UNIKASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit

Performanceevaluierung des Java-Parallelisierungs-Frameworks APGAS mit dem Benchmark-System Task Bench

Torben R. Lahnor
35200226

Kassel, 8. März 2022

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Albert Zündorf

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

APGAS	Asynchronous Partitioned Global Address Space (Java Bibliothek [18])
CF	CompletableFuture. Java-Klasse aus dem `java.util.concurrent`-Paket
FLOP	Floating Point Operation
JIT	Just-In-Time Compilation
JVM	Java Virtual Machine
METG	Minimum Effective Task Granularity (Metrik aus [17])
X10	Parallele Programmiersprache [9]

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

_____

Kassel, 8. März 2022

Torben Lahnor

1. Einleitung

Mit dem Fortschritt der Digitalisierung wächst die Menge an Daten, die für alle möglichen Zwecke gesammelt und verwendet werden. Damit steigt auch der Bedarf an Rechenleistung, die zur Auswertung dieser Daten benötigt wird. Rechenzentren müssen den stetig wachsenden Anforderungen gerecht werden, unter anderem durch mehr und bessere Hardware. Die Programme, die auf Rechenzentren zur Datenverarbeitung ausgeführt werden, müssen dabei auch in der Lage sein, die bereits vorhandene Hardware möglichst gut auszunutzen. Dazu verwenden diese Programme eines der vielen verfügbaren Parallelisierungs-Frameworks. Diese Frameworks bieten eine Abstraktionsebene für die Programmierung und ermöglichen die effiziente Nutzung von Multi-Core-Architekturen und verteilten Rechenclustern [5]. Aus Sicht eines Entwicklers stellt sich bereits in der Anfangsphase der Programmentwicklung die Frage, welches dieser Frameworks genutzt werden soll. Dabei sind viele verschiedene Gesichtspunkte wie die Programmiersprache, Lizenzfragen und Unterstützung auf der Zielarchitektur zu beachten. Ein weiterer wesentlicher Faktor bei dieser Entscheidung ist die zu erwartende Performance.

Viele wissenschaftliche Arbeiten haben sich bereits mit der Konzeption von Benchmarks zur Messung der Performance dieser Frameworks beschäftigt. Ein Ansatz für Benchmarks sind Mini-Apps (auch Proxy-Apps genannt). Diese leiten sich von einem realen Anwendungsfall ab und bilden wichtige Performance-Charakteristiken einer „echten“ Anwendung ab. Zum Zweck des Benchmarkings wird bei der Implementierung von Mini-Apps aber auf komplexere Details verzichtet, die in einer echten Anwendung nicht vernachlässigbar wären [13]. Ein Beispiel dafür ist die Mini-App TeaLeaf, die eine Gleichung über die Wärmeleitung in einem zwei- oder drei-dimensionalen Netz auswertet [12].

Die Auswertung dieser Gleichung lässt sich auf mehrere Prozessoren und Rechner verteilen und setzt dabei auch ein gewisses Maß an Kommunikation zwischen diesen voraus, da die Berechnung dieser Gleichung von den jeweils benachbarten Knoten des Netzes abhängt. Eine solche Mini-App hat den Vorteil, relativ praxisnah zu sein, wodurch die Ergebnisse des Benchmarkings entsprechend aussagekräftig für das Anwendungsgebiet sind, welches von der Mini-App betrachtet wird. Diese Herangehensweise hat aber auch einige Nachteile. Eine Mini-App mit einem Framework zu implementieren geht mit einigem Programmieraufwand einher, und für jedes Framework muss eine eigene Implementierung geschrieben werden. Um einen Vergleich zwischen vielen Frameworks auf Basis eines solchen Benchmarks machen zu können, ist ein entsprechend großer Programmieraufwand erforderlich [17]. Dabei wird im Kern auch nur ein Anwendungsfall abgebildet, der nicht unbedingt repräsentativ für andere Anwendungsgebiete ist. Viele Frameworks haben unterschiedliche Stärken und Schwächen, die bei einem konkreten Anwendungsfall mehr oder weniger starken Einfluss auf die Performance haben können.

Um ein möglichst breites Spektrum an Testszenarien betrachten zu können, wird in dieser Arbeit das Benchmark-System Task Bench [17] genutzt, welches einen anderen Ansatz beim Benchmarking verfolgt. Bei Task Bench wird kein konkreter Anwendungsfall modelliert, sondern eine Abstraktion eines generischen parallelisierbaren Problems in Form eines Graphen betrachtet. Dieser Ansatz basiert auf der Annahme, dass sich Probleme dieser Art als azyklische, gerichtete Graphen darstellen lassen, deren Knoten sequentielle Arbeitspakete (sog. Tasks) abbilden. Auf dieser Annahme basieren auch Scheduling-Algorithmen, welche nach einer möglichst optimalen Verteilung von Tasks auf Multiprozessorsysteme suchen [7, 11]. Task Bench generiert Taskgraphen, deren Aufbau sich auf verschiedene Weisen konfigurieren lässt. Maßgebend ist vor allem der Graphtyp, der das grundlegende Abhängigkeitsschema zwischen den Tasks definiert. Task

Bench bezeichnet sich daher als einen parametrisierten Benchmark, wobei jeder der verfügbaren Graphtypen als ein einzelner Benchmark bezeichnet wird. Vorgestellt wurde Task Bench in einem Paper von Slaughter et al. [17], in dem auch Messergebnisse für verschiedene Parallelisierungs-Frameworks präsentiert werden. Die Graphtypen und weitere Parametern werden in Abschnitt 2.2 näher erläutert.

Um Benchmarks von Task Bench für ein bestimmtes Parallelisierungs-Framework auszuführen, muss zunächst ein entsprechendes Programm entwickelt werden. Ein solches Programm benutzt von Task Bench bereitgestellte Methoden, um zunächst einen Graphen nach gegebenen Parametern zu generieren. Danach werden die Tasks dieses Graphen — unter Beachtung der vorgegebenen Abhängigkeiten — durch Verwendung des Parallelisierungs-Frameworks ausgeführt. Die Laufzeit der Ausführung des gesamten Taskgraphen wird gemessen, wobei dies das Ergebnis des Benchmarks darstellt. Durch die abstrakte Betrachtung eines parallelisierbaren Problems als Graph kann Task Bench eine Reihe von unterschiedlichen Benchmarks (in Form der versch. Graphtypen) anbieten, die auf Basis des gleichen Programms für ein Parallelisierungs-Framework ausgeführt werden können. Im Folgenden wird ein solches Programm als Interface bezeichnet. Durch die verschiedenen Möglichkeiten, den Aufbau eines Taskgraphen zu konfigurieren, lassen sich verschiedene Szenarien abbilden und die Performance des Frameworks messen.

Das Thema dieser Arbeit ist die Messung der Performance der Java-Bibliothek APGAS [1], welche an der Universität Kassel weiterentwickelt und verwendet wird. Dazu wurde ein Interface entwickelt, welches APGAS nutzt, um Benchmarks von Task Bench auszuführen. Dieses Interface wird genutzt, um die Performance von APGAS zu messen. Anhand dieser Messungen wird herausgearbeitet, wie Task Bench angewandt werden kann, um Erkenntnisse über die Performance von APGAS in verschiedenen Szenarien zu gewinnen.

Zunächst werden in Abschnitt 2 die verwendeten Technologien sowie notwendiges Hintergrundwissen vorgestellt. Danach behandelt Abschnitt 3 die grundsätzliche Funktionsweise des Interfaces. Abschnitt 4 beschreibt die konkrete Implementierung des Interfaces und dessen Funktionsweise ausführlicher. In Abschnitt 5 werden Messergebnisse für verschiedene Szenarien präsentiert und mit den Ergebnissen des zugrundeliegenden Papers [17] zu Task Bench verglichen. Abschließend werden in Abschnitt 6 die wichtigsten Erkenntnisse zusammengefasst und Ausblicke auf die mögliche Weiterentwicklung des Interfaces gegeben.

2. Hintergrund

In diesem Kapitel wird die Java-Bibliothek APGAS, sowie das zugrundeliegende Parallelisierungs-Modell präsentiert. Außerdem wird Task Bench ausführlicher vorgestellt und die Metriken und Szenarios, die bei den Messungen in Kapitel 5 betrachtet werden, definiert und erläutert.

2.1. APGAS

Im Rahmen dieser Arbeit bezeichnet „APGAS“ die Java-Bibliothek (verfügbar unter [1]), die auf dem gleichnamigen Parallelisierungs-Modell basiert.

Das Parallelisierungs-Modell APGAS (Asynchronous Partitioned Global Address Space) ist als Erweiterung des PGAS-Modells entstanden. Das PGAS-Modell arbeitet mit sogenannten „Places“, welche als logische Einheit zur Abgrenzung von Speicher- und Arbeitsbereichen dienen. Die Speicherbereiche von Places sind voneinander isoliert, zwischen Places können Daten nur durch explizite Anweisungen ausgetauscht werden. Das APGAS-Modell erweitert dieses Modell um die Möglichkeit, Berechnungen asynchron auszuführen. Für eine Berechnung wird eine sogenannte Aktivität erstellt, die asynchron von einem der verfügbaren Worker-Threads ausgeführt wird. Worker-Threads sind place-spezifisch und können auf den gleichen Speicherbereich zugreifen [18].

Die APGAS-Bibliothek setzt das APGAS-Modell in Java um und verwendet den Fork/Join-Pool zur Verwaltung und Ausführung von Aktivitäten [16]. Eine APGAS-Anwendung startet zunächst in der `main`-Methode von Java, welches die Hauptaktivität darstellt. Von dieser Hauptaktivität aus können weitere Aktivitäten gestartet werden. Die Anwendung läuft, bis die Hauptaktivität

beendet ist. Das Starten einer Aktivität geschieht durch Verwendung der Funktion `async`. Diese Funktion erwartet als Parameter einen `SerializableJob`, welcher — ähnlich wie ein Java `Runnable` — üblicherweise als Lambda-Funktion ausgedrückt wird. Aktivitäten können Variablen aus dem umliegenden Code verwenden. Dabei gilt die Einschränkung, dass die verwendeten Variablen `final` oder `effectively-final` und außerdem serialisierbar sein müssen. Der Job wird dann zu dem Fork/Join-Pool des jeweiligen Places hinzugefügt und wird entweder sofort oder zu einem späteren Zeitpunkt ausgeführt [16]. Mithilfe eines `finish`-Blocks kann auf die Fertigstellung von Aktivitäten gewartet werden, die innerhalb des `finish`-Blocks gestartet wurden. Durch die Einteilung in Places können mehrere JVMs (Java Virtual Machines) miteinander verknüpft werden, die bspw. in einem Cluster auf mehreren Rechnern laufen, welche keinen gemeinsamen Speicher haben. Jeder Place stellt in diesem Fall einen Rechner dar. Durch die logische Aufteilung der Speicherbereiche kann so auch die physische Aufteilung der Rechner abgebildet werden. APGAS verwendet intern die Bibliothek Hazelcast [10, 18] zur Verknüpfung und Koordination zwischen JVMs. Durch die Funktion `asyncAt` lässt sich eine Aktivität auf einem anderen Place starten. Dabei ist zu beachten, dass die Variablen, die in einem `asyncAt` benutzt werden, serialisiert und ggf. über eine Netzwerkverbindung zu dem aufgerufenen Place übertragen werden müssen.

Wie diese Konzepte in dem APGAS-Interface für Task Bench angewendet wurden, ist in den Kapiteln 3 und 4 beschrieben. In dieser Arbeit wurde die gleiche APGAS-Version verwendet wie in [15]. Diese befindet sich auch auf der beiliegenden CD als .jar-Datei. Es wurde Java-Version 11 verwendet, weitere Details zum Kompilieren und zur Ausführung sind auf der CD dokumentiert.

2.2. Task Bench

Die Basis für diese Arbeit ist ein Paper der Stanford University [17], in welchem das Benchmark-System Task Bench vorgestellt wird.

2.2.1. Konzept

Task Bench abstrahiert eine parallelisierbare Anwendung durch eine Menge an sequentiellen Arbeitspaketen (im Folgenden „Tasks“ genannt). Ein Task umfasst dabei einen Kernel, der aus einer Reihe von Operationen besteht, die ausgeführt werden müssen, um den Task abzuschließen. Außerdem haben Tasks untereinander Abhängigkeiten. Diese Abhängigkeiten sind in der Praxis durch ihre Eingabe- und Rückgabewerte gegeben. Als Eingabe braucht ein Task die Rückgabewerte seiner Vorgängertasks, ein Task darf (nach Vorgabe von Task Bench) erst gestartet werden, wenn dessen Vorgängertasks abgeschlossen und ihre Rückgabewerte vorhanden sind. Die Tasks mit ihren Abhängigkeiten lassen sich als Graph darstellen (im Folgenden Taskgraph genannt). Die Taskgraphen von Task Bench sind azyklische, gerichtete Graphen, wobei die Abhängigkeiten durch die Kanten und deren Richtung darstellt werden. In Abbildung 2.1 ist ein Taskgraph mit dem Graphtyp *stencil* dargestellt. Die Pfeile in dieser Abbildung zeigen jeweils von dem Vorgängertask zu dem abhängigen Task.

Ein Graph ist in Zeitschritte (sogenannte time steps) eingeteilt, in Abbildung 2.1 sind diese auf der y-Achse aufgetragen. Tasks auf der gleichen horizontalen Ebene befinden sich also im gleichen time step. Tasks können nur direkte Abhängigkeiten auf den jeweils vorherigen time step haben, dadurch ist der Graph azyklisch. Auf der x-Achse lassen sich die Tasks außerdem in Spalten zusammenfassen, ein Task ist bei jedem Graphtypen immer von dem jeweils vorherigen Task der gleichen Spalte abhängig. Einzeln betrachtet ist eine Spalte daher eine Reihe von Tasks, die

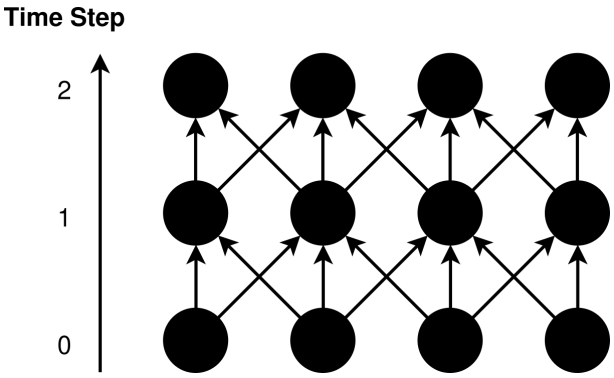


Abbildung 2.1.: Darstellung eines Taskgraphen des Typs stencil

sequentiell ausgeführt werden und niemals parallel zueinander ausgeführt werden können. Die time steps implizieren somit eine zeitliche Abfolge, geben diese aber nicht strikt vor, der Task in time step 2 der letzten Spalte (der oberste rechte Task in der Abbildung) kann bspw. auch vor dem Task in time step 0 der ersten Spalte (der unterste linke Task) ausgeführt werden, da diese nicht voneinander abhängig sind.

Eine wichtige Eigenschaft dieser Tasks ist, dass der Kernel, der den tatsächlichen Rechenaufwand darstellt, in C geschrieben ist. Dieser C-Code ist der sogenannte „Task Bench Core“ (verfügbar unter [2]) und wird von allen Task Bench Interfaces verwendet. Damit sind die Kernels unabhängig vom jeweiligen Parallelisierungs-Framework. Die Kernels müssen für ein Framework also auch nicht extra implementiert werden. Ein Interface muss nur in der Lage sein, diesen Code aufzurufen. Dadurch ist (laut [17]) ein fairerer Vergleich zwischen verschiedenen Frameworks möglich. Kapitel 5 befasst sich u.a. mit dem Vergleich von APGAS mit anderen Frameworks.

Des Weiteren gibt es verschiedene Konfigurationsmöglichkeiten, mit denen sich der Aufbau der Taskgraphen und die Kernels einstellen lassen. Im folgenden Abschnitt werden die Konfigurationsmöglichkeiten von Task Bench erläutert.

2.2.2. Konfiguration

Der von Task Bench generierte Taskgraph lässt sich durch die folgenden Parameter konfigurieren.

- `width`: Die maximale Breite des Graphen in einem time step.
- `steps`: Die Anzahl der time steps bzw. Höhe des Graphen.
- `graph type`: Der Typ des Graphen. Dieser gibt das Abhängigkeitsschema und die Form des Graphen an.
- `kernel`: Der Kernel, der in jedem Task ausgeführt wird. Dieser hat selbst einige Konfigurationsmöglichkeiten.
  - `kernel type`: Zur Auswahl stehen u.a. `compute-bound`, `memory-bound`, `load-imbalance` und `communication-bound`.
  - `iter`: Die Anzahl an Iterationen der Berechnung, die in dem Kernel ausgeführt wird.

Dabei ist anzumerken, dass in jedem Task eines erstellten Graphen der gleiche Kernel mit der gleichen Konfiguration verwendet wird[1]. Der Kerneltyp erlaubt es festzulegen, welcher Aspekt der Hardware den Bottleneck bei der Ausführung bilden soll. Es gibt noch weitere Konfigurationsmöglichkeiten für die verschiedenen Kerneltypen, welche für diese Arbeit aber nicht relevant sind, da ausschließlich der `compute-bound`-Kernel betrachtet wird. Bei `compute-bound` ist der Bottleneck die Rechenleistung der CPU. Die Anzahl an Iterationen bestimmt den Rechenaufwand eines einzelnen Tasks und beeinflusst somit auch stark die gesamte Ausführungsdauer eines Graphen.

[1] Der `load-imbalance`-Kernel stellt in gewisser Weise eine Ausnahme dazu dar, die Anzahl an Iterationen ist pro Task unterschiedlich (daher kommt Imbalance zu stande).

Der Graphtyp ist maßgeblich für die Komplexität und Anzahl der Abhängigkeiten zwischen den Tasks. Die verschiedenen Typen eignen sich, um unterschiedliche Szenarien darzustellen, die bei anderen Benchmarking-Ansätzen von einer eigenen Mini-App simuliert werden würden. Wie bereits in der Einleitung erwähnt, liegt die Besonderheit von Task Bench darin, dass sich durch eine einzige Implementierung viele verschiedene Benchmarks ausführen und testen lassen. Im Rahmen dieser Arbeit wird lediglich der Typ *stencil* betrachtet.

Die Breite des Graphen bestimmt, wie viele Tasks maximal in einem time step sein können. In anderen Worten ist dies die Anzahl an Spalten im Taskgraphen. Bei *stencil* und den meisten anderen Graphtypen sind in jedem time step die maximale Anzahl an Tasks vorhanden, bei einigen Graphtypen (z.B. *tree*) ändert sich die Anzahl an Tasks pro time step allerdings. Da die Tasks in einer Spalte unabhängig vom Graphtypen sequentiell ausgeführt werden müssen, limitiert die Breite die Anzahl an Tasks, die parallel ausgeführt werden können. Wenn die Breite geringer als die Anzahl an verfügbaren CPU-Cores (bzw. Threads) ist, wird die maximale Auslastung der Hardware verhindert. Die Breite des Graphen multipliziert mit der Anzahl an time steps ergibt die Anzahl an Tasks in einem Graphen des Typs *stencil*. Die Anzahl an time steps zu erhöhen, verändert das Testszenario an sich nicht so stark, da lediglich weitere Tasks „angehangen“ werden. Eine gewisse Anzahl an time steps ist aber oft notwendig, um verlässliche Ergebnisse zu bekommen.

Im Folgenden wird eine Kombination von Breite, time steps, Kerneltyp und Iterationen (bzw. Task-Größe) als eine Konfiguration bezeichnet. Mehrere Konfigurationen können zu einem Szenario zusammengefasst werden, wobei die Konfigurationen sich innerhalb eines Szenarios nur um ein oder zwei der Parameter unterscheiden. Wie solche Szenarios aussehen können und was durch die Veränderung der Konfigurationen gemessen werden kann, wird im folgenden Abschnitt erläutert.

2.3. Metriken

Bei einem Benchmark ist es zunächst wichtig zu definieren, was gemessen werden soll. Maßgebend dafür ist die Frage: Was macht ein gutes Parallelisierungs-Modell aus? Grundsätzlich ist eine Anforderung, dass ein parallelisierbares Problem möglichst schnell gelöst bzw. berechnet werden soll. Dies wird dadurch erreicht, dass die Hardware möglichst effizient genutzt wird. Eine 100%-ige Effizienz wäre in der Theorie gegeben, wenn alle Recheneinheiten der Hardware durchgehend zu 100% ausschließlich mit der Berechnung des Problems ausgelastet wären, was in der Praxis aber nicht der Fall ist. Faktoren, die die Effizienz verringern, sind vor allem Overhead — also Rechenaufwand, der nicht direkt mit dem eigentlichen Problem zusammenhängt — und suboptimale Aufgabenverteilung, die dazu führt, dass nicht immer alle Prozessorkerne mit Arbeit ausgelastet sind.

Bei Task Bench werden nach der Ausführung eines Benchmarks drei Werte ausgegeben: die Laufzeit des Benchmarks, die Anzahl an Gleitkommaoperationen pro Sekunde (FLOP/s bzw. FLOP-Rate) und die verarbeiteten Bytes pro Sekunde (B/s). Letzterer Wert ist nur relevant für den `memory bound`-Kernel, der im Rahmen dieser Arbeit nicht betrachtet wird. Zur Bewertung der Performance von verschiedenen Konfigurationen eignet sich die Laufzeit nur bedingt, da diese bei größeren Konfigurationen entsprechend länger ist und immer im Kontext der Problem-Größe[2] bewertet werden muss. Die FLOP-Rate hingegen errechnet sich aus der Problem-Größe (in FLOP-Anzahl) geteilt durch die Laufzeit; somit ist der Wert abhängig von der Problem-Größe und lässt sich daher besser mit den Ergebnissen von anderen Konfigurationen vergleichen.

[2] Die Problem-Größe einer Konfiguration ergibt sich aus der Anzahl an Tasks im Task-Graphen, multipliziert mit der FLOP-Anzahl pro Task. In [17] wird mit „problem-size“ die Laufzeit eines einzelnen Tasks bezeichnet.

Im Rahmen dieser Arbeit wird unter dem Begriff „Effizienz“ die „Applikations-Effizienz“ verstanden. Diese definiert die Höchstleistung auf einem System anhand der praktisch erreichbaren Leistung [14]. Die Effizienz bei einer Messung ist dann gegeben als:

$$\frac{gemessene\ FLOP/s}{Höchstleistung\ in\ FLOP/s}$$

Um diese Höchstleistung bestimmen zu können, muss zunächst ein möglichst ideales Szenario gefunden werden, bei dem die Laufzeit größtenteils durch die Laufzeit der Tasks bestimmt ist. Dies lässt sich dadurch erreichen, dass ein Kernel mit einer hohen Anzahl an Iterationen verwendet wird, sodass jeder einzelne Task eine lange Laufzeit hat und der Overhead — der u.a. von der Anzahl an Tasks abhängig ist — in Relation dazu sehr gering ist. Die FLOP-Rate, die mit dieser „optimalen“ Konfiguration erreicht wird, stellt nach der Definition der Applikations-Effizienz die Höchstleistung auf der gegebenen Hardware dar.

Eine pauschale Aussage darüber, wie „gut“ ein Parallelisierungs-Modell ist, lässt sich allein anhand der Effizienz bei einem einzelnen Szenario allerdings nicht treffen.

2.3.1. METG

Task Bench führt eine eigene Metrik ein, die minimum effective task granularity (METG). Die Definition lautet: „METG(50%) für [ein Szenario] A ist die kleinste durchschnittliche Task-Granularität (bzw. Task-Laufzeit), mit der A insgesamt eine Effizienz von mindestens 50% erreicht“ [17]. METG ist eine parametrisierte Metrik. Daher lässt sich die Definition analog mit anderen Effizienz-Werten anwenden. In [17] und in dieser Arbeit wird aber ausschließlich METG(50%) betrachtet.

Um METG(50%) für ein Szenario zu bestimmen, wird zunächst die Effizienz einer Konfiguration dieses Szenarios mit einer hohen Anzahl an Iterationen — also einer groben Granularität — gemessen. Dabei ist zu erwarten, dass diese Konfiguration annähernd eine Effizienz von 100% erreicht, da der Overhead noch sehr gering sein sollte. Es werden weitere Messungen dieses Szenarios durchgeführt, wobei die Task-Größe sukzessiv verringert wird. Je feiner die Granularität wird, desto geringer wird erwartungsgemäß auch die Effizienz, da der Overhead im Verhältnis zur Gesamtausführungszeit größer wird. Die Task-Größe, bei der die Effizienz noch knapp über 50% liegt, wird als METG(50%) bezeichnet. Diese Metrik soll dazu dienen, die Effizienz eines Frameworks für eine Anwendung anhand der Granularität der Tasks vorhersagen zu können [17].

Der Wert METG(50%) kann für eine Framework laut [17] allerdings aufgrund von verschiedenen Faktoren variieren. Als Gründe werden u.a. die genutzte Hardware und die konkrete Implementierung genannt. Es ergibt sich also eher eine Tendenz bzw. ein Bereich für die Task-Granularität, bei der für ein Framework Einbußen in der Performance zu erwarten sind.

2.3.2. Skalierbarkeit

Bei der Bewertung von Parallelisierungs-Frameworks ist auch die Skalierbarkeit entscheidend. Dabei geht es um die Ausführung einer Anwendung auf einer wachsenden Anzahl an Rechnern (im Folgenden „Knoten“ oder „Nodes“ genannt) und die Betrachtung des erreichten Speedups. Der Speedup bezeichnet den Faktor, mit dem sich die Laufzeit einer Anwendung bei Ausführung auf mehreren Knoten, im Vergleich zu der Ausführung auf einem einzelnen Knoten, verbessert. Allgemeinen wird der Speedup im Hinblick auf die Anzahl an Prozessor-Kernen oder Threads betrachtet. Da alle Knoten des Clusters, auf denen die Messungen in Kapitel 5 ausgeführt wurden, über die gleichen Anzahl und Art von Prozessoren

verfügen[3], lässt sich dieses Konzept analog bei der Skalierung auf mehrere Knoten anwenden. Eine Anwendung hat in der Theorie einen gewissen Anteil an parallelisierbarem Code und nicht-parallelisierbarem Code. Der praktisch erreichbare Speedup ist begrenzt durch das Verhältnis zwischen den beiden Anteilen [6, 8].

Das Verhältnis zwischen diesen Teilen einer Anwendung hängt bei Task-Bench-Szenarios vor allem von der Granularität der Tasks ab. Das exakte Verhältnis wird hier aufgrund der Praktikabilität nicht betrachtet. In [17] wurden mit dem MPI-Interface einige Messungen zum strong-scaling und weak-scaling gemacht. Daher befasst sich diese Arbeit ebenfalls mit diesen beiden häufig verwendeten Konzepten.

Strong-Scaling

Das bekannte „Amdahl's Law“ [6] beschreibt die Skalierbarkeit einer Anwendung mit einer fixen Größe. Daraus leitet sich die Metrik des „strong-scaling“ ab, wobei der tatsächliche Speedup bei der Ausführung einer Anwendung auf einer wachsenden Zahl Prozessoren (bzw. Knoten) gemessen wird. Dadurch verringert sich die Menge an Arbeit, die auf jedem einzelnen Knoten ausgeführt wird. Der Speedup ist dadurch begrenzt, dass durch die Aufteilung der Arbeit auf mehrere Knoten auch zusätzlicher Scheduling- und Kommunikationsaufwand entsteht. Bei Task Bench bedeutet dies, das für alle Messungen eines Strong-Scaling-Szenarios die gleiche Konfiguration verwendet wird.

[3] jeweils zwei AMD Opteron 2676-Prozessoren

Weak-Scaling

Ähnlich bekannt wie Amdahl's Law ist der Ansatz von Gustafson, der davon ausgeht, dass „Die Problem-Größe in der Praxis mit der Anzahl an Prozessoren skaliert“ [8]. Dieser Ansatz wird beim sogenannten „weak-scaling“ angewendet. Bei den Szenarios für Task Bench wächst die Breite des Taskgraphen zusammen mit der Menge an Knoten, sodass bei allen Konfiguration eines Szenarios die gleiche Menge an Arbeit pro Knoten ausgeführt werden muss. Anders als beim strong-scaling ist hier also keine Verbesserung der Laufzeit zu erwarten, sondern im Idealfall bleibt die Laufzeit bei den Messungen innerhalb eines Szenarios gleich. Der Speedup besteht darin, dass die Laufzeit nicht so stark ansteigt, wie die Breite — und dementsprechend die Größe — des Taskgraphen. Je mehr die Laufzeit bei der Skalierung ansteigt, desto schlechter ist die Skalierbarkeit bei dem jeweiligen Szenario.

3. Überblick zum Programmablauf

Eine erste Implementierung eines APGAS-Interfaces für Task Bench wurde im Rahmen einer Projektarbeit erstellt, die lediglich den Graphtypen *trivial* unterstützte. Bei diesem Graphtypen hat jeder Task nur eine Abhängigkeit, nämlich auf den Task derselben Spalte im jeweils vorhergehenden time step. Diese Implementierung beschränkte sich auf die Erarbeitung und Umsetzung der grundsätzlichen Funktionsweise und war nur für die Ausführung auf einem einzelnen Rechner geeignet. Es gab außerdem einige weitere Probleme, die bei der Projektarbeit offen blieben.

Das Ziel dieser Bachelorarbeit war es, den *stencil* Benchmark auf dem Cluster der Universität Kassel mit mehreren Knoten auszuführen, um die Performance von APGAS bei der Aufgabenverteilung auf mehrere Places auszuwerten. Dazu sollte die Abhängigkeitsstruktur unterstützt werden, die beim trivialen Graphen-Typen nicht vorhanden ist (jede Spalte wird unabhängig von den anderen Spalten sequentiell ausgeführt). Des Weiteren sollten Messungen durchgeführt werden, die sich mit den Messwerten der anderen Frameworks vergleichen lassen. Interessant ist dabei vor allem die parallele Programmiersprache X10, da X10 ebenfalls auf dem APGAS-Modell basiert Im Paper [17] wurden auch Messwerte zu X10 publiziert.

Der Programmablauf beginnt damit, dass dem Interface einige Parameter übergeben werden. Diese beinhalten zum einen Einstellungen für APGAS, um die Anzahl an Places und Threads festzulegen, zum anderen die Konfigurationsoptionen für Task Bench, die in Abschnitt 2.2.2 aufgelistet sind. Das Interface startet zunächst mit der Hauptaktivität, die nur auf einem Place läuft, der im folgenden als der „Hauptplace“ bezeichnet wird. Die anderen Places (falls vorhanden) warten zunächst darauf, dass durch `asyncAt` eine Aktivität

auf ihnen gestartet wird. Am Anfang wird der Taskgraph initialisiert. Dies geschieht durch Funktionen aus dem Task Bench Core [2], die die übergebenen Konfigurationsparameter verwenden. Darüber hinaus werden im Interface einige Datenstrukturen und Variablen initialisiert, die im weiteren Verlauf auf den verschiedenen Places benötigt werden. Dabei wird unter anderem festgelegt, welcher Place für welche Tasks zuständig sein soll. Jeder Place bekommt ungefähr die gleiche Anzahl an Tasks zugeteilt (durch Rundung entstehen ggf. leichte Ungleichheiten). Sobald die Initialisierung abgeschlossen ist, beginnt der eigentliche Benchmark. Die Initialisierung selbst wird also nicht mitgemessen. Dies ist bei sämtlichen Task Bench Interfaces der Fall.

Der Hauptplace initialisiert die einzelnen time steps nacheinander. Dazu müssen in jedem time step zunächst die Abhängigkeiten der einzelnen Tasks berechnet bzw. zusammengestellt werden. Um die Abhängigkeiten eines Tasks aus dem Taskgraphen einzulesen, werden Methode aus dem Task Bench Core [2] verwendet. Daraus wird vom Interface eine Liste von Tasks aus dem jeweils vorherigen time step erstellt. Danach wird auf jedem Place mit einem `asyncAt` eine Aktivität gestartet, die die entsprechenden Tasks mit den jeweiligen Abhängigkeiten enthält. Die Aktivität besteht daraus, die Tasks auszuführen, nachdem die Abhängigkeiten eines jeweiligen Tasks abgeschlossen sind. Es ist an dieser Stelle noch einmal anzumerken, dass das Starten einer Aktivität nicht bedeutet, dass sofort mit der Ausführung dieser Aktivität begonnen wird. Somit wird bereits mit der Berechnung der Tasks begonnen, während die restlichen Aktivitäten noch gestartet werden. Da die verschiedenen Places keinen gemeinsamen Speicher haben, muss der Rückgabewert eines Tasks ggf. an andere Places weitergegeben werden, die das Ergebnis dieses Tasks für eigene Tasks brauchen. Dies geschieht ebenfalls über `asyncAt`-Aufrufe. So werden nach und nach alle time steps initialisiert und die Aktivitäten zum Ausführen der Tasks auf den verschiedenen Places gestartet. Der Hauptplace selbst führt ebenfalls Tasks aus und wartet

schließlich darauf, dass alle Tasks abgeschlossen sind. Wenn alle Places mit ihren Tasks fertig sind, gibt der Hauptplace über eine Task-Bench-Funktion die Laufzeit, die durchschnittlichen FLOP/s (bzw. B/s), sowie die Konfiguration des Taskgraphen aus.

4. Implementierung

Nach der Beschreibung des grundsätzlichen Ablaufs in Kapitel 3 befasst sich dieses Kapitel mit dessen konkreter Umsetzung im Code.

4.1. Ausgangspunkt

Wie in Kapitel 3 erwähnt, wurde ein grundlegendes Interface für Task Bench im Rahmen einer Projektarbeit entwickelt. Als Basis für das APGAS-Interface wurde der Code des Task Bench Interfaces für Spark verwendet, welches in Scala und Java geschrieben ist. Scala-Code lässt sich relativ leicht in Java umschreiben, so konnte mit wenig Aufwand ein funktionierendes APGAS-Interface implementiert werden.

Wie bereits in Abschnitt 2.2.1 erwähnt wurde, wird für jedes Interface der Task Bench Core benötigt, welcher in C geschrieben ist. Zum Task Bench Core gehört Code zum Erstellen des Taskgraphen, sowie zum Ausführen der Tasks bzw. Kernels. Das Spark-Interface ist — wie der Task Bench Core und viele weitere Interfaces — im GitHub-Repository von Task Bench zu finden (siehe [2]). Um den C-Code in Java aufrufen zu können, wurde das Tool „swig“ [4] verwendet. Dieses Tool wurde auch bei den Task Bench Interfaces für Spark und Swift genutzt. Aus den Core-Dateien in C werden damit Java-Klassen generiert, welche die Funktionen der C-Klassen abbilden und in der Implementierung des APGAS-Interfaces aufgerufen werden können.

4.2. Weiterentwicklung

Um die weiteren Anforderungen zu erfüllen, die in Kapitel 3 beschrieben wurden, musste eine Möglichkeit gefunden werden, das Abhängigkeitsschema von Task Bench mit APGAS umzusetzen. Das Problem dabei ist, dass die Async-Finish-Struktur an sich nicht erlaubt, an verschiedenen Stellen auf bestimmte Asyncs zu warten, sondern Asyncs nur zusammen gruppiert werden können, um auf sie gemeinsam zu warten. Um mit einem Finish auf eine Gruppe auf Asyncs zu warten, müssen sie also auch an der gleichen Stelle erstellt werden. Da die Abhängigkeitsstruktur von Task Bench aber voraussetzt, dass das Ergebnis eines Tasks an mehreren Stellen gebraucht wird und jeder Task von mehreren Tasks abhängig ist, musste eine weitere Datenstruktur zum Speichern und Synchronisieren der Ergebnisse verwendet werden.

4.2.1. Tasks als Completable-Futures

Die Klasse `CompletableFuture` (im Folgenden mit „CF“ abgekürzt) aus dem `java.util.concurrent` Paket wurde verwendet, um einen Task und dessen Ergebnis darzustellen. Bis das Ergebnis eines Tasks errechnet ist, gilt das dazugehörige CF als unvollständig und die Tasks, die dieses Ergebnis brauchen, können auf dessen Vervollständigung warten. Der Kernel eines Tasks gibt als Ergebnis ein Byte-Array zurück, welches den time step und die Spalte im Taskgraphen kodiert. Task Bench benutzt diese Kodierung intern, um beim Aufruf eines Tasks festzustellen, ob dieser auch die richtigen Eingaben erhalten hat. So wird sichergestellt, dass das Interface den Taskgraphen korrekt ausführt.

Entsprechend dem zweidimensionalen Aufbau eines Taskgraphen, werden die CFs in einem zweidimensionalen Array namens *completableValues* gespeichert, wobei der äußere Index den time step und der innere Index die Spalte des Graphen

darstellt. Genau genommen wird eine Wrapper-Klasse verwendet, die nur ein Feld mit dem Typen `CompletableFuture<byte>` hat, um Typsicherheit innerhalb der Arrays zu gewährleisten. Im Folgenden wird nicht zwischen der Wrapper-Klasse und der CF-Klasse unterschieden. *completableValues* wird vor dem Start des Benchmarks auf jedem Place initialisiert und mit leeren bzw. unvollständigen CFs gefüllt. Da die Places keinen gemeinsamen Speicher haben, hat jeder Place eine eigene Instanz von *completableValues*, die nur Daten zu den Tasks des Places und den benötigten Abhängigkeiten enthält.

4.2.2. Serialisierung der Abhängigkeiten

Wie in Abschnitt 2.1 erwähnt, müssen alle Variablen die einem `async` oder `asyncAt` übergeben werden, serialisierbar sein. Da CFs aber nicht serialisierbar sind, wurde eine weitere Klasse verwendet, um die Abhängigkeiten so zu modellieren, dass sie auf dem Hauptplace berechnet und dann an die anderen Places übergeben werden konnten. Diese Klasse heißt *AsyncPointVal* und enthält nur die Indexe der *completableValues*-Datenstruktur. Somit funktioniert diese Klasse quasi wie eine Referenz auf einen Task. Dadurch muss auch nur eine relativ kleine Datenmenge übertragen werden, da die Abhängigkeiten als eine Liste von Indexpaaren übergeben werden und der jeweilige Place die entsprechenden CFs aus der eigenen Datenstruktur nehmen kann.

4.2.3. Starten der Tasks

Listing 4.1 führt den Code-Ausschnitt auf, in dem die Tasks eines time steps auf einem Place gestartet werden. Dieser Ausschnitt wird in jeden time step jeweils einmal für jeden Place ausgeführt, um alle Tasks zu starten. Die Variable *inputs* enthält dabei die Abhängigkeiten aller Places für diesen time step. In den Zeilen 2—5 werden die Abhängigkeiten für den jeweiligen Place (der in *placeId*

angegeben ist) in die Variable *relevantVals* kopiert. Dadurch werden beim Aufruf von `asyncAt` nur die notwendigen Daten an den Place geschickt.

In Zeile 6 wird auf dem Place eine Aktivität gestartet, diese wird als Lambda-Funktion übergeben. Die Aktivität selbst besteht daraus, zunächst über alle zugeteilten Tasks zu iterieren und die jeweiligen Abhängigkeiten zusammenzufassen. Dies geschieht in der Methode *wrapDependenciesInFuture*, die mit `AsyncPointVals` in die CFs der eigenen Instanz von *completableValues* umwandelt, und zu einem neuen CF kombiniert. Dieses CF wird dann genutzt, um die Ausführung des Tasks selbst vorzubereiten.

Die Methode `thenRunAsync` in Zeile 13 sorgt dafür, dass die als Lambda übergebene Aktivität zum Fork-Join-Pool hinzugefügt wird, sobald alle Abhängigkeiten des Tasks abgeschlossen wurden. Innerhalb dieses Lambdas wird der Task in einer ausgelagerten Methode synchron ausgeführt. Nach dem Abschluss des Tasks wird eine weitere Methode in Zeile 22 aufgerufen, die das Ergebnis des Tasks an *completableValues* propagiert, ggf. auch an die Instanzen auf anderen Places.

Ein Problem bei der Implementierung war zunächst, dass mit der Methode `join` auf den Abschluss der Abhängigkeiten gewartet wurde, statt `thenRunAsync` zu benutzen. Der Unterschied dabei ist, dass jeder `join`-Aufruf einen Thread belegt. Dies war zum einen relativ ineffizient, da Threads in dieser Zeit nicht zur Berechnung von Tasks benutzt werden konnten, zum anderen führte es bei größeren Konfigurationen auch zu einem Deadlock, da alle Threads irgendwann im Wartezustand waren. Da `thenRunAsync` Threads nicht blockiert, konnten damit diese beiden Probleme behoben werden.

```
1  final List<AsyncPointVal>[] relevantVals =
2    new List[placeInputLengths[placeId]];
3  System.arraycopy(inputs, placeStartIndexes[placeId], relevantVals,
4    0, placeInputLengths[placeId]);
5
6  asyncAt(places().get(placeId), () -> {
7    ...
8    for (int relevantValsIndex = start; relevantValsIndex < end;
       relevantValsIndex++) {
9      CompletableFuture<Void> dependencies = wrapDependenciesInFuture(
10       timeStep, relevantVals[relevantValsIndex]
11     );
12     final int finalIndex = relevantValsIndex;
13     dependencies.thenRunAsync(() -> {
14       final byte[] result = Common.callExecutePoint(
15         taskGraphs[graphNumber],
16         timeStep,
17         finalIndex + placeStartIndex,
18         relevantVals[finalIndex],
19         false,
20         true
21       );
22       completeValues(
23         result, timeStep + 1, relevantVals.length,
24         finalIndex, finalIndex + placeStartIndex
25       );
26       ...
27     });
28   }
29 });
```

Listing 4.1: Schleife zum Starten von Tasks auf Place

4.2.4. Terminierungserkennung

Task Bench selbst stellt keine Funktionen bereit, die überprüfen, ob ein Benchmark korrekt abgeschlossen wurde. Es werden lediglich die Eingaben beim Starten eines Tasks validiert, es wird nicht überprüft, ob alle Tasks

eines Graphen berechnet wurden. Insbesondere wird der letzte Zeitschritt gar nicht validiert, da dessen Ergebnisse nicht mehr verwendet werden. Bevor der Benchmark und die Zeitmessung beendet werden können, muss noch explizit auf die Fertigstellung aller Tasks auf allen Places gewartet werden. Dadurch, dass die Aktivitäten in Listing 4.1 nicht die Tasks darstellen (die Aktivitäten selbst erstellen nur das CF zum Starten des Tasks), kann auch nicht durch ein `finish` auf die Fertigstellung der Tasks gewartet werden. Auch für diesen Zweck kommen Java-Synchronisations-Mechanismen zum Einsatz. Jeder Place zählt mit einem `AtomicInteger` die Menge an offenen Tasks im letzten time step. Der Hauptplace verfügt über ein `CountDownLatch`, welches mit der Anzahl an Places initialisiert wird. Beim Abschluss eines Tasks im letzten time step wird das `AtomicInteger` dekrementiert, wenn der Wert 0 erreicht ist, bedeutet das, dass alle Tasks auf diesem Place abgeschlossen sind. Es wird dann über ein `asyncAt` das `CountDownLatch` auf dem Hauptplace dekrementiert. Der Hauptplace selbst wartet nach dem Starten aller Tasks darauf, dass dieses Latch den Wert 0 erreicht. Wenn das Latch diesen Wert erreicht hat, sind alle Tasks abgeschlossen und der Benchmark wird beendet.

Anschließend können die Ergebnisse des letzten time steps noch optional verifiziert werden. Dazu wurde eine Funktion geschrieben, die die Werte in allen Places überprüft und zum einen sicherstellt, dass für jeden Task ein Ergebnis vorliegt und zum anderen, dass die Werte auch stimmen. Diese Funktionalität wurde vor allem während der Entwicklung genutzt, da so Fehler bei der Implementierung schnell aufgedeckt werden konnten, die sonst erst später aufgefallen wären. Diese Verifizierung findet nach der Zeitmessung statt und wird auch nur ausgeführt, wenn der JVM beim Starten ein entsprechender Parameter übergeben wird.

4.3. Offene Themen

Das Interface wurde mit dem Ziel entwickelt, den Graphtypen *stencil* ausführen zu können, was auch erreicht wurde. Um die Komplexität zu minimieren, wurde jedoch auf die notwendige Generalität verzichtet, um auch andere Graphtypen ausführen zu können. Konkret fehlt die Möglichkeit, eine beliebige Menge an Tasks in der *wrapDependenciesInFuture*-Methode in Listing 4.1 zusammenzufassen, sowie das Weitergeben von Ergebnissen an sämtliche Places innerhalb der Methode *completeValues*. Da bei *stencil* nur benachbarte Tasks voneinander abhängig sein können (siehe Abbildung 2.1), wurde dies entsprechend trivial implementiert, sodass nur benachbarte Places miteinander kommunizieren. Um dies generisch zu machen, müssten die Abhängigkeiten eines Tasks zusätzlich in *completeValues* betrachtet werden, um alle abhängigen Places benachrichtigen zu können.

Ein Detail, auf welches bei der Implementierung verzichtet wurde, ist das Durchführen eines „Warmup-Durchlaufs“. Bei den anderen Interfaces zu Task Bench (siehe [2]) wird bei jeder Ausführung eines Benchmarks der dazugehörige Taskgraph zweimal hintereinander ausgeführt. Der erste Durchlauf soll dabei dazu dienen, mögliche „just in time compilation“ (JIT) durchzuführen, die für Overheads sorgen könnten. Es wird nur die Laufzeit des zweiten Durchlaufs gemessen, sodass die Overheads durch JIT das Messergebnis nicht beeinträchtigen sollten. Dies war zunächst auch in dem APGAS-Interface eingebaut, doch es ließen sich zumindest bei Messungen mit dem Graphtypen *trivial* keine Unterschiede feststellen. Um Zeit bei der Ausführung zu sparen, wurde dieser Warmup also weggelassen. Ggf. könnte das wieder eingebaut werden, um zu untersuchen, ob sich bei anderen Graphtypen oder bestimmten Konfigurationen ein Unterschied messen lässt.

Task Bench bietet auch die Möglichkeit, mehrere Taskgraphen in einem Benchmark auszuführen, was es ermöglichen soll, diverse Anwendungsfälle zu simulieren, da sich dadurch auch die verschiedenen Graph- und Kerneltypen kombinieren lassen. Dies könnte ebenfalls in dem Interface eingebaut werden. Prinzipiell müssten dazu einige der benutzten Datenstrukturen in einem Array, mit den Taskgraphen als Dimension, bereitgestellt werden. Dabei müsste auch die Verteilung der Tasks der verschiedenen Graphen auf Places geklärt werden.

5. Messungen

Mit dem Interface wurden verschiedene Messungen auf dem Cluster der Universität Kassel durchgeführt. Dabei wurde die Public-Partition verwendet, deren Knoten jeweils über zwei AMD Opteron 6276-Prozessoren verfügen. Die durchgeführten Messungen orientieren sich an den Szenarios, die im Paper zu Task Bench betrachtet wurden [17], um die Ergebnisse damit vergleichen zu können. Die Messungen in [17] wurden auf einer deutlich leistungsstärkeren Hardware durchgeführt. Dies hat vor allem zur Folge, dass längere Laufzeiten für die verschiedenen Messungen zu erwarten waren. Bei der Auswertung der Messungen werden hauptsächlich die Effizienz und die Skalierbarkeit betrachtet, worauf diese Unterschiede keinen großen Einfluss haben sollten. Zu jeder Konfiguration eines Szenarios wurden jeweils 5 Messungen durchgeführt. In den abgebildeten Graphen wird der jeweilige Mittelwert aus diesen Messungen dargestellt. Damit sollten möglichst repräsentative Ergebnisse erreicht werden.

5.1. Bestimmung der Höchstleistung für Applikations-Effizienz

Wie in Abschnitt 2.3 erläutert, musste zunächst die höchste praktisch erreichbare Leistung gemessen werden, um diese als Maximalwert zur Berechnung der Applikations-Effizienz festzulegen. Dazu wurde eine Konfiguration mit dem Graphtypen *stencil*[1] gesucht, mit der eine möglichst gute Performance erreicht werden kann. Dazu wurde zunächst die höchste in Task Bench zulässige Task-Größe von 2^{30} gewählt. Je größer die einzelnen Tasks sind, desto geringer ist

[1] Prinzipiell könnte auch ein anderer Graphtyp verwendet werden. Die aktuelle Version des APGAS-Interfaces unterstützt außer *stencil* nur den Graphtypen *trivial*, der etwas gesondert implementiert wurde.

im Verhältnis dazu der Overhead. Da die Knoten des Cluster über 32 Threads verfügen, sollte als Breite ein Vielfaches davon gewählt werden, sodass sich die Tasks gleichmäßig auf die Threads verteilen lassen. Bei einer Breite von 128 wurden die besten Ergebnisse gemessen, wobei der Unterschied zu 256 marginal war. Eine gewisse Menge an time steps ist erforderlich, um verlässliche Ergebnisse zu bekommen, da gewisse Randphänomene (z.B. startup-overhead) bei sehr kleinen Szenarios einen großen Einfluss haben können. Die Anzahl an time steps zu erhöhen verlängerte entsprechend auch die Laufzeit einer Konfiguration, und verringerte die Abweichungen zwischen den einzelnen Messläufen. Dadurch konnte die maximale Leistung relativ konstant gemessen werden. Ab etwa 5 time steps wurden stabil über 150 GFLOP/s gemessen. Es wurden weitere Messungen mit mehr time steps gemacht, wobei meist etwa 155 GFLOP/s gemessen wurden. Für die Messungen mit mehr time steps wurde die Task-Größe teilweise etwas gesenkt, da sonst die Laufzeiten der einzelnen Messungen relativ lang gewesen wären.

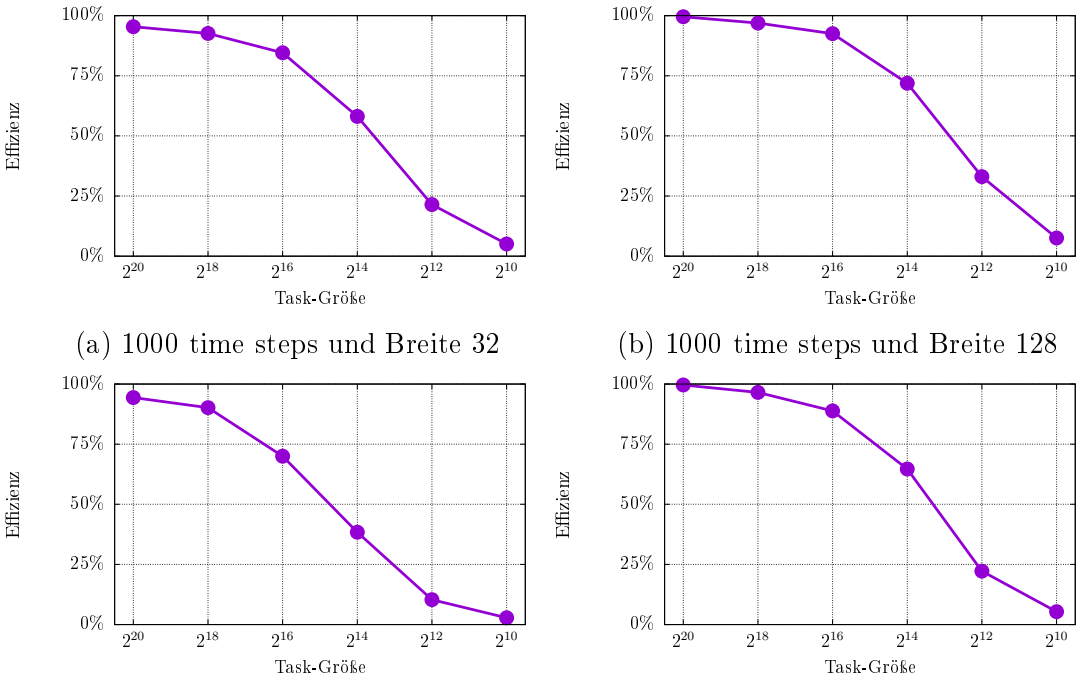
Als Maximalleistung wurden insgesamt knapp 156 GFLOP/s erreicht, was im Folgenden als Maßstab für die Applikations-Effizienz genutzt wird.

5.2. METG

Wie in Abschnitt 2.3 erläutert, wurde im Rahmen des Papers [17] die Metrik METG eingeführt. Diese leitet sich von der Effizienz-Metrik ab und beschreibt, bei welcher Task-Größe noch eine gewisse Effizienz (hier 50%) für ein Szenario erreicht werden kann.

5.2.1. Messung nach Task Bench Paper

Abbildung 5.1a zeigt die gemessenen Effizienz-Werte mit dem Szenario, welches in [17] verwendet wurde (1000 time steps, Breite 32). Es wurde dabei der



(a) 1000 time steps und Breite 32

(b) 1000 time steps und Breite 128

(c) 250 time steps und Breite 32

(d) 250 time steps und Breite 128

Abbildung 5.1.: Effizienzmessungen zu verschiedenen Szenarios

„kritische“ Bereich an Task-Größen gemessen, bei der die Effizienz stark abnimmt. Bei Task-Größen von über 2^{20} liegt die Effizienz bei nahezu 100%, erst unterhalb dieser Marke lässt sich eine merkliche Verschlechterung der Effizienz feststellen. Dies entspricht dem erwarteten Verhalten. Interessant ist besonders die Task-Größe, bei der die Effizienz unter einen bestimmten Wert sinkt. In Paper [17] wird der Schwellwert von 50% betrachtet — also METG(50%) — wie in Abschnitt 2.3.1 definiert. Bei APGAS wird dieser Schwellwert bei Task-Größen unter 2^{14} unterschritten und sinkt danach weiter auf etwa 5% bei der Größe 2^{10}. METG(50%) liegt für APGAS bei diesem Szenario also bei 2^{14}. Die Ergebnisse für METG(50%) bei den verschiedenen Szenarios, die im Rahmen dieser Arbeit gemessen wurden, sind am Ende des nächsten Abschnitts in Tabelle 5.1 aufgelistet.

Im Paper [17] wurde eine Vielzahl von Frameworks betrachtet, bei denen ein breites Spektrum an Werten für METG(50%) gemessen wurde. Das schlechteste

Ergebnis wurde bei Spark festgestellt mit 2^{26}. Chapel und MPI hatten mit 2^9 das insgesamt beste Ergebnis. X10 lag vergleichsweise im unteren Mittelfeld mit 2^{18}.

Es sind jedoch einige Aspekte zu beachten, bevor ein direkter Vergleich zwischen diesen Ergebnissen gezogen werden kann. Bei allen Messungen in Paper [17] wurde für sämtliche Frameworks die gleiche Konfiguration verwendet, die möglicherweise nicht für alle Frameworks gleich gut geeignet ist. In [17] wurde auch erwähnt, dass auf unterschiedlichen Systemen andere Werte für METG(50%) messbar sein können. Da die Messungen in dieser Bachelorarbeit auf einer anderen Hardware ausgeführt wurden als die Messungen von [17], sind auch aus diesem Grund Abweichungen der Ergebnisse möglich.

Um den möglichen Einfluss von Hardwareunterschieden zu überprüfen und um Vergleichswerte zu den Ergebnissen für APGAS messen zu können, wurde das Task Bench Interface für X10 ebenfalls auf dem Cluster der Universität Kassel ausgeführt. Bei Messungen mit der Konfiguration von [17] (1000 time steps, Breite 32) ergaben sich auf dem Cluster ähnliche Werte für die Effizienz, wie bei den Messungen in [3]. Die Laufzeit auf dem Universitäts-Cluster war deutlich länger, was auch zu erwarten war. Die Hardware dieses Clusters ist schließlich nicht so leistungsstark wie die Hardware, die in [17] verwendet wurde. Durch die Betrachtung der Effizienz-Metrik können die Ergebnisse der unterschiedlichen Systeme aber trotzdem leicht miteinander verglichen werden. Es kann also vorläufig geschlossen werden, dass die Unterschiede in der Hardware die Ergebnisse für diese Metrik nicht stark beeinflussen.

Um weitere Einsichten über die Performance von APGAS und X10 zu erlangen, wurde METG(50%) noch für weitere Szenarios ermittelt. Die betrachteten Szenarios benutzten wechselweise 250 time steps statt 1000 und eine Breite von 128 statt 32. Die Messreihen mit APGAS dazu sind in den Abbildungen 5.1b bis 5.1d dargestellt. Bei APGAS wurde generell eine deutlich bessere Performance

als bei X10 gemessen, was sich auch in den Werten von METG(50%) widerspiegelt(vgl. Tabelle 5.1).

Auch wenn bei APGAS die Tendenz für METG(50%) bei diesem Szenario relativ nah bei 2^{14} liegt, sind in den Messreihen deutliche Unterschiede in der Effizienz in dem Bereich um diese Task-Größe herum zu erkennen. Die besten Werte wurden bei 1000 time steps und der Breite 128 gemessen, METG(50%) lag bei 2^{13}. Das schlechteste Ergebnis für METG(50%) war mit 2^{15} bei dem Szenario mit 250 time steps und Breite 32 gegeben. Besonders fällt auf, dass die gemessene Performance zwischen den Szenarios mit gleicher Breite und weniger time steps deutlich schlechter ist.

Wie in Abschnitt 2.2.2 erwähnt, haben die time steps eigentlich keinen Einfluss auf den grundsätzlichen Aufbau des Graphen, sondern machen diesen einfach nur „länger“. Bei den Messungen mit kleineren Task-Größen lagen die Laufzeiten teilweise unter einer Sekunde. Ein möglicher Grund für die gemessenen Unterschiede in der Performance könnte die kurze Laufzeit bei diesen Konfigurationen sein.

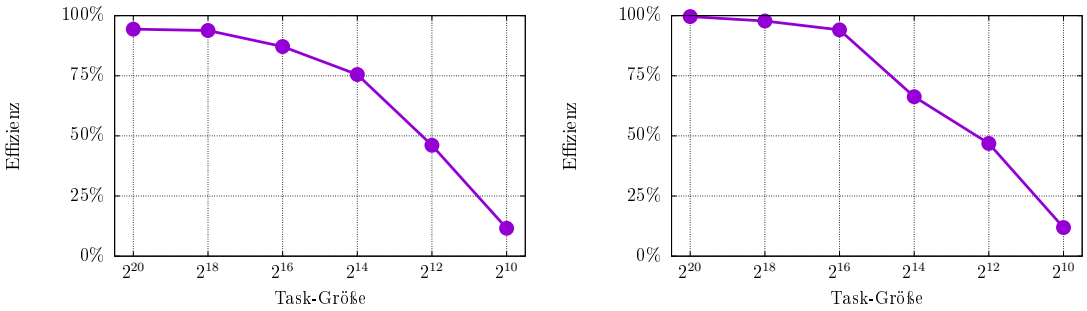
Wenn die Werte für METG tatsächlich von der Laufzeit abhängen, ist es schwierig, verlässliche Aussagen mit dieser Metrik bzgl. der Effizienz in (direkter) Abhängigkeit von der Granularität zu machen. Zwar bezieht sich ein Wert für METG laut Definition nur auf ein konkretes Szenario, doch die Ergebnisse sollten sich zwischen zwei relativ ähnlichen Szenarios nicht stark unterscheiden. Auch wenn die Abweichung in diesem Fall nicht sonderlich groß ist, ist denkbar, dass es bei anderen Systemen und Szenarios zu deutlicheren Unterschieden kommen könnte. Im folgenden Abschnitt wird ein alternativer Messvorgang beschrieben, mit dem ein Szenario so gemessen wird, dass das Ergebnis nicht durch kurze Laufzeiten verfälscht wird.

5.2.2. Messungen mit Anpassung der time steps

Wie in Abschnitt 4.3 erwähnt, führt das APGAS-Interface keinen Warmup-Lauf durch, da dies im Normalfall keine messbaren Auswirkungen auf das Ergebnis hatte. Die Messung von METG zielt aber darauf ab, die Grenzen eines Frameworks auszuloten, indem die Performance bei feiner Granularität der Tasks gemessen wird. Dies hat ebenfalls zur Folge, dass die gesamte Ausführungszeit bei einer Messung relativ kurz wird. Aus diesem Grund ist durchaus denkbar, dass die Messungen in diesem Bereich stark von fixen (bzw. statischen) Overheads abhängen, wozu auch JIT zählt. Auch unabhängig davon ist es möglich, dass verschiedene statische Faktoren die Effizienz bei relativ kurzen Laufzeiten stärker beeinflussen, wodurch das Ergebnis möglicherweise verfälscht wird bzw. sich der Einfluss der Granularität nicht direkt in den Messergebnissen widerspiegelt.

Um dies zu untersuchen, wurde eine Reihe an Messungen durchgeführt, wo die Anzahl an time steps proportional mit der Verringerung der Task-Größe erhöht wurde, sodass die Gesamtmenge an Rechenarbeit bei jeder Konfiguration des Szenarios gleich war. Dieses Vorgehen entspricht nicht dem in [17] beschriebenen Verfahren, wo alle Messungen mit der gleichen Anzahl an time steps durchgeführt wurden.

Bei der Task-Größe 2^{20} wurden 250 time steps verwendet, bei 2^{18} das 4-fache davon, bei 2^{16} das 16-fache, usw. Durch diese Anpassung wird gewährleistet, dass jede Konfiguration eine ausreichend lange Laufzeit hat, um die statischen Overheads zu marginalisieren. Alternativ könnte man für dieses Szenario die time steps aller Konfigurationen so hoch setzen, dass für die kleinste Task-Größe (in diesem Fall 2^{10}) eine ausreichend lange Laufzeit gegeben ist. Das hätte aber zur Folge, dass die Messungen für alle anderen Task-Größen deutlich länger dauern würden, ohne dass dies das Ergebnis verändern würde.



(a) Breite 32 (b) Breite 128

Abbildung 5.2.: Effizienzmessungen mit skalierten time steps

Durch diese Anpassungen soll sichergestellt werden, dass die Ergebnisse tatsächlich die Effizienz in Abhängigkeit von der Granularität der Tasks darstellen, was METG auch ausdrücken soll. Dies ließe sich prinzipiell auch durch eine Veränderung der Breite erreichen, doch dies verändert den Aufbau des Graphen etwas grundlegender, während die Anzahl an time steps den Taskgraphen praktisch nur „verlängert“.

In Abbildung 5.2 sind die Ergebnisse dieser Messreihen dargestellt. Hier lässt sich deutlich erkennen, dass die beiden Messreihen eine sehr ähnliche Tendenz aufweisen, in beiden Fällen liegt METG(50%) bei 2^{13}, bei 2^{12} ist die Effizienz nur knapp unter 50%. Dieses Experiment legt nahe, dass die Anpassung der Messreihen ermöglicht, Schwankungen in den Ergebnissen für METG(50%) zu reduzieren. Dabei ist aber zu beachten, dass diese Szenarios auf Basis der gemessenen Schwächen des APGAS-Interfaces konzipiert wurden und sich von dem Szenario aus [17] unterscheiden. Bei den Messungen für X10 hingegen hatte die Anpassung der time steps fast keine Auswirkung auf METG(50%). Die Effizienz war bei den Konfigurationen mit mehr time steps sogar marginal schlechter. Es wäre eigentlich zu erwarten, dass sich die Performance entweder verbessert oder nahezu gleich bleibt. Dass X10 mit mehr time steps eine schlechtere Effizienz aufweist, ist überraschend.

Breite / time steps(ts)	1000 ts	250 ts	angepasste ts
Breite 32 (APGAS)	2^{14}	2^{15}	2^{13}
Breite 128 (APGAS)	2^{13}	2^{14}	2^{13}
Breite 32 (X10)	2^{18}	2^{18}	2^{18}
Breite 128 (X10)	2^{15}	2^{15}	2^{15}

Tabelle 5.1.: Resultate für verschiedene METG-Messungen

Die Anpassung der time steps hilft also im Fall von APGAS dabei, die Ergebnisse für METG(50%) zwischen den beiden betrachteten Szenarios zu vereinheitlichen. Diese Herangehensweise scheint diesen Effekt aber nicht im Allgemeinen zu erzielen (oder zumindest nicht bei X10). Möglicherweise zeigt dies aber auch ein Problem mit der Implementierung des X10-Interfaces auf, welches sich stark von der Implementierung des APGAS-Interfaces unterscheidet, obwohl beide Frameworks auf dem APGAS-Modell basieren (wie in Abschnitt 2.1 erwähnt). Das X10-Interface benutzt für jeden Prozessorkern einen eigenen Place, was ein Grund für die deutlich schlechtere Performance bei der geringeren Breite sein könnte. Eine genauere Betrachtung dieses Problems liegt aber außerhalb des Themenbereichs dieser Arbeit. Die große Differenz bei den Ergebnissen für X10 zeigt aber beispielhaft auf, dass ein einzelnes Szenario nicht die Gesamtperformance eines Frameworks zusammenfasst.

Nichtsdestotrotz konnte für APGAS bei diesem Szenario, im Vergleich zu den anderen Frameworks, die in [17] betrachtet wurden, ein guter Wert für METG(50%) gemessen werden. Von den insgesamt 18 gebenchmarkten Frameworks waren nur MPI, MPI+OpenMP, Charm++ und Chapel deutlich besser. PaRSEC[2], Realm, OpenMP Tasks und OmpSs hatten eine sehr ähnliche Performance wie APGAS.

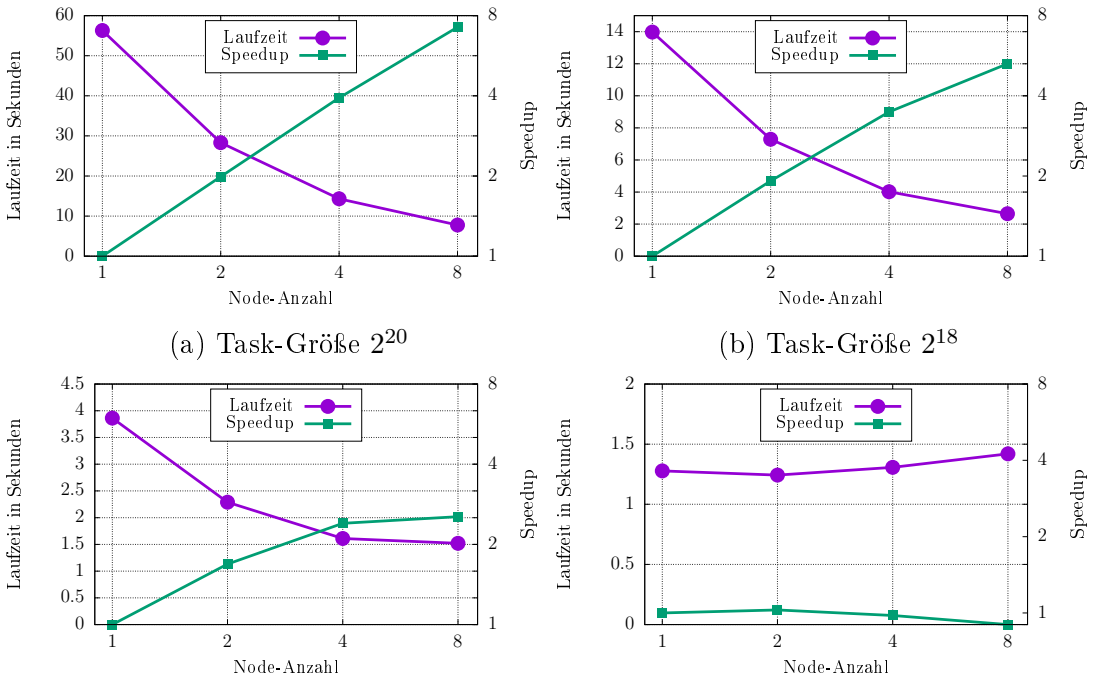
[2]PaRSEC DTD, PaRSEC PTG und PaRSEC shard

5.3. Strong-Scaling

Wie bereits in Abschnitt 2.3.2 erwähnt, wurden in [17] einige Messungen zu strong-scaling mit dem MPI-Interface durchgeführt. Zu diesen Messungen sind in [3] leider keine Messergebnisse veröffentlicht worden. Ein Vergleich ist daher nur sehr begrenzt möglich. Es ist trotzdem sinnvoll Messungen zum strong-scaling mit dem APGAS-Interface durchzuführen, da es sich um eine häufig benutzte Metrik handelt.

Bei der Erstellung von Strong-Scaling-Szenarios ist bei Task Bench (aber auch bei realen Anwendungen) zu beachten, dass eine Anwendung nur begrenzt parallelisierbar ist. Wie in Abschnitt 2.2.2 erwähnt ist dies bei Task Bench durch die Breite des Taskgraphen begrenzt. Um eine vernünftige Messung des strong-scalings durchführen zu können, muss also eine Konfiguration mit ausreichender Breite gewählt werden. Die Breite muss bei jeder der betrachteten Node-Anzahlen mindestens so groß wie die Anzahl an CPU-Kernen (bzw. Threads) sein. Für die Messungen wurden 250 time steps und eine Breite von 256 gewählt. Diese Konfiguration wurde auf bis zu 8 Nodes ausgeführt. Bei 8 Nodes ist so noch eine Breite von 32 pro Node gegeben, was der Anzahl an CPU-Kernen entspricht. Analog zu den Messungen in [17] wurde die Anzahl an Nodes exponentiell skaliert (1, 2, 4 und 8 Nodes).

In Abbildung 5.3a ist die Messreihe mit der Task-Größe 2^{20} dargestellt. Hier lässt sich eine nahezu ideale Skalierbarkeit erkennen, die x-Achse sowie die rechte y-Achse sind logarithmisch skaliert, sodass eine gerade Linie des Speedups von Punkt (1,1) zu (8,8) einer perfekten Skalierbarkeit entsprechen würde. Bei jeder Verdopplung der Node-Anzahl halbiert sich die Laufzeit fast, bei 8 Nodes liegt der Speedup bei 7,24. Bei der Task-Größe 2^{18} (Abbildung 5.3b) sind die Ergebnisse bei der Skalierung zwar etwas schlechter, aber es ist immer noch ein deutlicher Speedup gegeben, bei 8 Nodes liegt dieser bei ca. 5,27.



(a) Task-Größe 2^{20}

(b) Task-Größe 2^{18}

(c) Task-Größe 2^{16}

(d) Task-Größe 2^{14}

Abbildung 5.3.: Strong Scaling mit 250 time steps und Breite 256

Die Performance bei der Skalierung ist in Abbildung 5.3c deutlich schlechter, insgesamt ist der Speedup bei jeder Node-Anzahl deutlich unter den Idealwerten, der Unterschied zwischen den Laufzeiten bei 4 und 8 Nodes ist nur noch sehr gering. In Abbildung 5.3d musste die rechte y-Achse etwas verschoben werden, da der Speedup ab 4 Nodes hier sogar unter 1 liegt. Die Laufzeit des Benchmarks war mit mehreren Nodes also teilweise sogar schlechter als bei der Ausführung mit einem Node.

Diese Messungen zeigen relativ deutlich eine Grenze der Skalierbarkeit bei diesen Szenarios auf. Es ist nicht auszuschließen, dass statische Overheads — wie auch bei den Messungen zu METG — bei den relativ kurzen Laufzeit einen stärkeren Einfluss auf die Ergebnisse hatten. Ggf. sollten diese Messungen mit einem Warmup-Lauf wiederholt werden, wie in Abschnitt 4.3 beschrieben.

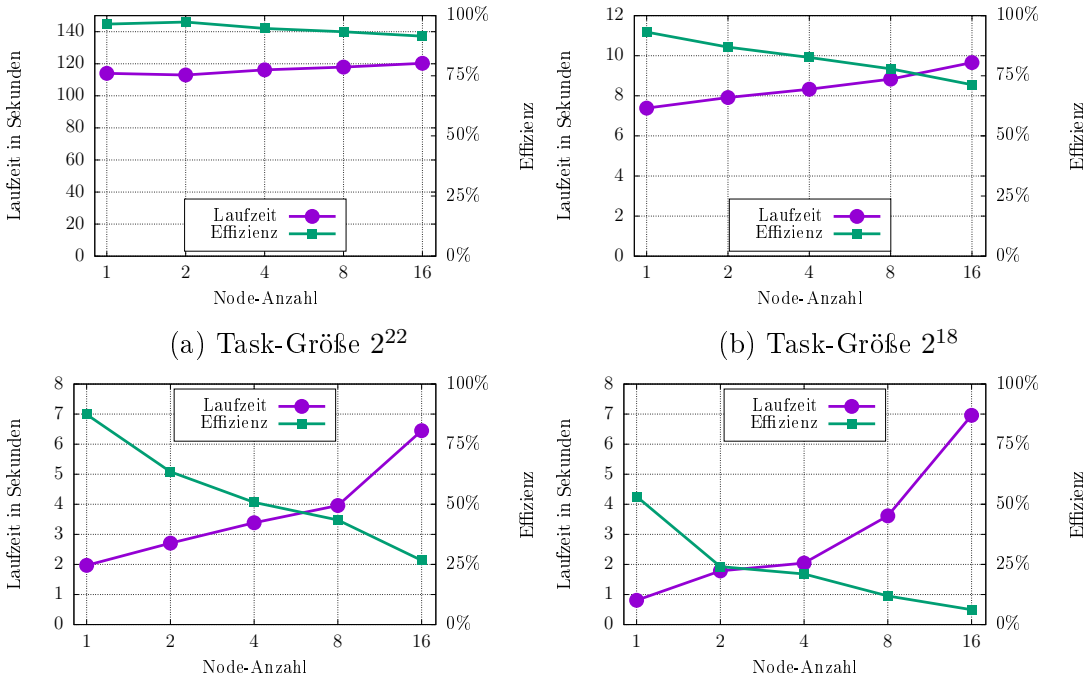
In der graphischen Darstellung der Messungen in [17][3] ist für MPI eine etwas bessere Skalierbarkeit zu erkennen. Dort ist bei Task-Größen von bis zu 2^{14} noch eine nahezu idealer Speedup bei der Skalierung von 1 auf 8 Nodes gegeben. Ab der Task-Größe 2^{12} flacht der Speedup deutlich ab, ähnlich zu den Messungen mit APGAS bei der Größe 2^{16}. Ähnlich zu den Messungen zu METG(50%) scheint der Unterschied zwischen den beiden Frameworks auch hier in der Größenordnung von 2^4 zu liegen.

Aufgrund der mangelhaften Datenlage ist ein genauerer Vergleich leider nicht möglich. Es wurden für andere Frameworks anscheinend keine Messungen durchgeführt, oder zumindest nicht in [17] erwähnt. Aufgrund der relativ geringen Differenz zu MPI (das in fast allen Messungen in [17] die beste Performance hatte), lässt sich aber vorsichtig vermuten, dass APGAS im Vergleich zu anderen Frameworks auch in diesem Bereich relativ gut abschneiden würde.

5.4. Weak-Scaling

Die Messungen zum weak-scaling wurden mit 1000 time steps und einer Breite von 32 pro Node durchgeführt. Hier gibt es in [17] auch einige Messungen zu anderen Frameworks. Diese Ergebnisse sind in dem Paper aber etwas anders dargestellt. Anstatt die Laufzeiten und die Effizienz bei den verschiedenen Szenarios zu betrachten wurde METG(50%) für alle Frameworks bei den verschiedenen Node-Anzahlen gemessen. Da sich METG von der Effizienz ableitet (siehe dazu Abschnitt 2.3.1), sollten so die „Overheads in einer Zahl zusammengefasst werden“ [17]. Dabei ist METG(50%) als Laufzeit eines einzelnen Tasks angegeben, anstatt als Anzahl an Iterationen. Da die Laufzeit eines Tasks stark von der benutzten Hardware abhängt, ist ein Vergleich mit den Messungen dieser Arbeit

[3] Es ist leider nicht angegeben, welches Szenario dafür genutzt wurden. Es ist davon auszugehen, dass es sich von dem hier gewählten unterscheidet.



(a) Task-Größe 2^{22}

(b) Task-Größe 2^{18}

(c) Task-Größe 2^{16}

(d) Task-Größe 2^{14}

Abbildung 5.4.: Weak-Scaling mit 1000 time steps und Breite 32 pro Node

mittels der Grafik in [17] also nicht möglich. In [3] sind die genauen Messwerte angegeben, anhand dieser Werte können die Messungen verglichen werden.

Die Abbildung 5.4 zeigt zum einen die gemessene Laufzeit, zum anderen die gemessene Effizienz bei den einzelnen Konfigurationen an. Wie in Abschnitt 2.3.2 erläutert bleibt die Laufzeit (und Effizienz) in einem Szenario gleich, wenn eine optimale Skalierbarkeit bei dem Szenario gegeben ist. Da es hier nicht möglich ist, einen Speedup darzustellen, wird stattdessen zusätzlich die Effizienz betrachtet. Dabei ist anzumerken, dass Skalierbarkeit und Effizienz zwei verschiedene Eigenschaften ausdrücken. Die Effizienz ist ein Maß für die Performance (wie in Abschnitt 2.3 definiert) und bezieht sich auf eine einzelne Konfiguration. Die Skalierbarkeit beschreibt, wie gut die Performance bei der Skalierung in einem Szenario beibehalten wird. Bei einem Szenario kann die Skalierbarkeit also auch

dann gut sein, wenn die Effizienz bei jeder Konfiguration schlecht ist, solange sie nicht innerhalb des Szenarios sinkt.

In Abbildung 5.4a ist eine nahezu ideale Skalierbarkeit bei der Task-Größe 2^{22} erkennbar. Die Laufzeit steigt nur leicht an und die Effizienz bleibt durchgehend über 90%. Bei den Task-Größen 2^{18} bis 2^{14} ist gut zu sehen, wie die Skalierbarkeit in diesem Bereich abnimmt. In Abbildung 5.4b zu Task-Größe 2^{18} sinkt die Effizienz auf etwa 70% bei 16 Nodes ab, was noch einigermaßen akzeptabel ist. Bei der Task-Größe 2^{16} sinkt die Effizienz von knapp 90% auf ca. 26% (dargestellt in Abbildung 5.4c), was hier deutlich ein Limit der Skalierbarkeit aufzeigt.

In Abbildung 5.4d zu Task-Größe 2^{14} liegt die Effizienz bereits bei einem Node nur bei 50%, und sinkt auf etwa 6% bei 16 Nodes. Besonders fällt hier ein Sprung zwischen einem und zwei Nodes auf, wo sich die Laufzeit verdoppelt und die Effizienz dementsprechend halbiert. Ein ähnlicher Sprung wurde auch bei einigen anderen Frameworks gemessen (siehe [17]). Besonders auffällig ist dabei Chapel, wo die Effizienz bei der Größe 2^{13} von ca. 90% auf etwa 50% sinkt. Auch bei geringeren Task-Größen weist Chapel ungefähr eine Halbierung der Effizienz beim weak-scaling von einem auf zwei Nodes auf, was bei den weiteren Messungen mit APGAS zur Task-Größe 2^{10} (siehe Anhang) ebenfalls der Fall ist. Bei anderen Frameworks ist dieses Verhalten weniger stark ausgeprägt, bei einigen ist es gar nicht festzustellen. Daraus kann man für APGAS schließen, dass es sich bei bestimmten Szenarios nicht unbedingt lohnt, zwei Knoten zu benutzen, da die Gesamtperformance dadurch kaum besser wird.

Die Messungen zeigen, dass APGAS sich beim weak-scaling relativ gut verhält und sich auch bei feinerer Granularität noch einigermaßen effizient skalieren lässt. Insgesamt schneidet APGAS ähnlich gut ab wie bei den Messungen zu METG in Abschnitt 5.2. Im Vergleich zu X10 hat APGAS bei fast jeder Konfiguration eine deutlich bessere Effizienz, X10 behält die Effizienz aber bei

der Skalierung besser bei. Generell fällt bei den Ergebnissen von [17] auf, dass einige Frameworks wie z.B. Regent kaum Effizienz bei der Skalierung verlieren. Dies ist aber vor allem bei Frameworks der Fall, die ohnehin eine schlechte Performance bei den jeweiligen Task-Größen haben. Es ist daher naheliegend, dass bei diesen Frameworks Overheads bereits bei einem Knoten einen so starken Einfluss haben, dass die Overheads, die durch Verteilung auf mehrere Knoten entstehen, die Effizienz im Vergleich dazu weniger beeinträchtigen. Bei APGAS und bei performanteren Frameworks ist dies anscheinend nicht der Fall. Zwar ist es aus Sicht der Skalierbarkeit gut, wenn sich ein Problem ohne großen Effizienzverlust skalieren lässt, doch wenn die Effizienz im Vergleich zu anderen Frameworks generell deutlich schlechter ist, ist dies in der Praxis auch nicht von Vorteil.

6. Fazit

Im Rahmen dieser Arbeit wurde ein Interface für APGAS entwickelt, mit dem erfolgreich Messungen mit dem *stencil* Benchmark des Benchmarking-Systems Task Bench ausgeführt werden konnten. Durch diesen Benchmark konnte die Ausführung verschiedener Anwendungsszenarios simuliert und gemessen werden, wodurch Erkenntnisse über die Skalierbarkeit und die Grenzen von APGAS gewonnen werden konnten.

Bei der Implementierung des Interface kam besonders zum Vorschein, dass die Benutzung von Java-Konstrukten wichtige Möglichkeiten bietet, die die Konzepte des APGAS-Modells bereichern und die Entwicklung vereinfachen.

Außerdem wurde die Metrik METG vorgestellt und Probleme bezüglich der Vergleichbarkeit von Frameworks mit dieser Metrik aufgezeigt. Beim Vergleich zwischen einer Vielzahl von Frameworks, können die Eigenarten dieser Frameworks nur dann fair in den Vergleich einbezogen werden, wenn eine entsprechend vielfältige Auswahl an Szenarios betrachtet wird. Task Bench ermöglicht es, durch die Abstraktion als Taskgraph, Benchmarks auf eine intuitiv verständliche Weise konfigurierbar zu machen. Um die Konfigurationsoptionen aber gut nutzen zu können, müssen die Eigenarten eines Frameworks festgestellt und berücksichtigt werden, um Messergebnisse adäquat bewerten zu können.

Es bleiben noch einige Verbesserungen für das Interface offen: die Unterstützung weiterer Graphtypen, die Durchführung eines Warmup-Laufs, die gleichzeitige Ausführung mehrerer Graphen, sowie die verschiedenen Kerneltypen bieten noch Möglichkeiten für die Weiterentwicklung des Interfaces und weitere Messungen mit APGAS und Task Bench.

Literaturverzeichnis

[1] APGAS-Library. `https://github.com/posnerj/PLM-APGAS`.

[2] Task Bench source code. `https://github.com/StanfordLegion/task-bench`.

[3] Task Bench results. `https://github.com/StanfordLegion/task-bench-results`.

[4] Swig interface compiler. `http://www.swig.org/`.

[5] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Design patterns percolating to parallel programming framework implementation. *International Journal of Parallel Programming*, 42(6):1012—1031, 2014.

[6] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, page 483—485, 1967.

[7] Tatjana Davidović and Teodor Gabriel Crainic. Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems.

[8] John L. Gustafson. Reevaluating Ahmdal's Law. *Communications of the ACM*, 31(5):532—533, 1988.

[9] IBM. The X10 parallel programming language. `http://x10-lang.org/`.

[10] Hazelcast Inc. `https://hazelcast.com/`.

[11] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed*

Computing, 59:381—422, 1999.

[12] Simon McIntosh-Smith, Matthew Martineau, Tom Deakin, Grzegorz Pawelczak, Wayne Gaudin, Paul Garrett, Wei Liu, Richard Smedley-Stevenson, and David Beckingsale. TeaLeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, page 842—849, 2017.

[13] Simon McIntosh-Smith, James Price, Tom Deakin, and Andrei Poenaru. A performance analysis of the first generation of HPC-optimized Arm processors.

[14] S. J. Pennycook, J. D. Sewall, and V. W. Lee. A metric for performance portability.

[15] Jonas Posner and Claudia Fohry. Transparent resource elasticity for task-based cluster environments with work stealing. In *50th International Conference on Parallel Processing Workshop*, page 1—10, 2021.

[16] Jonas Posner, Lukas Reitz, and Claudia Fohry. Comparison of the HPC and Big Data Java Libraries Spark, PCJ and APGAS. In *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, page 11—22, 2018.

[17] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Leek, Sean Treichler, Patrick McCormick, and Alex Aiken. Task Bench: A parameterized benchmark for evaluating parallel runtime performance.

[18] Olivier Tardieu. The APGAS Library: Resilient parallel and distributed programming in Java 8.

A. Anhang

Auf der beigefügten CD befinden sich die verwendeten Messdaten, sowie der Code des APGAS-Interfaces. In dem GitLab Repository `https://code.plm.eecs.uni-kassel.de/jposner/taskbenchapgas` wurde dies ebenfalls hochgeladen. Details zur Ausführung des Interfaces sind in einer Readme-Datei dokumentiert.