UNIKASSEL
VERSITÄT

**Universität Kassel**

# Erkennung und Behandlung flüchtiger Fehler in Taskpools am Beispiel GLB

**Bachelorarbeit**

Vorgelegt im

Fachbereich 16 Elektrotechnik/Informatik

Fachgebiet Programmiersprachen und –methodik

Matthias Beer

31202505

Erstgutachter : Prof. Dr. Claudia Fohry

Zweitgutachter : Prof. Dr. Gerd Stumme

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt wurde. Insbesondere versichere ich, dass alle wörtlichen und sinngemäßen Übernahmen aus fremden Quellen als solche gekennzeichnet sind.

Matthias Beer

Hann. Münden, den 25.10.2015

# Inhaltsverzeichnis

# 1 Einleitung

Um die Geschwindigkeit von Programmen zu erhöhen, wird immer häufiger parallele Programmierung eingesetzt. Das Prinzip dabei ist es, ein Problem in kleinere Teilprobleme aufzuteilen und diese auf den verschiedenen Kernen eines Prozessors bzw. den Knoten eines Clusters gleichzeitig zu berechnen. Verschiedene Programmiersprachen verwenden dabei unterschiedliche Herangehensweisen. In *Shared Memory* Sprachen wie OpenMP greifen alle Prozesse auf denselben Adressraum zu, was einen hohen Synchronisationsaufwand für gemeinsam benutzte Variablen bedeutet. In Programmiersprachen wie MPI ist der Adressraum getrennt und die Prozesse senden sich Daten über Nachrichten (Message Passing). Eine weitere Variante ist das PGAS Programmiermodell, auf welchem unter anderem die Programmiersprache X10 beruht. Dabei ist der Adressraum aufgeteilt, sodass jeder Prozess über eigenen lokalen Speicher verfügt, aber auch auf den anderer Prozesse zugreifen kann. X10 verfügt außerdem über das GLB (Global Load Balancing) Framework, welches die Arbeit eines parallelen Programms gleichmäßig auf die verschiedenen Knoten verteilt.

Ein Problem der parallelen Programmierung ist die erhöhte Wahrscheinlichkeit von Hardwarefehlern. Bei diesen kann man zwischen permanenten und flüchtigen Fehlern unterscheiden.

Permanente Fehler (Hard Errors) bezeichnen einen während der Laufzeit auftretenden Ausfall der Hardware, z.B. ein ausfallender Knoten im Cluster. Ohne Fehlerbehandlung hat ein solcher Ausfall den Abbruch des Programms zur Folge, ohne ein verwertbares Ergebnis zu liefern. Es gibt jedoch viele veröffentlichte Arbeiten, die sich mit der Behandlung von permanenten Fehlern beschäftigen.

Flüchtige Fehler (Soft Errors) treten meistens in Form von *Silent Data Corruption* auf. Sie stellen ein besonderes Problem dar, da sie im Gegensatz zu permanenten Fehlern unerkannt bleiben können. Das Programm terminiert dann mit einem verfälschten Ergebnis, welches nur bei mehrfacher Ausführung des Programms auffallen würde. Die Zunahme flüchtiger Fehler hängt mit den kleiner werdenden Chips und der angelegten Spannung zusammen [1].

Aufgrund dieser Entwicklung nimmt man an, dass in absehbarer Zukunft von einem Fehler pro Tag auszugehen ist.

In der vorliegenden Bachelorarbeit wurde eine in Java implementierte Variante von GLB erweitert, um in diese die Fehlertoleranz gegenüber flüchtigen Fehlern zu integrieren. Dies wurde realisiert, indem die auf einem Place durchgeführten Berechnungen von einem weiteren Place überprüft werden. Die Änderungen an den vorgegebenen GLB-Konstrukten wurden dabei minimal gehalten, um die Funktionalität des Frameworks nicht einzuschränken. Außerdem wurde die Kommunikation der einzelnen Prozesse gering gehalten um die Performance nicht zu stark zu beinträchtigen. Zusätzlich wurde die Implementierung so umgesetzt, dass die erweiterte GLB-Variante für beliebige Nutzerprogramme verwendet werden kann.
Das Konzept der Fehlerbehandlung orientiert sich an der Funktionsweise von ACR, welche später genauer erläutert wird. Im Rahmen der Arbeit wurden dabei zwei Varianten implementiert. In der ersten Variante wird vor einem Arbeitsschritt jeweils die gesamte Arbeitsmenge eines Places an einen zweiten Place gesendet und auch von diesem ausgeführt, um die berechneten Ergebnisse miteinander vergleichen zu können. Um die Performance zu verbessern, wurde in der zweiten Variante die Kommunikation zwischen den Places verringert, indem nur Teile der Arbeitsmenge an bestimmten Stellen an weitere Places gesendet werden. Zeitmessungen zeigen, dass beide implementierten Varianten deutlich langsamer als die ursprüngliche GLB-Variante sind, was sich durch den höheren Kommunikationsaufwand zwischen den Places erklären lässt. Um die Zeitmessungen durchzuführen wurde als Beispielprogramm eine bereits implementierte Variante des *UTS-Benchmarks* verwendet, welches später beschrieben wird.
In Kapitel 2 werden die Programmiersprache X10, das Programmiermodell APGAS sowie GLB vorgestellt. Danach werden in Kapitel 3 verwandte Arbeiten wie ACR beschrieben. In Kapitel 4 wird der eigene Ansatz zur Fehlerbehandlung und in Kapitel 5 dessen genaue Implementierung erklärt. Kapitel 6 diskutiert die Performance der implementierten Varianten im Vergleich zur ursprünglichen GLB-Variante. Kapitel 7 beinhaltet eine kurze Zusammenfassung.

# 2 Grundlagen

In diesem Kapitel wird zunächst die Programmiersprache X10 sowie das APGAS-Programmiermodell und das APGAS-Framework erklärt. Anschließend wird das GLB-Framework vorgestellt und die in GLB verwendeten Konstrukte erläutert. Danach wird der genaue Ablauf des GLB Algorithmus beschrieben.

## 2.1 X10 und APGAS

X10 [2] ist eine objektorientierte Programmiersprache, die 2004 von IBM speziell für die parallele Programmierung entwickelt wurde. X10 folgt dem *Partitioned Global Adress Space* (PGAS) Modell, welches zum APGAS Modell [3] erweitert wurde.
APGAS steht für „Asynchronous Partitioned Global Address Space“ und erweitert somit das ältere PGAS Modell um Asynchronität. Im Gegensatz zu *shared memory* Sprachen wie OpenMP ist beim APGAS Modell der globale Adressraum in Bereiche aufgeteilt. Diese Bereiche und die darauf laufenden Prozesse werden Places genannt. Ein Place kann auch auf den Speicherbereich eines anderen Places zugreifen. Der Zugriff auf fremden Speicher ist allerdings zeitaufwendiger als der Zugriff auf den lokalen Speicher. Ein Prozess, den man auf einem X10 Place startet, wird *Activity* genannt. Zur Laufzeit können dynamisch neue Activities erzeugt und ausgeführt werden, was die Programmierung mit X10 sehr flexibel macht.
Die Verteilung von Daten und Berechnungen in X10 wird vom Programmierer bestimmt. Um auf einen anderen Place zugreifen zu können wird das Schlüsselwort *at* verwendet. Mit *async* wird eine asynchrone Activity auf dem aktuellen Place gestartet. Die beiden Schlüsselwörter können auch kombiniert werden, um einen asynchronen Prozess auf einem beliebigen Place zu starten. Ein weiteres Konstrukt ist *finish*. Dieses dient als Barriere für asynchrone Aufrufe, indem es garantiert, dass sämtliche innerhalb des finish-Konstrukts gestarteten asynchronen Activities vor dem Verlassen des Konstrukts beendet wurden.

Im Juni 2015 wurde die *APGAS-library* für Java 8 veröffentlicht [4]. Diese wurde entwickelt, um die in X10 bzw. Resilient X10 verwendeten Konstrukte auch für Java verfügbar zu machen. Für den Einsatz von *at*, *async* und *finish* werden dabei die in Java 8 eingeführten Lambda-Expressions verwendet. Das APGAS-Framework wird in der Bachelorarbeit verwendet um die Behandlung flüchtiger Fehler zu realisieren.

## 2.2 GLB

### 2.2.1 Allgemein

Das Global Load Balancing Framework GLB [5] basiert auf dem lifeline graph work-stealing Algorithmus [6] und wurde für X10 entwickelt. Das für diese Arbeit verwendete GLB-Framework wurde mithilfe der oben genannten APGAS-library in Java implementiert [7]. GLB wird dazu verwendet, um die Teilaufgaben eines parallelen Programms, welche in GLB als Tasks bezeichnet werden, effizient auf die existierenden Places zu verteilen. Auf jedem Place arbeitet ein Worker, welcher über einen Taskpool verfügt. Der Worker bearbeitet alle Tasks aus seinem Taskpool. Bei der Bearbeitung eines Tasks, können auch neue Tasks erzeugt werden. Um die Lastenverteilung zu verbessern, können Worker, deren Taskpool bereits leer ist, Stehlanfragen an andere Places senden, um einen Teil der noch nicht bearbeiteten Tasks zu erhalten. Dies ist besonders effektiv bei Algorithmen, bei denen während der Ausführung dynamisch weitere Tasks erzeugt werden, da der Rechenaufwand der Places sonst sehr ungleichmäßig verteilt wäre. GLB lässt sich allerdings auch bei Algorithmen verwenden, bei denen die Menge der Tasks bereits zu Beginn statisch feststeht. Die Anzahl der durchgeführten Stehlanfragen ist zwar nicht so hoch wie bei den dynamischen Algorithmen, dennoch kann auch hier eine Performance Verbesserung festgestellt werden.
GLB ist jedoch nicht auf alle Probleme anwendbar. Um GLB für ein Programm verwenden zu können, müssen einige Voraussetzungen erfüllt sein. Zunächst muss man das Hauptproblem in ausführbare bzw. berechenbare Teilprobleme (Tasks) aufteilen können. Die einzelnen Tasks des Programms müssen mit den lokal zur Verfügung stehenden Informationen

berechnet werden können, es darf also während der Berechnung eines Tasks keine Kommunikation zwischen den Places stattfinden. Die Tasks dürfen des Weiteren keine Seiteneffekte verursachen und müssen auf jedem beliebigen Place ausführbar sein. Außerdem müssen die Ergebnisse der einzelnen Tasks reduzierbar sein, d.h. das Gesamtergebnis des Programms muss sich aus den Teilergebnissen der einzelnen Tasks berechnen lassen. Der Reduzierungsoperator, der vom Nutzer implementiert wird, muss kommutativ und assoziativ sein, damit das Endergebnis auch dann dasselbe ist, wenn die Tasks in anderer Reihenfolge bearbeitet werden.

### 2.2.2 Konstrukte

In diesem Abschnitt werden die Interfaces und Methoden von GLB aufgeführt, welche vom Nutzer selbst implementiert werden müssen, um GLB verwenden zu können.

**TaskBag:**

Nutzer von GLB müssen eine eigene Klasse erstellen, die das GLB Interface *TaskBag*, sowie die Methoden `split()` und `merge()` implementiert, welche von der Klasse *TaskQueue* (s. unten) benötigt werden. Tasks die mit `split()` von der TaskQueue getrennt werden, werden in einen TaskBag eingefügt. Mit `merge()` werden die Tasks eines TaskBag in eine TaskQueue eingefügt.

**TaskQueue:**

Ähnlich wie TaskBag ist auch TaskQueue ein Interface welches vom Nutzer programmspezifisch implementiert werden muss. Für die TaskQueue werden folgende Methoden benötigt:

- `process(n: Long) : boolean`
  Diese Methode versucht bis zu `n` Tasks aus der TaskQueue zu berechnen. Dabei werden berechnete Tasks aus der TaskQueue entfernt und neu erzeugte an die TaskQueue angehangen. Befinden sich nach der Berechnung noch Tasks in der TaskQueue, liefert die Methode *true* zurück, andernfalls *false*.

- `split() : TaskBag`
  Fügt die Hälfte der Tasks der TaskQueue in einen TaskBag ein und gibt diesen zurück. Gibt „Null“ zurück, wenn die TaskQueue zu wenige Tasks besitzt, um sie zu halbieren.

- `merge(TaskBag)`
  Fügt die Tasks des übergebenen TaskBag in die eigene TaskQueue ein.

- `getResult()`
  Gibt das aus den berechneten Tasks einer TaskQueue durch `reduce()`-Aufrufe reduzierte Ergebnis zurück.

- `reduce()`
  Fasst die Ergebnisse der Tasks der Queue zusammen. Werden die Ergebnisse der Worker zusammengefasst, erhält man das Gesamtergebnis.

### 2.2.3 Ablauf

Zu Beginn des Programms werden die Start-Tasks auf die Places verteilt. Alternativ erhält zunächst nur ein Place die Start-Tasks. Als nächstes beginnen die Worker mit der Arbeit. Jeder Worker ruft in einer Schleife `process(n)` auf, um damit bis zu n Tasks aus seiner

TaskQueue zu bearbeiten. Ist ein Task berechnet, wird das Teilergebnis mit dem Reduzierungsoperator zu einer Result-Struktur der TaskQueue hinzugefügt. Zwischen den einzelnen `process(n)` Aufrufen reagieren die Worker auf die Stehlanfragen anderer Worker. Dabei wird mithilfe der `split()` Methode jeweils die Hälfte der Tasks aus der eigenen Queue in einen TaskBag eingefügt, welcher an die sogenannten *thieves* oder an *lifelinethieves* gesendet wird. Ein thief ist ein Worker der einen Stehlanfrage gesendet hat. Lifelinethieves sind diejenigen thieves, die zusätzlich zur lifeline des Workers gehören. Lifelines werden bei der Initialisierung von GLB mit dem lifeline graph work-stealing Algorithmus [6] für jeden Place berechnet. Hat ein Worker alle Tasks seiner TaskQueue bearbeitet, sendet er eine Stehlanfrage an einen anderen Worker und wartet solange, bis er eine positive oder negative Antwort bekommt. Bei einer Stehlanfrage werden zwei Phasen ausgeführt. Zunächst wird bei *w* zufälligen Places angefragt. Danach wird die Anfrage an die lifelines gesendet. Erhält der Worker einen TaskBag von einem anderen, werden dessen Tasks mit der `merge()`-Methode an die eigene TaskQueue angefügt und er beginnt wieder damit, `process(n)` aufzurufen. Werden alle Anfragen abgelehnt, stellt der Worker seine Aktivität ein. Ein nicht mehr arbeitender Worker kann von einem anderen wieder aktiviert werden, indem dieser seine Tasks mit den lifelinethieves teilt.
Wenn kein Worker mehr arbeitet, fasst GLB die Teilergebnisse der Worker, welche sich in den Result-Strukturen ihrer jeweiligen TaskQueues befinden, mithilfe der vom Nutzer implementierten `reduce()` Methode zusammen, um das Endergebnis zu erhalten.

## 2.3 Beispielprogramm UTS

Um die in der Bachelorarbeit erstellte Erweiterung von GLB zu testen, wurde als Beispielprogramm eine bereits implementierte Variante [7] des UTS-Benchmarks [8] verwendet. Beim UTS (*unbalanced tree search*) Problem geht es darum, die Knoten eines ungleichmäßigen Suchbaumes zu zählen. Die Eigenschaften des Baumes lassen sich durch

Parameter wie Tiefe oder Verzweigungsgrad beeinflussen. Zur Erzeugung der Kindknoten des Suchbaums wird der SHA1-Algorithmus verwendet. Der Baum wird dabei während der Laufzeit dynamisch erstellt. Das bedeutet, dass die Knotenzahl der Teilbäume zu Beginn des Programms nicht bekannt ist. Eine Verteilung der Tasks ohne Lastenverteilung wie von GLB würde zu einer ungleichmäßigen Auslastung der Places führen. Aus diesem Grund eignet sich UTS besonders dazu, die verschiedenen Schemata zur Lastenverteilung zu testen.
Im vorliegenden UTS Beispiel erhält zunächst nur ein Place die Start-Tasks. Die Verteilung auf die übrigen Places findet über work-stealing statt. Ein Task steht in dieser Variante für einen Knoten des Suchbaums. Mit dem Aufruf von `process(n)` werden bis zu `n` Tasks berechnet, was in diesem Fall bedeutet, dass die jeweiligen Kinder des Baumknotens berechnet und anschließend als neue Tasks an die TaskQueue angehangen werden. Die TaskQueue enthält außerdem eine Result-Struktur, in welcher die Anzahl der bisher erzeugten Knoten gespeichert wird. Haben alle Worker ihre Arbeit beendet, werden die Ergebnisse der TaskQueues zusammengezählt, was die Gesamtanzahl der Knoten im Suchbaum ergibt.

# 3 Verwandte Arbeiten

Es existieren Arbeiten, die sich mit der Fehlertoleranz gegenüber permanenten und flüchtigen Fehlern befassen. Die Programmiersprache *Resilient X10* [9] beispielsweise, verfügt über eine *DeadPlaceException* welche bei einem Hardwareausfall ausgelöst wird und im Quellcode ähnlich einer Java Exception behandelt werden kann. Eine Möglichkeit permanente Fehler in GLB zu behandeln, ist es, während der Laufzeit Backups auf Nachbarknoten zu speichern, welche im Fall eines Fehlers die Arbeit des ausgefallenen Knotens übernehmen [10].
Eine Möglichkeit permanente und flüchtige Fehler zu behandeln, wird in *ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection* [1] vorgestellt. ACR verwendet Checkpoints und Replikation, um Silent Data Corruption und Hardwareausfällen vorzubeugen. Dabei werden zu Beginn einige Knoten als Ersatz markiert. Diese Ersatzknoten

werden nicht vom Programm genutzt, sondern ersetzen bei einem permanenten Fehler die ausgefallenen Knoten. Die restlichen Knoten werden in zwei gleichgroße Mengen aufgeteilt. Dabei wird jeder Knoten aus der ersten Menge mit einem Knoten aus der zweiten verbunden. Zwei so verbundene Knoten werden *buddies* genannt. Beide Mengen führen nun gleichzeitig dasselbe Programm aus. Während der Laufzeit werden auf den Knoten lokale Checkpoints erzeugt. Der Nutzer muss dabei eine Funktion implementieren, die festlegt, welche Daten in den Checkpoints gespeichert werden. Ein lokaler Checkpoint dient als ein remote Checkpoint für den Buddy-Knoten. Bei einem permanenten Fehler sendet der Buddy-Knoten des ausgefallenen Knotens seinen lokalen Checkpoint an einen der Ersatzknoten. Dieser ersetzt somit den ausgefallenen Knoten und übernimmt dessen Arbeit, indem er vom Checkpoint aus dieselbe Arbeit verrichtet wie sein neuer Buddy-Knoten. Um einen flüchtigen Fehler zu erkennen, sendet jeder Knoten der ersten Menge seinen lokalen Checkpoint nach dessen Erstellung an den jeweiligen Buddy-Knoten. Dort werden der übersendete, sowie der lokale Checkpoint miteinander verglichen. Beinhalten die Checkpoints nicht dieselben Daten, werden beide Knoten zum letzten sicheren lokalen Checkpoint zurückgesetzt und die Arbeit wird erneut ausgeführt. Um die Korrektheit von ACR zu gewährleisten, wird davon ausgegangen, dass während der Vergleiche der Checkpoints keine flüchtigen Fehler auftreten.

# 4 Behandlung flüchtiger Fehler

Im diesem Kapitel werden zwei Varianten beschrieben, die im Rahmen dieser Arbeit implementiert wurden um die Fehlertoleranz gegenüber flüchtigen Fehlern in GLB umzusetzen. Im Gegensatz zu ACR werden bei den Implementierungen keine Checkpoints erstellt, da nur die Fehlertoleranz gegenüber flüchtigen Fehlern Bestandteil der Arbeit ist. Wie bei ACR wird auch hier davon ausgegangen, dass während der Vergleiche keine flüchtigen Fehler auftreten.

## 4.1 Grundkonzept

Das Grundkonzept orientiert sich an der Funktionsweise von ACR. Es werden zunächst die verfügbaren Places in zwei gleichgroße Gruppen aufgeteilt. Die Places in der ersten Gruppe werden im Folgenden als LeadingPlaces bezeichnet. Die übrigen werden ShadowPlaces genannt. Beim Programmstart wird jeder LeadingPlace mit einem ShadowPlace verbunden. Zunächst führen die LeadingPlaces wie in Kapitel 2.2.3 beschrieben ihre Arbeit aus. Vor dem Aufruf von `process(n)` werden zwei Kopien der TaskQueue erstellt. Die erste wird an den ShadowPlace gesendet, welcher asynchron `process(n)` startet um die nächsten `n` Tasks der ihm gesendeten TaskQueue zu berechnen. Ist die Berechnung abgeschlossen, sendet der ShadowPlace die TaskQueue zurück an den LeadingPlace. Gleichzeitig wird auf dem LeadingPlace ebenfalls `process(n)` auf der zweiten Kopie der TaskQueue gestartet. Haben sowohl der Leading- als auch der ShadowPlace ihre Berechnungen abgeschlossen, werden die neu berechneten TaskQueues mit einer compare-Methode verglichen. Diese Vergleichsmethode muss vom Nutzer implementiert werden. Für die Bachelorarbeit wurde eine Vergleichsmethode für den verwendeten UTS-Algorithmus implementiert. In der Methode wird die Anzahl der bisher gezählten Baumknoten der TaskQueues miteinander verglichen. Schlägt der Vergleich fehl, werden die Berechnungen auf dem Leading- sowie dem ShadowPlace wiederholt, indem erneut zwei Kopien der ursprünglichen TaskQueue erstellt werden. Ist der Vergleich erfolgreich, wird die TaskQueue des LeadingPlaces durch

die berechnete Kopie ersetzt. Erst nachdem durch den Vergleich sichergestellt wurde, dass die TaskQueue korrekt berechnet wurde, wird auf Stehlanfragen anderer Worker eingegangen, welche die TaskQueue durch Aufrufe von `split()` bzw. `merge()` verändern können. Danach wird die Arbeit wie in 2.2.3 fortgesetzt. In Abb. 1 wird dieser Ablauf verdeutlicht. Obwohl diese Variante dazu führt, dass flüchtige Fehler erkannt und behandelt werden können, ist sie sehr ineffizient, da vor jedem einzelnen Arbeitsschritt eines Places eine Kopie der gesamten TaskQueue erzeugt wird und an den ShadowPlace gesendet werden muss, bevor dieser mit der Arbeit beginnen kann. Außerdem muss der ShadowPlace nach dem Arbeitsschritt seine gesamte TaskQueue auch wieder an den LeadingPlace zurücksenden, bevor der Vergleich der Queues auf dem LeadingPlace stattfinden kann. Die Vergleichsmethode trägt zum Zeitaufwand bei. Der Grund dafür ist, dass die Anzahl der bisher gezählten Baumknoten in einem Result-Konstrukt gespeichert wird, welches Teil der TaskQueue ist.

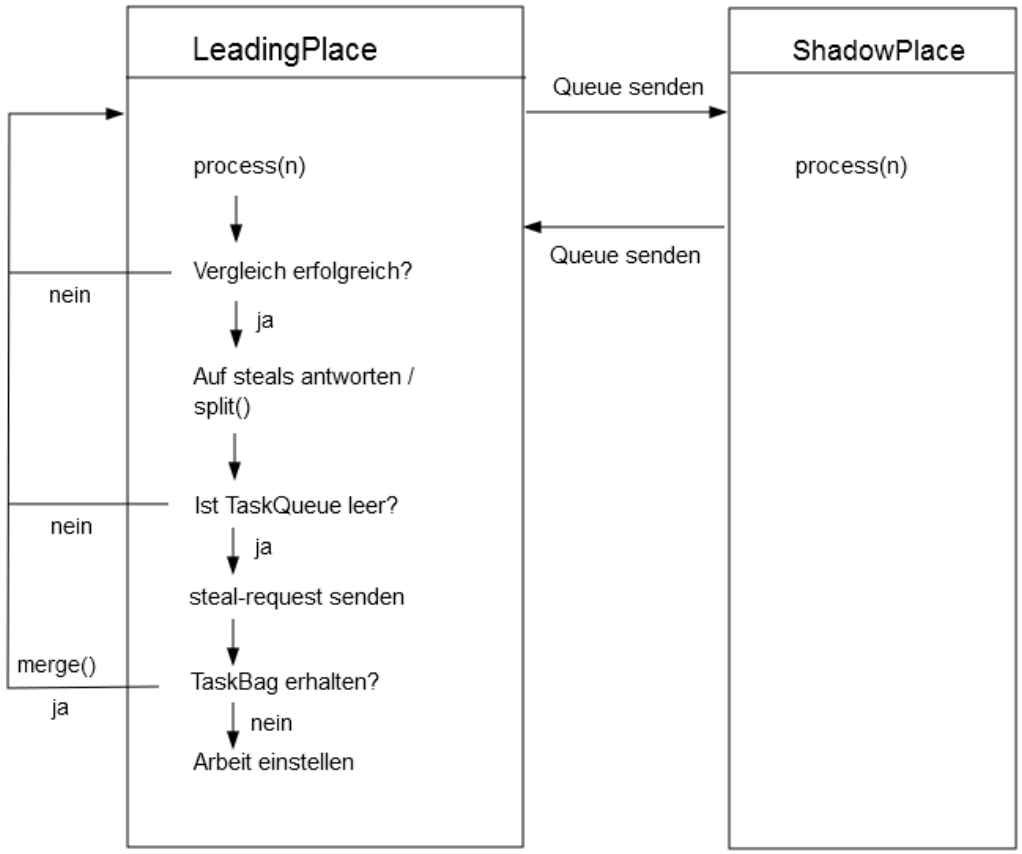


**Abb. 1 : Ablauf Variante 1**

## 4.2 Verbesserungsansatz

In der nächsten Variante wurde der Kommunikationsaufwand verringert um die Performance zu verbessern. Insbesondere ist das Senden der kompletten TaskQueue nur noch in Sonderfällen nötig. Wie in der ersten Variante werden auch hier die Places in Leading- und ShadowPlaces aufgeteilt. Die ShadowPlaces werden allerdings gleich zu Beginn des Programms mit denselben TaskQueues initialisiert wie ihre jeweiligen LeadingPlaces. Beim Aufruf von `process(n)` auf dem LeadingPlace muss jetzt nur noch `process(n)` auf dem ShadowPlace aufgerufen werden, da beide Places von Beginn an über dieselbe Queue verfügen. Dies verringert die Kommunikation zwischen den Places, da nicht mehr vor jedem Arbeitsschritt eine Kopie der Queue an den ShadowPlace gesendet werden muss. Eine Kopie der TaskQueue wird dennoch auf dem LeadingPlace erstellt, um im Fall eines flüchtigen Fehlers die TaskQueue zurücksetzen zu können.
Um die Queues auf Leading- und ShadowPlace identisch zu halten, müssen die Stehlanfragen von anderen Places beachtet werden.
Ruft ein LeadingPlace die `split()` – Methode auf um die Hälfte seiner TaskQueue an einen anderen Place zu senden, muss auch sein ShadowPlace `split()` aufrufen, um seine eigene TaskQueue ebenfalls zu halbieren. Der TaskBag der bei diesem Aufruf auf dem ShadowPlace entsteht kann verworfen werden, da nur gewährleistet werden muss, dass der ShadowPlace dieselben Tasks in der TaskQueue hat wie sein LeadingPlace.
Beim Aufruf von `merge()` ist nun wieder Kommunikation erforderlich. Wenn ein LeadingPlace durch einen steal einen TaskBag von einem anderen LeadingPlace erhält, fügt er dessen Tasks mit `merge()` in seine TaskQueue ein. Zuvor muss nun eine Kopie dieses TaskBags erstellt werden, welche dann an den ShadowPlace gesendet wird. Dort werden die Tasks des TaskBags ebenfalls mit `merge()` an die TaskQueue des ShadowPlaces angefügt. Die Aufrufe von `split()` bzw. `merge()` auf dem ShadowPlace werden synchron aufgerufen, was gewährleistet, dass Leading- und ShadowPlace auch nach einem steal über dieselben TaskQueues bzw. dieselben Tasks verfügen.

Im Falle eines flüchtigen Fehlers ist allerdings wieder größerer Kommunikationsaufwand nötig. Sollte der Vergleich von Leading- und ShadowPlace fehlschlagen, muss wie in der ersten Variante eine Kopie der ursprünglichen TaskQueue des LeadingPlace an den ShadowPlace gesendet werden, da `process(n)` die berechneten Tasks aus der TaskQueue entfernt. Die ursprüngliche TaskQueue ist noch korrekt, da der LeadingPlace wie weiter oben beschrieben zunächst auf einer lokalen Kopie seiner eigenen TaskQueue Berechnungen ausführt.

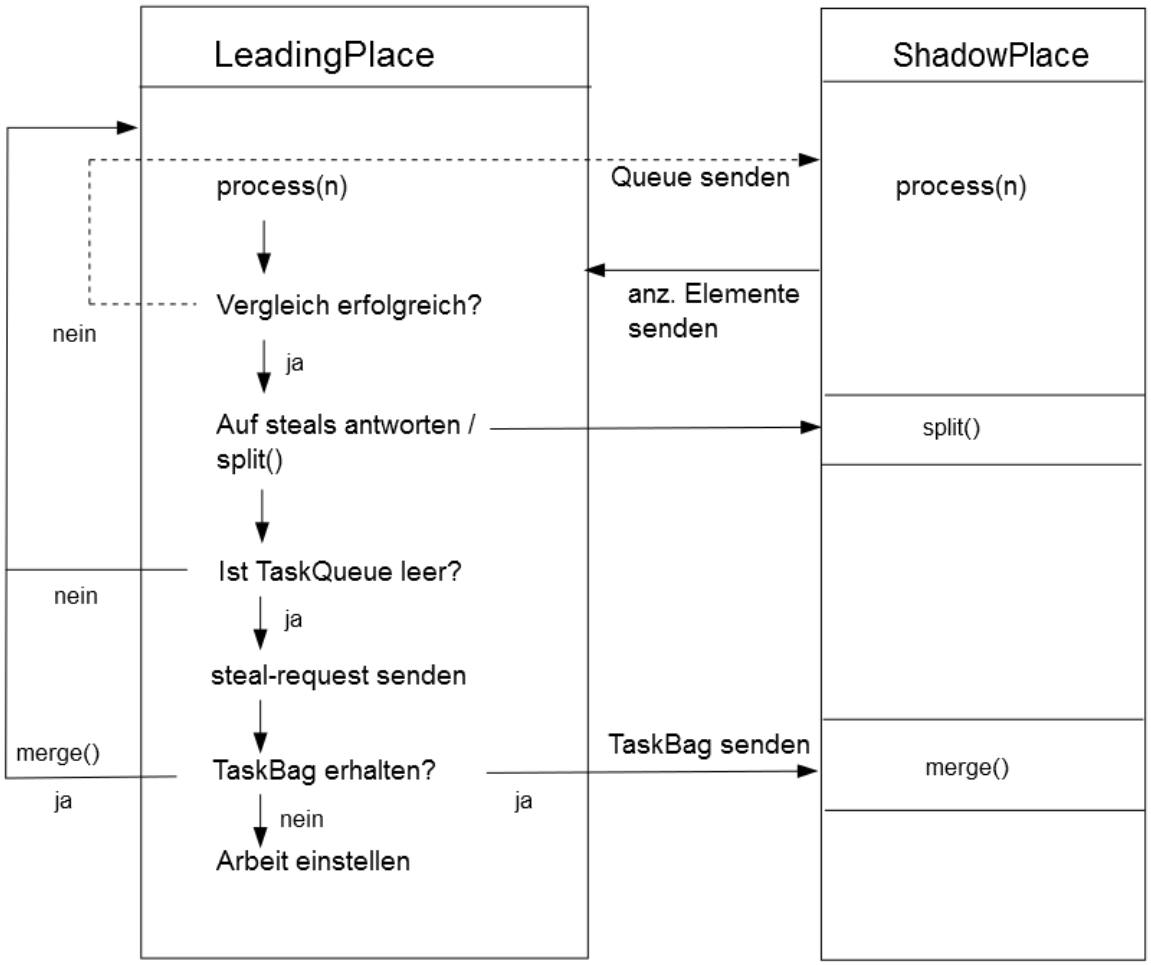


**Abb. 2 : Ablauf Variante 2**

Um den Kommunikationsaufwand weiter zu verringern wurde außerdem die Vergleichsmethode abgeändert. Anstelle des Result-Konstrukts der TaskQueues werden nun

die Anzahl der Tasks in den Queues verglichen. Auf diese Weise muss nach der Berechnung des ShadowPlaces nicht mehr die gesamte TaskQueue an den LeadingPlace zurückgesendet werden, sondern nur noch ein Integer Wert. Dieser Vergleich stellt sicher, dass bei der Berechnung eines Tasks bzw. Baumknotens die gleiche Anzahl Kindknoten auf dem Leading- und ShadowPlace generiert wurden. Es wird angenommen, dass kein flüchtiger Fehler während des Vergleichs auftritt. In den meisten Fällen würde ein flüchtiger Fehler während des Vergleichs die Korrektheit des Programms nicht beeinflussen, da bei einem fehlgeschlagenen Vergleich die Berechnung erneut ausgeführt wird. Die Ausnahme bestände darin, dass ein eigentlich fehlerhafter Vergleich durch *Silent Data Corruption* als korrekt angesehen wird. In diesem Fall würde das Programm in einen Deadlock laufen, da der nächste Arbeitsschritt immer fehlschlagen würde.

# 5 Implementierung

In diesem Abschnitt wird die Implementierung der in Kapitel 4 eingeführten Algorithmen genauer beschrieben. Nach einer kurzen Beschreibung der ursprünglichen GLB-Variante wird zunächst die Initialisierung der ShadowPlaces erklärt. Anschließend wird der reguläre Ablauf des Workers erläutert. Schließlich werden die Änderungen für Variante 1 und Variante 2 beschrieben.

## 5.1 GLB-Implementierung

Im Folgenden werden die wichtigsten Klassen der bereits implementierten GLB-Variante beschrieben:

**GLBParameters:**

Enthält die Parameter für GLB, wie beispielsweise `n` oder die Anzahl der zu verwendenden Places die beim Aufruf des Programmes eingelesen werden.

**TaskBag/TaskQueue:**

Diese beiden Klassen sind die in Kapitel 2.2.2 beschriebenen Interfaces, welche vom Nutzer mit den dazugehörigen Methoden wie `split()` und `merge()` implementiert werden muss.

**GLBResult:**

In GLBResult werden die Teilergebnisse der Tasks in einem Array gespeichert. Jede TaskQueue beinhaltet ein GLBResult. GLBResult ist eine abstrakte Klasse und muss somit auch vom Nutzer implementiert werden.

**GLB:**

Eine Instanz der GLB-Klasse muss im Nutzerprogramm erstellt werden, um GLB nutzen zu können. Im Konstruktor werden die Worker mit den Start-Tasks initialisiert. Die Worker beginnen ihre Arbeit durch den Aufruf von `run()` oder `runParallel()` welche die `processStack()` – Methode auf einem Worker, bzw. auf allen Workern aufrufen, je nachdem, ob die Start-Tasks auf einen oder alle Places verteilt wurden. Am Ende beider Methoden wird `collectResult()` aufgerufen, welches den vom Nutzer implementierten Reduzierungsoperator verwendet, um das Gesamtergebnis aus den Teilergebnissen der einzelnen Worker zusammenzufügen.

**Worker:**

Im Konstruktor der Worker-Klasse, welcher aus der unten beschriebenen GLB-Klasse heraus aufgerufen wird, werden mit dem lifeline graph work-stealing Algorithmus die IDs der lifeline Places für diesen Worker in einem Array gespeichert. Die Hauptarbeit des Workers geschieht in der processStack() – Methode, deren genaue Funktion in 5.3 näher ausgeführt wird.

## 5.2 ShadowPlaces

Der erste Schritt ist die Initialisierung der ShadowPlaces. Die dazu nötigen Änderungen wurden im Konstruktor der GLB-Klasse vorgenommen. In diesem wurde zunächst das APGAS-Konstrukt GlobalRef erzeugt. Dafür sind hier zwei Parameter notwendig: Eine Liste der verfügbaren Places, sowie eine Klasse, welche später über GlobalRef angesprochen werden soll. In diesem Fall handelt es sich dabei um den Worker. GlobalRef ist ein Objekt, das von jedem Place aus aufgerufen werden kann, und ein auf dem Place lokales Objekt zurückliefert. Um die ShadowPlaces zu erzeugen, wird GlobalRef nur noch mit k Places initialisiert, wobei k die Hälfte der verfügbaren Places darstellt. Danach wird ein zweites GlobalRef erstellt. Als Parameter erhält dieses die zweite Hälfte der Places und die ShadowWorker-Klasse. Beim ShadowWorker handelt es sich um eine stark vereinfachte Variante des Workers, die nur über zwei Attribute und drei Methoden verfügt. Bei den Attributen handelt es sich zum einen um die vom ShadowWorker verwendete TaskQueue und zum anderen um die ID des LeadingPlaces. Diese wird direkt im Konstruktor des ShadowWorkers gesetzt. Die drei Methoden dienen dazu, um vom LeadingPlace aus jeweils `process(n)`, `split()` und `merge()` auf der Queue des ShadowWorkers aufrufen zu können.

Wie bereits in 2.2.3 beschrieben, erhält bei Programmen, in denen die Anzahl der Tasks nicht von Beginn an feststeht (Bsp. UTS), zunächst nur ein Place die Start-Tasks. Für Variante 2 wird deshalb an dieser Stelle eine Kopie der TaskQueue des LeadingPlaces erstellt, und an dessen ShadowPlace gesendet, damit beide über dieselben Start-Tasks verfügen.

## 5.3 Worker Variante 1

Die Hauptarbeit des Workers findet in der Methode `processStack()` statt, welche aufgerufen wird, sobald die Places initialisiert und die Start-Tasks verteilt sind.
Andere wichtige Methoden des Workers sind `distribute()`, `reject()` und `steal()`. In `steal()` wird eine Stehlanfrage wie in 2.2.3 beschrieben an einen anderen Place gesendet. Solange der Worker keine Antwort erhält, wartet er und führt keine Arbeit aus. `distribute()` wird aufgerufen, um einem anderen Place eine positive Antwort auf dessen Stehlanfrage zu senden. Dabei wird dem *thief* die Hälfte der eigenen Tasks in einem TaskBag gesendet, damit er diesen mit `merge()` in seine TaskQueue einfügen kann. Der *thief* verlässt danach die Warteschleife in `steal()` und kann weiterarbeiten. `reject()` gibt dem *thief* eine negative Antwort auf den Stehlanfrage und sendet dementsprechend auch keine Tasks. Die Methode `processStack()` führt vier Schritte aus:

1. `process(n)` wird aufgerufen um bis zu `n` Tasks aus der TaskQueue abzuarbeiten
2. `distribute()` wird aufgerufen, um auf Stehlanfragen anderer Places zu reagieren und dementsprechend eigene Tasks abzugeben
3. `reject()` wird aufgerufen, um Stehlanfragen abzulehnen, falls nicht genügend Tasks verfügbar sind
4. `steal()` wird aufgerufen, um Stehlanfragen an andere Places zu senden, falls die eigene TaskQueue keine Tasks mehr hat

Wie man in Abb. 3 sehen kann, wird nach jedem `process(n)`–Aufruf versucht, Tasks auf *thieves* zu verteilen. Erst wenn keine Tasks mehr zu bearbeiten sind – also `queue.process(n)` false liefert – wird eine Stehlanfrage gesendet. Wird dieser abgelehnt, stellt der Worker seine Arbeit ein.

```
1 do {
2   while (queue.process(n)) {
3       distribute(st);
4       reject(st);
5   }
6   empty.set(true);
7   reject(st);
8   synchronized (waiting) {
9       cont = steal(st) || 0 < queue.size();
10      this.active.set(cont);
11      size = this.queue.size();
12  }
13} while (cont);
```

**Abb. 3 : processStack in Worker**

**Änderungen am Worker**

Für die erste Variante wurde zunächst die while-Schleife (Zeilen 2-5) angepasst. Die vorgenommenen Änderungen werden in Abb. 4 dargestellt. Wie bereits in Kapitel 4 beschrieben, werden zunächst zwei Kopien der TaskQueue erzeugt. Die hier verwendete `copy()`–Methode muss vom Nutzer erstellt werden, um die jeweilige Implementierung der TaskQueue kopieren zu können. In der do-while-Schleife (Zeile 7-27 in Abb. 4) wird nun mit *asyncAt* der ShadowPlace angesprochen. Ihm wird die Kopie der lokalen TaskQueue zugewiesen und anschließend `shadowProcess()` aufgerufen. Beim Parameter *st* handelt es sich um das in 5.2 beschriebene GlobalRef Objekt, welches nach dem Aufruf von `process(n)` benötigt wird um die vom ShadowPlace berechnete Queue zurück an den LeadingPlace zu übergeben. Dies geschieht, indem ähnlich wie in Zeile 11 von Abb. 4 mit einem *asyncAt* das übergebene GlobalRef Objekt angesprochen wird. Dort wird der Variablen *compareQueue* auf dem LeadingPlace die auf dem ShadowPlace berechnete Queue zugewiesen.

Da die Arbeit auf dem ShadowPlace asynchron gestartet wurde, wird auf dem LeadingPlace zeitgleich `process(n)` auf der zweiten Kopie der TaskQueue ausgeführt. Das die Anweisungen umschließende finish-Konstrukt stellt dabei sicher, dass sowohl der LeadingPlace, als auch der ShadowPlace ihre `process(n)` Aufrufe abgeschlossen haben.

Anschließend werden *tempQueue* und *compareQueue* mit der vom Nutzer implementierten *compareResult* Methode verglichen. Ist der Vergleich erfolgreich, wird die lokale TaskQueue durch eine der Kopien ersetzt. Sind nun noch Tasks in der TaskQueue enthalten, werden mit `distribute()` bzw. `reject()` die Stehlanfragen der anderen Places beantwortet. Wird die while-Schleife (Zeile 4) verlassen, wird die processStack-Methode wie in Abb. 3 Zeile 6 fortgesetzt.

```
1 do {
2   working = true;
3   compResult = false;
4   while (working) {
5       tempQueue = queue.copy();
6       shadowQueue = queue.copy();
7       do {
8           finish(() -> {
9               asyncAt(places().get(shadowPlace), () -> {
10                  // give the copied queue to the shadowWorker
11                  shadowRef.get().queue = shadowQueue;
12                  // process the copied queue
13                  shadowRef.get().shadowProcess(st, n);
14              });
15              // process Tasks here
16              working = tempQueue.process(n);
17          });
18
19          compResult = tempQueue.compareResult(compareQueue, n);
20
21          //reset queue copies
22          if(compResult == false) {
23              tempQueue = queue.copy();
24              shadowQueue = queue.copy();
25          }
26
27      } while (!compareResult);
28
29      queue = tempQueue;
30
31      if (working) {
32          distribute(st, shadowRef);
33          reject(st);
34      }
35  }
36  ...
37}
```

**Abb. 4 : processStack Variante 1**

**Vergleich**

Die vorliegende GLB-Variante verfügt die TaskQueue über eine GLBResult Klasse, welche ebenfalls vom Nutzer implementiert werden muss. In GLBResult befindet sich ein result-Array, in dem die Ergebnisse der berechneten Tasks der TaskQueue gespeichert werden. Haben alle Worker ihre Arbeit eingestellt, wird das Endergebnis des Programms berechnet, indem das result-Array aller TaskQueues der Worker zusammengerechnet wird. Im Beispielprogramm UTS handelt es sich um ein Array vom Typ *long*, welches die Anzahl der gezählten Baumknoten enthält.

Die compareResult-Methode in der ersten Variante wird in der TaskQueue implementiert und vom LeadingPlace aus aufgerufen. Die vom ShadowPlace berechnete TaskQueue dient als Parameter. In der Methode werden die beiden result-Arrays der TaskQueues verglichen und dementsprechend *true* bzw. *false* zurückgegeben.

## 5.4 Worker Variante 2

**Änderungen am Worker**

Die `processStack()` – Methode ist ähnlich wie in Variante 1, bis auf den Unterschied, dass die TaskQueue des LeadingPlace nur im Fehlerfall an den ShadowPlace gesendet wird (siehe Abb. 5). Um sicherzustellen, dass die TaskQueues von Leading- und ShadowPlace zur Berechnung identisch sind, wurden Änderungen an den Methoden `distribute()` und `processLoot()` vorgenommen. Letztere wird im Rahmen von `distribute()` aufgerufen, wenn ein Worker durch eine erfolgreiche Stehlanfrage einen TaskBag erhält und fügt dessen Tasks mit `merge()` in seine eigene TaskQueue ein. Führt ein LeadingPlace `split()` aus, um in `distribute()` seine Tasks mit einem *thief* zu teilen, wird auf dem ShadowPlace mit *at* ebenfalls `split()` aufgerufen, damit dessen TaskQueue auch halbiert wird. Bekommt der LeadingPlace einen TaskBag so wird dieser mit *at* an den ShadowPlace gesendet und dort ebenfalls mit `merge()` in dessen TaskQueue eingefügt.

```
1 do {
2   working = true;
3   compareResult = false;
4   while (working) {
5       tempQueue = queue.copy();
6       do {
7           finish(() -> {
8               asyncAt(places().get(shadowPlace), () -> {
9                   // process the Tasks at the shadowPlace
10                  shadowRef.get().shadowProcess(st, n);
11              });
12              // process Tasks here
13              working = tempQueue.process(n);
14          });
15
16          compareResult = tempQueue.compareResult(shadowQueueSize, n);
17
18          // reset queues
19          if(compareResult == false) {
20              tempQueue = queue.copy();
21              shadowQueue = queue.copy();
22              at(places().get(shadowPlace), () -> {
23                  shadowRef.get().queue = shadowQueue;
24              });
25          }
26
27      } while (!compareResult);
28
29      queue = tempQueue;
30
31      if (working) {
32          distribute(st, shadowRef);
33          reject(st);
34      }
35
36  }
37  ...
38}
```

**Abb. 5 : processStack Variante 2**

### Vergleich

In dieser Variante werden nicht mehr die result-Arrays verglichen, sondern nur noch die Anzahl der Tasks in der TaskQueue. Dementsprechend wird nicht mehr die gesamte TaskQueue des ShadowPlaces an den LeadingPlace gesendet, sondern nur noch ein Integerwert, welcher als Parameter für den Aufruf von `compareResult()` dient.

# 6 Performance

Nach der Implementierung beider Varianten wurden Zeitmessungen lokal auf einem Rechner mit 4 Prozessoren mit je 3.10GHz und 8GB Arbeitsspeicher durchgeführt, um die Performance gegenüber der ursprünglichen GLB-Variante zu testen. Für die Zeitmessung wurde jeweils das arithmetische Mittel aus fünf Durchläufen gebildet, da die Ausführungszeit leicht schwankt. Die Parameter wurden so gewählt, dass jeweils zwei Places, bzw. zwei Paare aus Leading- und ShadowPlace, die in 2.3 beschriebene UTS-Variante berechnen. Für `n` wurden nacheinander die Werte 100, 500 und 1000 verwendet. Der erstellte Suchbaum wird mit einem Randomseed initialisiert und hat eine maximale Tiefe von 11 und einem Verzweigungsgrad von 4.
Um einen flüchtigen Fehler zu simulieren, wurde ein Zähler in den ShadowWorker eingefügt, damit bei jedem 10. Aufruf von `process(n)` auf dem ShadowPlace ein falscher Wert zum Vergleichen gesendet wird. Das Auftreten von Fehlern in dieser Größenordnung dient lediglich zur Zeitmessung und ist in der Realität eher nicht zu erwarten.
Bei den Zeitmessungen wurde außerdem zwischen der Zeit zur Initialisierung der Places und der Bearbeitungszeit der Worker unterschieden.

| **Original GLB** | **n = 100** | **n = 500** | **n = 1000** |
|---|---|---|---|
| **setup-time** | 0.412s | 0.399s | 0.400s |
| **process-time** | 1.756s | 1.849s | 1.733s |

**Tabelle 1 : Zeitmessung Original GLB**

In Tabelle 1 werden die Messergebnisse der ursprünglich zu verändernden GLB-Variante abgebildet. Die *setup-time,* also die Zeit, die für die Initialisierung der Places notwendig ist, bleibt unabhängig vom gewählten `n`. Dieses beeinflusst vor allem die *process-time* welche die Dauer der Berechnung aller Tasks angibt. Bei einem kleinen `n` werden mehr Stehlanfragen durchgeführt und es wird öfter auf Stehlanfragen reagiert, sodass die Ausführungszeit bei einem kleineren `n` höher ist.

| Variante 1 | n = 100 | n = 500 | n = 1000 |
|---|---|---|---|
| **setup-time** | 0.552s | 0.593s | 0.603s |
| **process-time** | 32.114s | 14.487s | 9.679s |
| **process-time mit Fehler** | 36.202s | 15.126s | 10.558s |

**Tabelle 2 : Zeitmessung Variante 1**

Tabelle 2 zeigt die Messungen für die erste implementierte Variante, bei der vor jedem Aufruf von `process(n)` die gesamte TaskQueue an den ShadowPlace gesendet wird. Wie zu erkennen ist, ist die *setup-time* nur geringfügig höher, was daran liegt, dass zusätzlich zu den LeadingPlaces nun auch die ShadowPlaces initialisiert werden müssen. Die *process-time* ist hier deutlich höher als bei der ursprünglichen Variante. Dies liegt zum einen daran, dass vor jedem `process(n)` – Aufruf die TaskQueue an den ShadowPlace und anschließend das Ergebnis des ShadowPlaces wieder zurück an den LeadingPlace gesendet werden muss. Die Kommunikation zwischen den einzelnen Places ist dabei relativ zeitaufwendig. Außerdem werden Leading- und ShadowPlace nach einem Aufruf synchronisiert, damit der Vergleich durchgeführt werden kann, was die Ausführungszeit zusätzlich erhöht.

| Variante 2 | n = 100 | n = 500 | n = 1000 |
|---|---|---|---|
| **setup-time** | 0.632s | 0.624s | 0.587s |
| **process-time** | 26.692s | 11.787s | 9.066s |
| **process-time mit Fehler** | 30.029s | 12.941s | 9.837s |

**Tabelle 3 : Zeitmessung Variante 2**

Wie in Tabelle 3 zu sehen, ist die zweite implementierte Variante selbst im Fehlerfall schneller als die Erste, da, wie in Kapitel 4 beschrieben, die nötige Kommunikation zwischen den Places verringert wurde. Ansonsten verhält sich die Performance ähnlich zu Variante 1. Der Anstieg der *process-time* bei auftretenden Fehlern verhält sich ebenfalls wie in Variante 1. Das liegt daran, dass in beiden Varianten das Auftreten eines Fehlers dadurch behandlet wird, indem wieder die gesamte TaskQueue an den ShadowPlace gesendet wird.

Tabelle 4 zeigt die *process-time* aller Varianten im Vergleich.

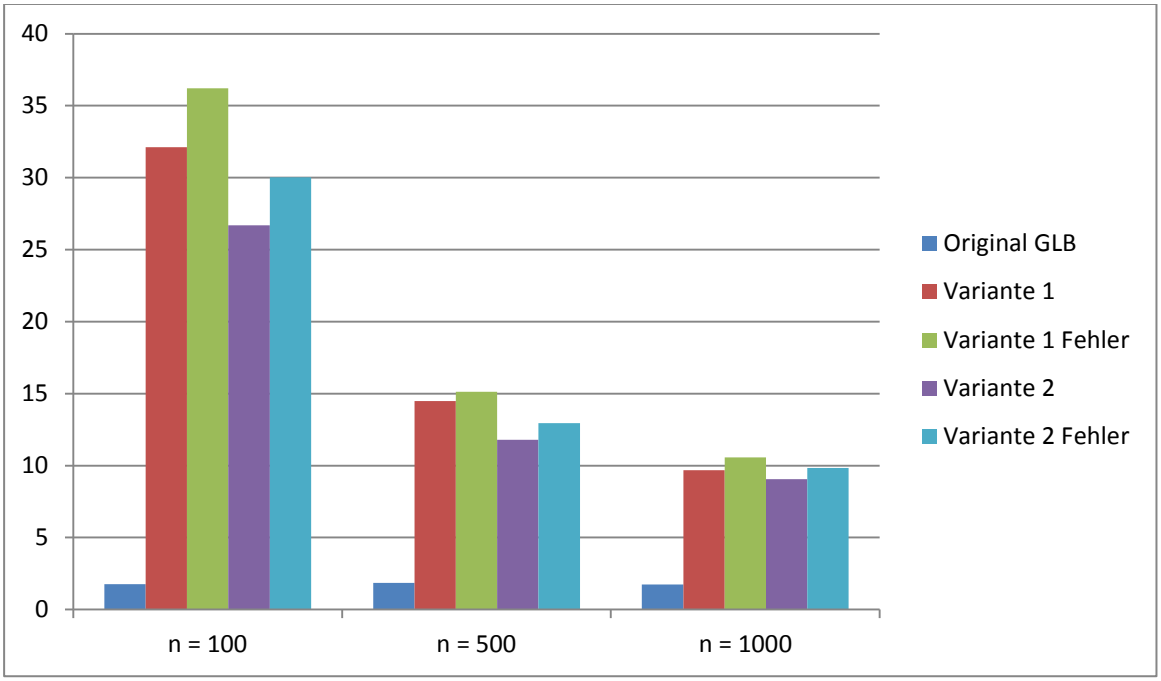


**Tabelle 4 : Zeitmessungen Vergleich**

# 7 Zusammenfassung

Das Thema der Bachelorarbeit war die Erweiterung einer vorliegenden GLB-Variante um die Behandlung auftretender flüchtiger Fehler. Dazu wurden zwei verschiedene Varianten implementiert. In der Ersten sendet ein Place vor jedem Arbeitsschritt eine Kopie der gesamten TaskQueue an einen weiteren Place um das Ergebnis der Berechnung anschließend vergleichen zu können. In der zweiten Variante wurde die nötige Kommunikation zwischen den Places reduziert, indem nur noch Teile der TaskQueue an bestimmten Stellen gesendet werden, um an Performance zu gewinnen. Die Änderungen an der ursprünglichen GLB-Variante wurden gering gehalten, sodass zur Benutzung der erweiterten Variante nur zwei Methoden des Interfaces TaskQueue vom Nutzer implementiert werden müssen. Dabei handelt es sich zum einen um die Vergleichsmethode, welche die Ergebnisse der Arbeitsschritte von Leading- und ShadowPlace vergleicht, und zum anderen um eine Methode die eine Kopie der vom Nutzer implementierten TaskQueue erzeugt, die verwendet wird, wenn eine Kopie der TaskQueue an den ShadowPlace gesendet werden muss. Die Zeitmessungen wurden anschließend mit einer bereits implementierten Variante des UTS-Benchmarks vorgenommen.

Das Hauptaugenmerk bei der Implementierung der Fehlertoleranten Varianten lag bei der korrekten Erkennung und Behandlung flüchtiger Fehler. Der Performancetest zeigt, dass für einen messbaren Zeitverlust der beiden Varianten eine hohe Anzahl flüchtiger Fehler auftreten muss. Dennoch sind beide Varianten deutlich langsamer als die ursprüngliche GLB-Variante, was jedoch zu erwarten war, da für den gewählten Ansatz zu Fehlerbehandlung zusätzliche Kommunikation zwischen den einzelnen Places, sowie Synchronisation nach einem Arbeitsschritt nötig ist.

# Literaturverzeichnis

[1] X. Ni, E. Meneses, N. Jain und L. V. Kalé, „ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection,“ In: Proceedings of international conference high performance computing, networking, storage and analysis, SC 13. ACM, 2013.

[2] „X10 Homepage,“ [Online]. Available: http://x10-lang.org/. [Zugriff am 6 November 2015].

[3] V. Saraswat, G. Almasi, G. Bikshand, C. Cascaval und D. Cunningham, „The Asynchronous Partitioned Global Address Space Model,“ 'The First Workshop on Advances in Message Passing (co-located with PLDI 2010)' , Toronto, Canada ., 2010.

[4] O. Tardieu, „The APGAS Library: Resiliant Parallel and Distributed Programming in Java 8,“ ACM New York, Proceedings of the ACM SIGPLAN Workshop on X10, 2015.

[5] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat und M. Takeuchi, „GLB: Linfeline-based Global Load Balancing library in X10,“ ACM New York, PPAA '14 Proceedings of the first workshop on Parallel programming for analytics applications Seiten 31-40, 2014.

[6] V. Saraswat, P. Kambadur, S. Kodali, D. Grove und S. Krishnamoorthy, „Lifeline-based Global Load Balancing,“ ACM New York, PPoPP '11 Proceedings of the 16th ACM symposium on Principles and practice of parallel programming Pages 201-212, 2011.

[7] J. Posner, „Global Load Balancing and intra-node synchronization with the Java Framework APGAS, Masterarbeit,“ Universität Kassel, Fachbereich Elektrotechnik/Informatik, Fachgebiet Programmiersprachen/-methodik, 2015 (in Arbeit).

[8] S. Oliver, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan und C.-W. Tseng, „UTS: An Unbalanced Tree Search Benchmark,“ LCPC’06 Proceedings of the 19th international conference on Languages and compilers for parallel computing, Seiten 235–250, 2006.

[9] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi und O. Tardieu, „Resilient X10 - Efficient failure-aware programming,“ ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’14),, 2014.

[10] C. Fohry, M. Bungart und J. Posner, „Fault Tolerance Schemes For Global Load Balancing in X10,“ Scalable Computing: Practice and Experience, Vol 16, No 2, 2015.