergefahren. Sobald das Programm im dritten Teil angekommen ist, kann wieder Arbeit an andere Places verteilt werden; dann werden wieder neue Places gestartet.

5.3 Messergebnisse und Auswertung

Für alle Messungen wurde die `evotree`-Variante des synthetischen Benchmarks genutzt. Dadurch, dass nur die Elastizität der Jobs und nicht die Interaktion mit dem Scheduler implementiert wurde, werden zum Auswerten im Evolving-Modus die doppelte Anzahl an Nodes reserviert, als Places zum Programmbeginn gestartet werden. Dies soll simulieren, wie sich ein elastischer Job in Kombination mit einem elastischen Scheduler - unter der Annahme freier Nodes im Cluster - verhalten würde. Bei den Versuchen wurde pro Node maximal ein Place gestartet.
APGAS hat einige konfigurierbare Parameter: `Elastic` gibt an ob APGAS rigid, malleable oder evolving gestartet werden soll. Wenn `elastic` auf evolving gestellt wurde, kann per `mode` die Lastsammlung auf CPU- oder task-basiert gestellt werden. `Lowload` und `Highload` geben die Grenzen an, bei welcher Last ein `grow` oder `shrink` ausgeführt werden soll. Wenn die Nodes *Hyperthreading* unterstützen, muss `hyperthreading` auf `true` gesetzt werden, was die eingegebenen Grenzen halbiert. Hyperthreading wird nur bei `mode cpu` und hyperthreadfähigen Nodes genutzt, daher ist es bei diesen Messungen auf `false` gesetzt. Für die Messungen wurden die in Tabelle 5.1 gelisteten APGAS-Parameter genutzt.

Tabelle 5.1: APGAS-Parameter

Parameter	Rigid	Evolving CPU	Evolving Task
elastic	fixed	evolving	evolving
mode	none	cpu	task
lowload	0	10	10
highload	0	90	90
hyperthreading	false	false	false

Im Folgenden werden reale, exemplarische Läufe des Benchmarks vorgestellt. Der erste Programmlauf, zu sehen in Abb. 5.2, zeigt den Verlauf mit einem Place zu Programmbeginn und zwei reservierten Nodes - auf diesen wird im Detail eingegangen.

Der Programmlauf startet mit einem Place, nach 4 Sekunden startet ein `grow`, der nach 6 Sekunden abgeschlossen ist. Die Berechnung des ersten Teilbaums ist nach 54 Sekunden abgeschlossen, dies ergibt sich durch die Einstellung von 100 Sekunden für den ersten Teilbaum bei einem Place. Durch das Hinzufügen eines zweiten Place wird die Laufzeit halbiert plus den `grow` *Overhead*.
In Sekunde 55 nach Programmstart beginnt die Generierung des Asts, drei Sekunden

danach wird der nun inaktive Place durch einen `shrink` heruntergefahren. Die Laufzeit des Asts beträgt 200 Sekunden und ist somit bei Sekunde 255 abgeschlossen.
Daraufhin werden wieder mehr Tasks generiert und durch die hohe Last wird in Sekunde 256 ein `grow` ausgeführt. Die Berechnung des dritten Teilbaums dauert 54 Sekunden, wodurch sich eine Gesamtlaufzeit von 308 Sekunden ergibt.

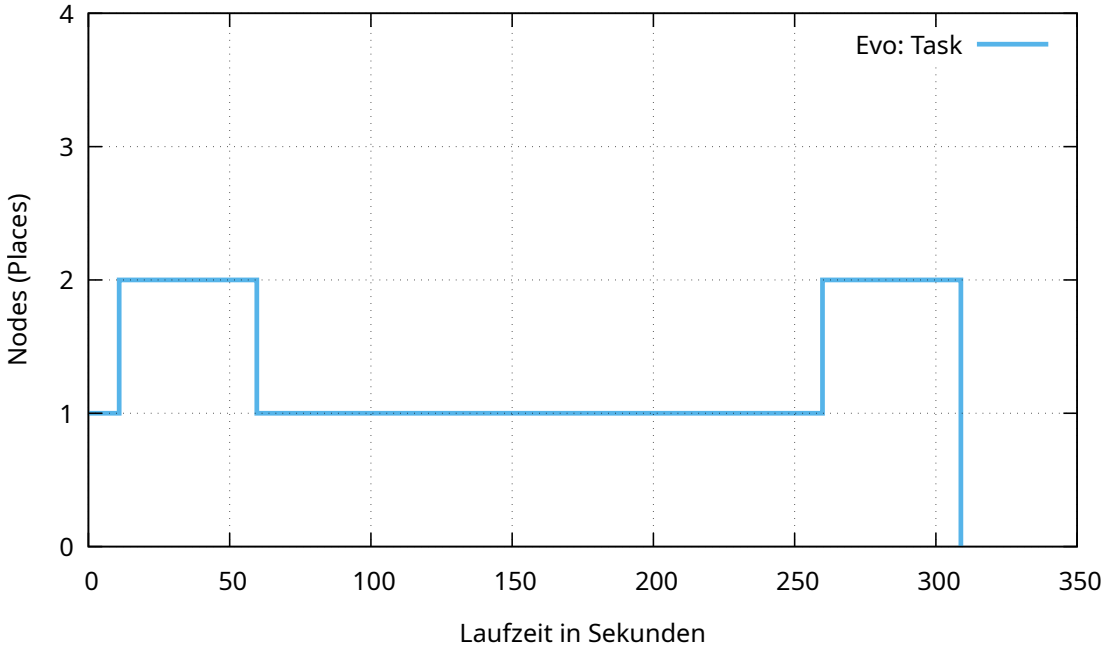

Abbildung 5.2: Nodes (Places) über die Laufzeit hinweg, bei einem Place zu Programmbeginn und zwei reservierten Nodes.

Abb 5.3 zeigt den Verlauf von drei Läufen, einem rigid-Lauf mit 6 Places zu Programmbeginn zu 6 reservierten Nodes sowie zwei evolving-Läufen. Einen mit CPU-basierter und einen mit task-basierter Lastsammlung bei jeweils 6 Places zu Programmbeginn zu 12 reservierten Nodes.

Abbildung 5.3 zeigt den in Abb. 5.2 beschriebenen Verlauf anhand von 6 Places zu Programmbeginn und 12 reservierten Nodes. Durch die Last zu Beginn des Programms bis zum Ast werden per `grow` auf allen freien Nodes Places gestartet. Sobald die Generierung des Astes beginnt, werden alle inaktiven Places via `shrink` heruntergefahren. Sobald der dritte Teilbaum erreicht wird, wird die Anzahl der Places erneut per `grow` erhöht bis die maximale Anzahl an Places erreicht ist.

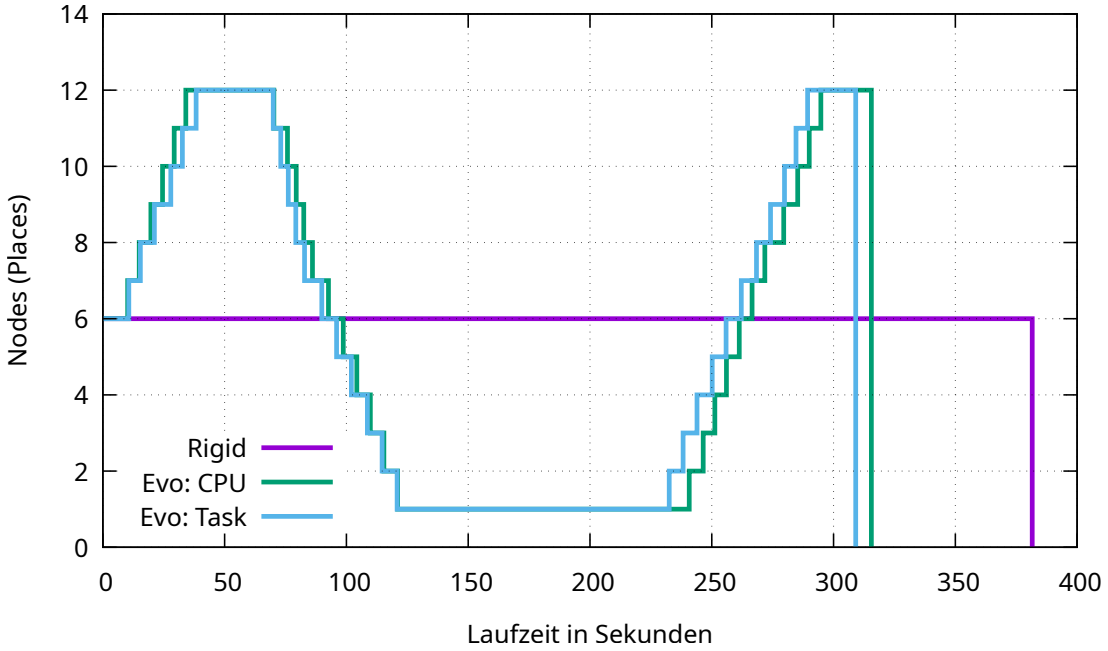

Abbildung 5.3: Nodes (Places) über die Laufzeit bei 6 Places zu Programmbeginn und 12 reservierten Nodes.

Die Abbildung 5.4 zeigt einen Vergleich der Laufzeiten von Jobs, welche rigid respektive evolving gestartet wurden. Dabei wurden die Kurven aus einem Mittel von 20 Läufen gebildet, die Dauer von `grow`- und `shrink`-Operationen liegen bei diesen Messungen im Durchschnitt bei circa 3,96 Sekunden, wobei `grow`-Operationen circa 4,09 und `shrink`-Operationen circa 3,71 Sekunden dauerten.

Das Absinken der durchschnittlichen Laufzeit der rigid Jobs ist bedingt durch die Art der Generierung des Teilbaums in Teil 1. Bei den genutzten Parametern der Messungen variiert es stark, wann die Ast-Generierung beginnt, das liegt einerseits an der Varianz der Tasklänge und andererseits an der Wahl den Ast immer am Blatt ganz rechts auf der letzten Ebene zu erzeugen. Durch die Wahl den Ast am Blatt ganz rechts auf der letzten Ebene des Teilbaums zu generieren, ist die Art der Generierung des kompletten Baums im Benchmark immer gleich. Allerdings kann die Struktur des Teilbaums dazu führen, dass die rein sequenzielle Dauer durch eine variierende Startzeit des Asts gleichsam variiert, da die Berechnung aller Tasks des Teilbaums aus Teil 1 noch nicht auf allen Places abgeschlossen sein muss, wenn ein Place anfängt den Ast zu generieren. Bei einem spezifischen Lauf mit kurzer Laufzeit von 311 Sekunden wurde der Ast bereits 15 Sekunden nach dem Start generiert, wodurch der sequenzielle Anteil der Laufzeit geringer ist, als bei einem Lauf mit

einer Laufzeit von 388 Sekunden, bei welchem die Ast-Generierung erst nach 65 Sekunden beginnt.

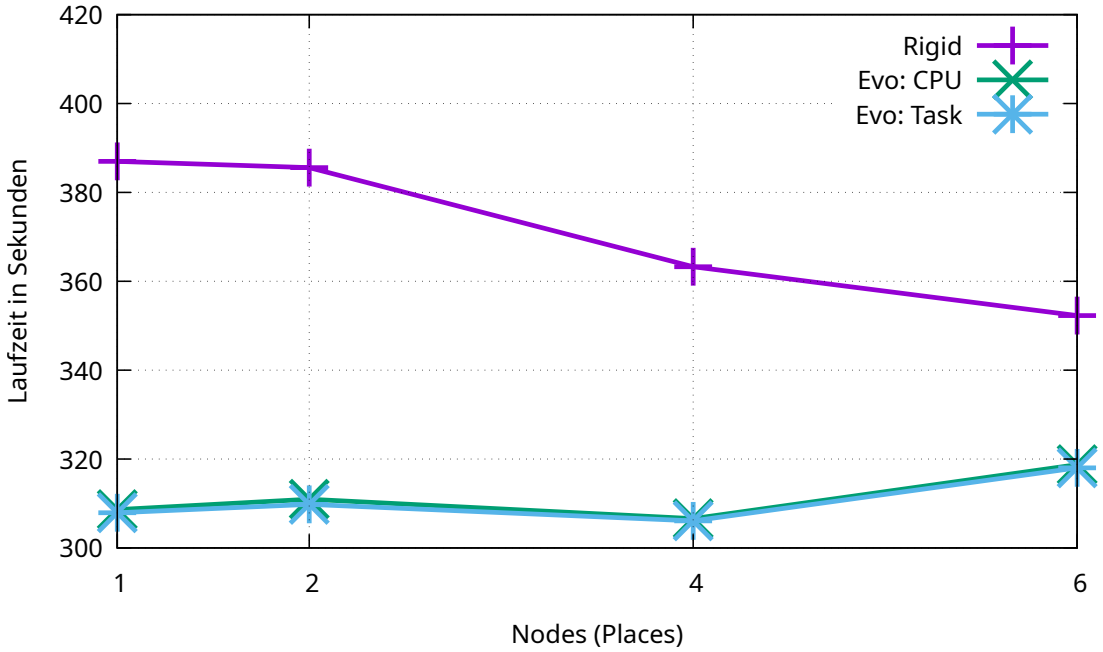

Abbildung 5.4: Mittel von 20 Läufen gestartet rigid, evolving CPU und evolving Task.

Wie in der Abbildung 5.4 gut ersichtlich wird, ist der Unterschied der Laufzeit von CPU- und task-basierter Evolving-Funktionalität sehr gering. Die Vergleiche der durchschnittlichen Laufzeit von Jobs mit halb so vielen Places zu Programmbeginn, wie Nodes zeigen, dass durch die Evolving-Funktionalität ein positiver Effekt auf die Laufzeit erzielt werden kann. Es wurde eine Verkürzung der Laufzeit von 10,5 - 21% auf die Programmläufe und von 21 - 42% im parallelisierbaren Anteil des Benchmarks unter Annahme von doppelt so vielen freien Nodes wie gestarteten Places gemessen. Der Speedup eines Programms durch Parallelität hängt grundsätzlich vom parallelisierbaren Anteil ab. Er fällt unter Annahme eines größeren, parallelisierbaren Anteils höher aus, da der genutzte Benchmark ausschließlich zur Vorstellung und Auswertung der Funktionalität dient und bis zu einem Anteil von 50% sequenziell ausgeführt wird. Bei einem größeren sequenziellen Anteil ist der Speedup des Jobs folglich geringer.

Wenn ein evolving Job im Fall eines Lasttals Nodes freigibt, welche er durch den darauf folgenden Start eines anderen Jobs nicht zeitnah wieder erhalten kann, kann dies die Laufzeit dieses spezifischen Jobs negativ beeinflussen. Dies wäre besonders kritisch unter der Annahme eines Schedulers, welcher rigid sowie evolving Jobs erlaubt, da diese Ressourcen bis zum Ende der Laufzeit eines rigid Jobs sicher nicht wieder verfügbar werden.

Durch die kurze Laufzeit der Messungen haben die `grow` und `shrink` Overheads in den Läufen einen nicht zu vernachlässigenden Anteil im Verhältnis zur gesamten Laufzeit.

Es wurde zusätzlich gemessen, wie gut ein evolving Job im Verhältnis zu einem rigid Job abschneidet, wobei beide Jobs bereits mit gleich vielen Places zu Programmbeginn, wie Nodes gestartet wurden. Dies soll simulieren, dass auch ohne das Freisein von weiteren Nodes im Cluster ein Gewinn durch Elastizität erzielt werden kann, da in diesem Fall Nodes freigegeben würden, welche sonst reserviert, aber inaktiv wären. Die Kosten für die Nutzung der inaktiven Nodes durch andere Jobs ist hierbei der durch die `grow`- bzw. `shrink`-Operationen generierte Overhead. Bei diesen Messungen wurde der Job evolving sowie rigid mit 12 Places zu Programmbeginn auf 12 Nodes gestartet. Bei 20 Läufen lag die durchschnittliche Laufzeit von rigid Jobs bei 395 Sekunden und die durchschnittliche Laufzeit von evolving Jobs bei 428 Sekunden: Das entspricht einer um circa 7,7% längeren Laufzeit von evolving Jobs. Diese Variabilität der Laufzeit entsteht durch den von `grow`- bzw. `shrink`-Operationen generierten Overhead. Durch das Herunterfahren von inaktiven Places sind somit bei diesen Messungen im Schnitt circa 144,5 Sekunden lang 11 Nodes inaktiv. Die 144,5 Sekunden ergeben sich aus dem 45 Sekunden Zeitfenster von Sekunde 75 bis Sekunde 120 nach Programmstart, bis alle Places bis auf einen heruntergefahren sind, was 22.5 Sekunden mit 11 heruntergefahrenen Places entspricht, plus die durchschnittlich 120 Sekunden mit einem Place (siehe Abb. 5.3).
Die Vergleiche der durchschnittlichen Laufzeit von Jobs mit genauso vielen Places zu Programmbeginn, wie Nodes zeigen, dass die Evolving-Funktionalität auch im Fall von keinen weiteren freien Nodes im Cluster einen positiven Effekt haben kann, da der evolving Job bei 12 Places auf 12 Nodes durchschnittlich nur 7,7% langsamer war, verglichen mit einem rigid Job mit den selben Parametern. Durch den evolving Job wären aber während der Laufzeit mehr Nodes im Cluster frei verfügbar, als es beim rigid Job der Fall wäre.

6 Verwandte Arbeiten

In diesem Kapitel wird auf artverwandte Arbeiten eingegangen, also auf Arbeiten, welche ebenfalls Ressourcenelastizität in parallelen Programmiersystemen ermöglichen oder mit Elastizität von Jobs bzw. Schedulern in Zusammenhang stehen. Dabei ist zu erwähnen, dass die Jobs und Ressourcen von einem Batch-System, auch Ressource Management System (RMS) genannt, verwaltet werden. Dieses setzt sich aus dem Scheduler und einem Ressource-Manager zusammen. Zuerst werden Arbeiten im Bezug auf dynamische, Malleability bezogene Batch-Systeme vorgestellt. Dies ist relevant, da Malleability die Basis für die Evolving-Funktionalität darstellt. Für evolving Jobs werden erweiterte Batch-Systeme benötigt, welche auch von malleable Jobs benötigt werden. Zuletzt wird auf evolving bezogene Arbeiten eingegangen.

- In der Dissertation *High-throughput Computation through Efficient Resource Management* von Iserte (2018) [9] wird die *Dynamic Management of Resources Library* (DMR) proponiert, welche SLURM als RMS nutzt. SLURM wurde dabei derart erweitert, dass es dynamisch Ressourcen verwalten kann und malleable Jobs annimmt. Zur Laufzeit meldet sich das Programm bei dem RMS zu festgelegten Stellen im Programm mit einem Vorschlag zur Rekonfiguration, daraufhin evaluiert das RMS seinen Systemstatus und gibt eine möglicherweise angepasste Konfiguration an das Programm zurück. Dabei kann es zu `shrink`- oder `grow`-Operationen kommen. Hier existiert bereits die Verbindung zur Clusterverwaltung. Allerdings wird die Anpassung nicht dynamisch ohne Eingriff des Nutzers vorgenommen, sie ist also malleable.
- In der Dissertation *Optimization techniques for adaptability in MPI applications* von Cruz (2015) [12] wird eine Variante von MPI [8] basierend auf MPICH [17] proponiert, welche MPI um eine performance-bewusste, dynamische Rekonfiguration erweitert. Diese Variante wird als *FLEX-MPI* bezeichnet. Diese Variante von MPI löst eine Rekonfiguration anhand der aktuellen Programm-Performance und der Verfügbarkeit von Ressourcen aus. FLEX-MPI nutzt dazu eine Monitoring-Komponente welche sich, wie auch die in dieser Arbeit genutzte Auswertungsschemata, an der Hardware und dem Parallelisierungssystem

(bei Cruz MPI, in der vorliegenden Arbeit APGAS) orientiert. FLEX-MPI bietet eine Process-Scheduler-Komponente, welche für das Erzeugen und Schließen von Prozessen zuständig ist. Für FLEX-MPI existiert bereits eine erweiterte Variante [33], welche eine Optimierung des Energieverbrauchs zum Ziel hat. Jene Erweiterung kann auf Minimierung der Energienutzung oder Maximierung der Leistung pro Watt konfiguriert werden.
Bei FLEX-MPI und DMR wird aufgrund der Malleability mit festgelegten Punkten zur Rekonfiguration gearbeitet. In der vorliegenden Arbeit wird evaluiert, wann Places gestartet oder heruntergefahren werden sollen, dies geschieht ohne Rekonfigurationspunkte. Allerdings werden noch keine Nodes freigegeben oder zugebucht, da die Verbindung zum Scheduler/RMS noch nicht implementiert ist. Bei FLEX-MPI und DMR ist die Verbindung bereits gegeben, allerdings ohne Evolving-Funktionalität. In meiner Arbeit ist nur die Basis für die job-seitige evolving-Funktionalität gegeben. Sobald die Verbindung zum Scheduler/RMS implementiert ist, wird die Entscheidung, ob eine Anpassung der Node-Anzahl beim Scheduler angefragt wird, vom evolving-Programm entschieden. Die finale Entscheidung ob eine Anpassung der Nodes stattfindet, obliegt in allen Varianten dem RMS. Die Zielsetzung der Ressourcenanpassungen des erweiterten FLEX-MPI unterscheidet sich grundsätzlich von der Zielsetzung in der vorliegenden Arbeit: Bei Cruz steht der Energieverbrauch im Fokus, in der von mir durchgeführten Arbeit die Ressourcenauslastung.

- Die Veröffentlichung *A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications* von Prabhakaran et al. (2015) [32] befasst sich mit dynamischem Scheduling für malleable- sowie evolving-Anwendungen, dabei wurde der *Torque* [36] Ressource-Manager angepasst. Die Arbeit *A Batch System with Fair Scheduling for Evolving Applications* [31] veröffentlicht 2014 von Prabhakaran et al. steht in direktem Zusammenhang zur zuvor genannten Veröffentlichung. Beide Arbeiten befassen sich mit den Verfahren eines Scheduler zur Verteilung von Ressourcen an Jobs. Dabei geht es um die Frage der Fairness eines Schedulers, welcher rigid sowie malleable und evolving Jobs bzw. Ressourcenanfragen akzeptiert. In der Arbeit wurde das Torque/Maui RMS angepasst. Die Auswertung anhand des ESP Benchmark [38] führte zu einer Reduzierung der Wartezeit, einer erhöhten Auslastung sowie einem erhöhten Durchsatz des Clusters.
In beiden Veröffentlichungen wird das Torque/Maui RMS genutzt, im Gegensatz zu dem für die vorliegende Arbeit genutzten SLURM [22]. In den Veröffentlichungen wurde Torque/Maui erweitert, um malleable und evolving Jobs zu unterstützen und im Besonderen auch eine Mischung von verschiedenen Job-Typen. Das gewählte Programmiersystem zur Parallelisierung ist in beiden Veröffentlichungen Charm++ [23], im Gegensatz zu APGAS

in der vorliegenden Arbeit. Beide Veröffentlichungen - besonders aber die Letztere - ziehen die Fairness eines RMS in Betracht. Fairness beim Scheduling kann den nötigen Anreiz geben, Programmiersysteme um Malleable- oder Evolving-Funktionalität zu erweitern. In diesen Arbeiten wird *Equipartitioning* (Gleichverteilung auf Jobs) [27, 35] als Basis für die Fairness gewählt. Ein `grow` wird als sofort ausführbare Operation angesehen. Wohingegen `shrink` den Checkpoint-Mechanismus von Charm++ zur sofortigen Ausführung nutzt. Bei der vorliegenden Arbeit ist die `grow`-Operation im Programmiersystem auch als sofortig ausführbare Operation anzusehen. APGAS bzw. GLB nutzt kein Checkpoint-System. Vor dem Herunterfahren eines Places werden Tasks vom herunterzufahrenden Place an verbleibende Places verteilt.

- Die Bachelorarbeit *Prototypische Entwicklung eines Schedulers für Malleable-Jobs* von Bürger (2023) [10] stellt einen angepassten Scheduler vor. Dabei wird ein Teil eines Clusters reserviert und dort ein Node als Management-Node dediziert, dieser ist in der Lage auf Anfragen von elastischen Jobs einzugehen. Das ist, von der Fragmentierung des Clusters abgesehen, der Teil, welcher zu vollständiger Elastizität dieser Ausarbeitung aktuell fehlt. Die Arbeit Bürgers (2023) und die vorliegende Arbeit bedürfen Anpassungen um zusammen eine elastische Umgebung bereitstellen zu können.
- Die Bachelorarbeit *Weiterentwicklung und Evaluation von Scheduling-Algorithmen für elastische Jobs im High-Performance-Computing* von Hupfeld (2023) [20] befasst sich mit verschiedenen Algorithmen für das Scheduling von malleable Jobs, dabei wurde ein Easy-Backfilling-Algorithmus um malleable-Unterstützung angepasst. Weitere Arbeiten die sich mit der Malleability von RMS befassen, auf welche aus Platzgründen nicht im Detail eingegangen wird, sind die Folgenden: *Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling* von Chadha et al. (2020) [11], *Towards realizing the potential of malleable jobs* von Gupta et al. (2014) [18], *Efficient Scalable Computing through Flexible Applications and Adaptive Workloads* (MPI) von Iserte et al. (2017) [21] und schließlich noch die Veröffentlichung *A Malleable-Job System for Timeshared Parallel Machines* (Charm++) von Kale et al. (2002) [24].

7 Fazit

Diese Arbeit hat das parallele Programmiersystem APGAS um evolving-Funktionalität erweitert. Im Verlauf der Arbeit wurden APGAS und GLB angepasst, um evolving Jobs unterstützen. Es wurde die Basis zur Auswertung der Last und Umsetzung der Ressourcenelastizität sowie zweier Faktoren zur Sammlung der Last der Places implementiert. Als Faktoren, nach welchen über die Place-Freigabe bzw. -Hinzubuchung entschieden wird, wurde von mir die CPU-Last der Nodes respektive die Last der Places, ermittelt anhand der Tasks, gewählt.

Im von mir für das evolving-fähige APGAS angepassten `evotree`-Benchmark, erwirkt die evolving-Funktionalität eine Verkürzung der Laufzeit von 10,5 - 21% bei einem Programmlauf und eine Verkürzung von 21 - 42% im parallelisierbaren Anteil des Benchmarks, unter Annahme von doppelt so vielen freien Nodes, wie gestarteten Places. Durch die Anpassungsfähigkeit steigt die potenzielle Auslastung eines Clusters, da wartende Jobs durch die Freigabe nicht genutzter Ressourcen bereits früher gestartet werden können oder elastische Jobs mit hoher Auslastung die freien Ressourcen nutzen können, wodurch sich ihre Laufzeit reduziert. Diese Effekte wären bei mehr elastischen Jobs auf dem selben Cluster stärker.

Die Freigabe von Nodes muss bei isolierter Betrachtung eines einzelnen Jobs nicht zwangsweise von Vorteil für den betrachteten Job sein. Im Gegenteil kann die Freigabe sogar negative Effekte auf die Laufzeit des Jobs haben, wenn nach der Freigabe andere Jobs im Cluster gestartet werden bzw. andere elastische Jobs die freigegebenen Nodes allokieren. Dies könnte besonders dann zu einem Problem werden, wenn der Scheduler sowohl rigid, als auch evolving Jobs akzeptiert.

Die Unterschiede hinsichtlich der Evaluation, wann ein `shrink` oder `grow` ausgeführt wird, fallen bei der Auswertung der Last durch die Hardware (CPU) oder anhand der softwareinternen (Task) Parameter deutlich geringer aus als erwartet. Die Annahme war, dass die Evaluation anhand der Softwareparameter genauer sein würde, verglichen mit jener anhand der Hardwareparameter, da die Evaluation anhand der Softwareparameter auf das Programmiersystem zugeschnitten und die Evaluation anhand der Hardware deutlich

generischer ist.
Die in den Messungen relativ hohe Einwirkung auf die Laufzeit durch den Overhead der Änderungen der Place-Anzahl muss in einem realistischen Szenario nicht unbedingt einen vergleichbar starken Einfluss auf die Laufzeit haben. In einem realistischen Szenario haben Jobs in aller Regel eine weitaus längere Laufzeit und es werden nicht absichtlich viele Änderungen der Place-Anzahl vorgenommen, wie es im Benchmark der Fall ist. Im Allgemeinen wird ein Job bei dem die Last der Nodes bzw. Places stetig stark variiert, mehr Overhead haben, verglichen mit einem Job der seltener größere Änderungen der Last erfährt. Generell, lässt sich zusammenfassend sagen: Je länger die Laufzeit und je geringer die Häufigkeit von stark variierender Last, desto geringer ist der Overhead durch die Evolving-Funktionalität.

Wie an den diversen Prototypen von Ressource Management Systemen des letzten Jahrzehnts gut zu erkennen ist, ist der Wandel zu Elastizität im HPC gerade erst im Anfangsstadium. Solange rigid Jobs noch der Standard sind bzw. die HPC-Landschaft sich im Übergang hin zu einer elastischen Umgebung befindet, sollte ein Anreiz geschaffen werden, die zusätzliche Arbeit der Anpassung an eine elastische Umgebung zu investieren. Diese konzeptionelle Idee kann gut durch eine Anpassung der Fairness der genutzten Ressource Management Systeme unterstützt werden.

7.1 Ausblick

Verbindung zum Scheduler: In dieser Arbeit wurde nur APGAS und GLB für evolving Jobs vorbereitet. Für die vollständige Elastizität auf einem Cluster muss die Kommunikation mit dem Scheduler noch programmiert werden. Sobald die Verbindung der Evolving-Funktionalität zum Scheduler gegeben ist, werden voraussichtlich die simulierten positiven Effekte für das Cluster annähernd real erzielt, da dann nicht nur die Place-Anzahl, sondern auch die dem Job zugewiesene Node-Anzahl dynamisch angepasst wird. Dann nimmt allerdings unweigerlich auch der Overhead zu, da zum Programmiersystem bedingten Overhead die Netzwerkkommunikation zwischen dem Programmiersystem und dem Scheduler hinzukommt. Für die Verbindung zum Scheduler muss die APGAS-Klasse `ElasticCommunicator` angepasst werden, um die APGAS-seitige Kommunikation bereitstellen zu können. An dieser Stelle möchte ich nochmals auf die Arbeit *Prototypische Entwicklung eines Schedulers für Malleable-Jobs* von Bürger (2023) [10] verweisen, die auch unter Kapitel 6 Verwandte Arbeiten aufgeführt und näher beschrieben wird, da diese Arbeit sich explizit mit der Elastizität für Scheduler befasst. Der Scheduler müsste bei der Vergabe von Nodes den Faktor Elastizität und die aktive Freigabe von Nodes eines Jobs mit in ein Prioritätensystem einbeziehen, um zu beurteilen welcher Job als nächstes freie Nodes erhält.
Last Evaluierungsverfahren: Die Evolving-Funktionalität kann um weitere Lastsammlungsverfahren, welche sich (zusätzlich) an anderen Parametern orientieren, erweitert werden. Durch die generische Implementierung zum dynamischen Wachsen und Schrumpfen, kann dies mit relativ geringen Aufwand umgesetzt werden. Dazu müsste eine neue Klasse angelegt werden, welche die Logik zum Auswerten der Last enthält, die neue Klasse muss vom Interface `getLoad` erben. Auch kann das bereits implementierte Verfahren der task-basierten Auswertung durch eine verbesserte Heuristik möglicherweise zu sophistizierteren Ergebnissen führen, verglichen mit der verhältnismäßig einfachen Auswertung der CPU-Last. Wodurch die starke Ähnlichkeit der Verhalten der implementierten Lastsammlungsverfahren möglicherweise reduziert werden würde.

Literaturverzeichnis

[1] APGAS Programming in X10 - x10 und APGAS vorstellung von olivier tardieu (IBM) am Hatree Centre. https://x10.sourceforge.net/tutorials/x10-2.5/APGASProgrammingInX10/APGASprogrammingInX10-slides-V8.pdf. Letzter Zugriff: 20.01.2024.

[2] Batch-scheduler HPC Wiki. https://hpc-wiki.info/hpc/Batch-Scheduler. Letzter Zugriff: 20.01.2024.

[3] Git projekt der APGAS for Java bibliothek. https://github.com/projectwagomu/apgas. Commit hash: 01b98e32 Letzter Zugriff: 20.01.2024.

[4] Git projekt der Lifeline-Based Global Load Balancer bibliothek. https://github.com/projectwagomu/lifelineglb. Commit hash: 8c2fb35 Letzter Zugriff: 20.01.2024.

[5] Scheduling basics HPC Wiki. https://hpc-wiki.info/hpc/Scheduling_Basics. Letzter Zugriff: 20.01.2024.

[6] SLURM HPC Wiki. https://hpc-wiki.info/hpc/SLURM. Letzter Zugriff: 20.01.2024.

[7] X10 Language Specification version 2.3. https://x10.sourceforge.net/documentation/languagespec/x10-230.pdf. Letzter Zugriff: 20.01.2024.

[8] Mpi: A message passing interface. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 878–883, 1993.

[9] Sergio Iserte Agut and Antonio José Peña Monferrer. *High-throughput Computation through Efficient Resource Management*. PhD thesis, Universitat Jaume I, 2018.

[10] Janek Bürger. Prototypische entwicklung eines schedulers für malleable-jobs. https://www.uni-kassel.de/eecs/index.php?eID=dumpFile&t=f&f=39497&token=a64d87b7c61780498d79ac0da88aa877e3cbfb99, 2023. Letzter Zugriff: 20.01.2024.

[11] Mohak Chadha, Jophin John, and Michael Gerndt. Extending slurm for dynamic resource-aware adaptive batch scheduling. In *2020 IEEE 27th International Conference*

on High Performance Computing, Data, and Analytics (HiPC), pages 223–232, 2020.

[12] Gonzalo Martin Cruz. *Optimization techniques for adaptability in mpi applications*. PhD thesis, Computer Sicence and Engineering Department-Universidad Carlos, 2015.

[13] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-performance computing: clusters, constellations, mpps, and future directions. *Computing in Science Engineering*, 7(2):51–59, 2005.

[14] Yuping Fan. Job scheduling in high performance computing. *CoRR*, abs/2109.09269, 2021.

[15] Dror G Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Workshop on job scheduling strategies for parallel processing*, pages 1–26. Springer, 1996.

[16] Patrick Finnerty, Jonas Posner, Leo Takaoka, and Takuma Kanzaki. Malleable apgas programs and their support in batch job schedulers.

[17] William Gropp. Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 7, Berlin, Heidelberg, 2002. Springer-Verlag.

[18] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kalé. Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

[19] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, may 1988.

[20] Fabian Hupfeld. Weiterentwicklung und evaluation von scheduling-algorithmen für elastische jobs im high-performance-computing. `https://www.uni-kassel.de/eecs/index.php?eID=dumpFile&t=f&f=39110&token=aadbd300129c78cb1fc32f3b00c7f880fc2915db`, 2023. Letzter Zugriff: 20.01.2024.

[21] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran, and Antonio J. Peña. Efficient scalable computing through flexible applications and adaptive workloads. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 180–189, 2017.

[22] Morris A. Jette and Tim Wickberg. Architecture of the slurm workload manager. In Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 3–23, Cham, 2023. Springer Nature Switzerland.

[23] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, oct 1993.

[24] L.V. Kale, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 230–230, 2002.

[25] Wooyoung Kim and Michael Voss. Multicore desktop programming with intel threading building blocks. *IEEE Software*, 28(1):23–31, 2011.

[26] Lett Yi Kyaw and Sabai Phyu. Scheduling methods in hpc system: Review. In *2020 IEEE Conference on Computer Applications(ICCA)*, pages 1–6, 2020.

[27] B.G. Patrick and M. Jack. Equipartitioning versus marginal analysis for parallel job scheduling. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 765–768, 2003.

[28] Jonas Posner and Claudia Fohry. Cooperation vs. coordination for lifeline-based global load balancing in apgas. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, page 13–17, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Jonas Posner and Claudia Fohry. A combination of intra- and inter-place work stealing for the apgas library. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 234–243, Cham, 2018. Springer International Publishing.

[30] Jonas Posner and Claudia Fohry. Transparent resource elasticity for task-based cluster environments with work stealing. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, Christian Windisch, and Felix Wolf. A batch system with fair scheduling for evolving applications. In *2014 43rd International Conference on Parallel Processing*, pages 351–360, 2014.

[32] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale. A batch system with efficient adaptive scheduling for malleable and evolving applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 429–438, 2015.

[33] Manuel Rodríguez-Gonzalo, David E. Singh, Javier García Blas, and Jesús Carretero. Improving the energy efficiency of mpi applications by means of malleability. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 627–634, 2016.

[34] Vijay A. Saraswat, George S. Almási, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar B. Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. 2010.

[35] Leyuan Shi and S. Olafsson. Hybrid equipartitioning job scheduling policies for parallel computer systems. In *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, volume 2, pages 1704–1709 vol.2, 1998.

[36] Garrick Staples. Torque resource manager. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[37] Olivier Tardieu. The apgas library: Resilient parallel and distributed programming in java 8. In *Proceedings of the ACM SIGPLAN Workshop on X10*, X10 2015, page 25–26, New York, NY, USA, 2015. Association for Computing Machinery.

[38] A.T. Wong, L. Oliker, W.T.C. Kramer, T.L. Kaltz, and D.H. Bailey. Esp: A system utilization benchmark. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 15–15, 2000.

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. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Kassel, 24. Januar 2024

Raoul W. Goebel