Fehlertoleranz und Elastizität für ein Framework zur globalen Lastenbalancierung

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Vorgelegt im Fachbereich 16 - Elektrotechnik/Informatik
der Universität Kassel

von M.Sc. Marco Bungart

Betreuerin: Prof. Dr. Claudia Fohry

Zweitgutachter: Prof. Dr. Thomas Rauber

Eingereicht am 10. September 2018 in Kassel

Verteidigt am 30. November 2018 in Kassel

Danksagung

„Das Glück ist das einzige was sich verdoppelt, wenn man es teilt."

(Albert Schweitzer)

Auch wenn die vorliegende Arbeit in selbstständiger Arbeit entstanden ist, so haben mich viele Leute während meiner Zeit als Doktorand unterstützt. Mein Dank geht an:

Prof. Dr. Claudia Fohry
Ohne dich wäre es mir nicht möglich gewesen, zu promovieren. Danke dafür, dass du meine Doktormutter bist. Danke für die Geduld mit mir. Danke für die vielen Anregungen und Gespräche.

Prof. Dr. Thomas Raubert
Für die Begutachtung meiner Dissertation.

M. Sc. Nikolas Luke
Du warst mein erster sozialer Kontakt – und später Freund – in Kassel. Ich erinnere mich gerne an die Schachpartien, die wir gegeneinander gespielt haben, auch wenn ich immer unterlegen war.

M. Sc. Jonas Posner
Es ist selten, dass man einen Kollegen hat, der in derselben Richtung forscht und mit dem man sich tagtäglich über seine Forschung austauschen kann. Noch seltener ist es, wenn sich daraus eine Freundschaft entwickelt.

M. Sc. Benjamin Herwig
Wie oft standen wir draußen, haben zusammen eine Zigarette geraucht und dabei geredet? Ohne dich wäre meine Zeit in Kassel nicht so schön gewesen. Auch wenn wir oft unterschiedlicher Meinung sind, so möchte ich dich als Freund nicht mehr missen. Und Java ist natürlich besser als C!

M. Sc. Christoph Eickhoff und M. Sc. Simon-Lennert Räsch
Der Kontakt zu euch ist leider erst viel zu spät zu Stande gekommen. Wenn ihr mal wieder ein Brettspiel auf seine Tauglichkeit als digitale Umsetzung für ein Studentenprojekt testen wollt, dann helfe ich dabei gerne.

Meinen Freunden Malte Breuer, Dr. Tobias Guggenmoser, Dr. Jörg Blank, Christiane Klassen und Dipl.-Phys. Florian Lotter
Sei es der Urlaub in Japan, die regelmäßigen Rollenspielabende, die gemeinsame Silvesterfeiern oder andere Aktivitäten. All dies hat mir geholfen, neue Energie zu sammeln und frisch gestärkt weiterzuarbeiten.

Meiner Mutter, Jutta Bungart
für alles.

ありがとうございました！

(arigatōgozaimashita!, Vielen Dank!)

 doi:10.17170/kobra-2018122577

Erklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbständig, ohne unerlaubte Hilfe Dritter angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Dritte waren an der inhaltlichen Erstellung der Dissertation nicht beteiligt; insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren durch mich verwendet worden.

Kassel, 10. September 2018

(Marco Bungart)

Zusammenfassung

Die Anzahl an Rechenknoten in Hochleistungsrechnern wächst stetig. In solchen Systemen nimmt die Bedeutung von Fehlertoleranz zu, da die Wahrscheinlichkeit eines permanenten Knotenausfalls ebenfalls stetig wächst.

Fehlertoleranz gegenüber permanenten Knotenausfällen wird typischerweise durch Checkpointing auf Systemebene realisiert. Findet das Checkpointing jedoch auf Anwendungsebene statt, können Laufzeitvorteile erzielt werden. Diese Implementierungen sind allerdings zeitintensiv und fehleranfällig. Anwendungen, die gegenüber solcher permanenten Knotenausfälle resistent sind, werden im weiteren Verlauf dieser Arbeit *fehlertolerant* genannt.

Neben der Fehlertoleranz steht auch die optimale Auslastung von Hochleistungsrechnern im Fokus der Forschung. Die Auslastung kann durch dynamische Umverteilung der Rechenressourcen zur Laufzeit verbessert werden. Hierzu müssen die Applikationen in der Lage sein, zur Laufzeit Rechenressourcen freizugeben und neue Rechenressourcen in die Berechnung aufzunehmen. Solche Applikationen werden *elastisch* genannt.

In dieser Arbeit entwickeln wir einen Algorithmus zur Realisierung von Fehlertoleranz und Elastizität für Taskpools. Die Fehlertoleranz arbeitet auf Applikationsebene und repliziert relevante Daten regelmäßig in den Hauptspeicher eines anderen Rechenknotens. Werden neue Rechenressourcen zu einer laufenden Berechnung hinzugefügt, so werden die hinzugefügten Rechenressourcen über die Lastenbalancierung des Taskpools mit Arbeit versorgt. Zu diesem Algorithmus entwickeln wir mehrere Varianten.

Wir haben unseren Algorithmus und seine Varianten als wiederverwendbares Framework in der parallelen Programmiersprache X10 implementiert.

Die experimentelle Auswertung mit drei verschiedenen Benchmarks hat ergeben, dass unser Framework und seine Varianten einen Overhead gegenüber einer nicht-fehlertoleranten Implementierung zwischen $4,17\%$ und $55,72\%$ haben. Für die Wiederherstellung ausgefallener Rechenknotens sowie das elastische Hinzufügen neuer Knoten konnten wir keine messbaren Laufzeiteinbußen feststellen.

Abstract

The number of computational nodes in a HPC cluster rises constantly. In such systems fault tolerance gains importance since the probability of a permanent node failure increases.

Fault tolerance with respect to permanent node failures is typically achieved through checkpointing on system-level. Checkpointing on application-level may achieves execution time improvements. Its implementation is, however, time-consuming and error-prone. Applications that can tolerate permanent node failures are called *fault-tolerant* in this thesis.

In addition to fault tolerance, techniques to improve cluster utilization are a major topic of current research. The utilization can be improved through a dynamic redistribution of computational resources among application at runtime. For this, the applications must be able to remove resources from and add new resources to their computation at runtime. Applications supporting this feature are called *elastic*.

In this work, we develop an algorithm to provide fault tolerance and elasticity for task pools. It implements application-level fault tolerance and replicates relevant data of a node into the main memory of another. When new nodes are added to a running computation, they receive work through the load balancing mechanism of the underlying task pool.

We develop different variants of our algorithm. The algorithm is implemented as a reusable framework in the parallel programming language X10.

An experimental evaluation with three different benchmarks showed that our framework and its variants achieve an overhead between 4.17% and 55.72% compared to a non-fault-tolerant implementation. For the restore of failed nodes and the elastic addition of new nodes, experimental results show no measurable execution time losses.

Inhaltsverzeichnis

 doi:10.17170/kobra-2018122577

doi:10.17170/kobra-2018122577

Abbildungsverzeichnis

Quellcodeabbildungsverzeichnis

Tabellenverzeichnis

doi:10.17170/kobra-2018122577

1. Einleitung

1.1. Motivation

Moderne Hochleistungsrechner sind komplexe Systeme, die aus zehntausenden untereinander vernetzten Rechenknoten und hunderttausenden Prozessorkernen[1] bestehen. Es wird davon ausgegangen, dass die nächste Generation von Hochleistungsrechnern aus hunderttausenden Rechenknoten und mehreren Millionen Prozessorkernen bestehen wird [1]. Die Komplexität dieser Systeme schlägt sich bei der Entwicklung effizienter Programme nieder: Sowohl die Nutzung der Prozessorkerne innerhalb eines Rechenknotens als auch die Kommunikation zwischen den Rechenknoten soll effizient gestaltet werden. Mit der steigenden Komplexität der Programme steigt auch ihr Wartungsaufwand. Ein Programm sollte daher nicht komplexer als nötig gestaltet werden.

Mit der Anzahl der Komponenten in einem Hochleistungsrechner steigt außerdem die Wahrscheinlichkeit, dass ein Fehler auftritt. Fehler, die den Ausfall eines Rechners bzw. Prozesses (und all seiner Daten) signalisieren, werden *permanente Fehler* oder auch *Fail-Stop Fehler* genannt [2]. Ein Maß für die Häufigkeit von Fehlern ist die mittlere Zeit zwischen zwei Fehlern (engl. *Mean Time Between Failures*, kurz *MTBF*). Heutzutage wird die MTBF für permanente Fehler auf modernen Clustern in Tagen bis Stunden angegeben [3, 4]. Für die nächste Generation von Hochleistungsrechnern wird eine MTBF im Minutenbereich erwartet [3, 5]. Somit sinkt die Wahrscheinlichkeit einer erfolgreichen Programmausführung. Besonders betroffen sind Programme, deren Ausführung Stunden oder Tagen benötigt und die dabei viele Rechenknoten zeitgleich verwenden. Durch diese Entwicklungstendenz ist Fehlertoleranz gegenüber permanenten Fehlern zu einem wichtigen Thema der aktuellen Forschung im Bereich der Parallelverarbeitung geworden.

Fehlertoleranz wurde bereits Mitte der 80er Jahre untersucht [6, 7]. Im Fokus der Forschung standen zu diesem Zeitpunkt Bitflips in verschiedenen

[1] exklusive Beschleuniger wie GPUs und Koprozessoren

Hardwarekomponenten, die das Ergebnis verfälschen können. Diese Fehler werden als *Silent Errors* bezeichnet. Ein Großteil der Silent Errors wird in heutiger Hardware durch integrierte Error Correction Codes erkannt und, soweit möglich, behoben. Im weiteren Verlauf dieser Arbeit verwenden wir den Begriff „Fehlertoleranz“ für Fehlertoleranz gegenüber permanenten Fehlern.

Um einen Hochleistungsrechner möglichst effizient auszulasten, ist eine globale Verwaltung der Rechenknoten notwendig. Dies übernimmt ein *Batchsystem*. Nutzer interagieren nicht direkt mit den Rechenknoten, sondern definieren Rahmenbedingungen für die Ausführung ihres Programms, wie beispielsweise die Anzahl an benötigten Knoten und die erwartete Ausführungszeit und schicken dies in Form eines *Jobs* an das Batchsystem. Das Batchsystem übernimmt die Verwaltung der Rechenknoten nur auf Jobebene. Das auszuführende Programm trägt die Verantwortung dafür, dass die ihm zugewiesenen Rechenknoten möglichst optimal ausgelastet werden. Sind laufende Jobs in der Lage, Knoten während der Berechnung aufzunehmen oder abzugeben, so kann die Auslastung eines Hochleistungsclusters um bis zu 25% gesteigert werden [8]. Die Fähigkeit eines Programms, zur Laufzeit neue Knoten in die Programmausführung aufzunehmen oder Knoten abzugeben, nennen wir *Elastizität*. Das Potenzial elastischer Jobs hinsichtlich einer besseren Auslastung von Hochleistungsrechnern wurde bereits in den 90er Jahren erkannt [9].

Wie eingangs erwähnt, schlägt sich die Komplexität eines Hochleistungsrechners in den Anforderungen an die Programmentwicklung nieder. Eine Möglichkeit, die Komplexität des Quellcodes zu reduzieren, ist die Einführung von Abstraktionen. So stellen Programmiersprachen Abstraktionen von Maschinencode dar, die für Menschen besser verständlich sind. Ebenso sind Bibliotheken und Frameworks Abstraktionen: Der Nutzer kennt die geforderte Eingabe und das Verhalten, muss jedoch kein Wissen über den internen Ablauf haben. Für die Realisierung der Kommunikation zwischen Rechnenknoten ist das Message Passing Interface (kurz *MPI*) [10, 11] der Quasi-Standard. MPI ist hardwarenah und bietet somit eine niedrige Abstraktionsschicht. Der Standard ist in verschiedenen Implementierungen, beispielsweise Open MPI [12] und MPICH [13] umgesetzt. User Level Failure Mitigation (kurz *ULFM*) [14, 15, 16] erweitert MPI um Fehlertoleranz. Mit Hilfe von MPI bzw. ULFM kann auch Elastizität realisiert werden.

Eine höhere Abstraktionsschicht bietet das PGAS-Programmiermodell (**P**artitioned **G**lobal **A**ddress **S**pace, deutsch partitionierter globaler Adress-

 doi:10.17170/kobra-2018122577

raum). In diesem wird der Arbeitsspeicher aller an der Berechnung beteiligten Knoten als ein globaler, partitionierter Speicher angesehen. Ein *Place* ist definiert als eine Menge an Rechenressourcen zusammen mit einer Speicherpartition. Jeder Place kann auf jede Speicherpartition zugreifen, der Zugriff auf eine lokale Speicherpartition ist jedoch effizienter als der Zugriff auf eine entfernte Speicherpartition. Das PGAS-Programmiermodell wurde auf verschiedene Arten realisiert. So sind beispielsweise Chapel [17, 18] und X10 [19, 20, 21] eigenständige Sprachen. UPC [22, 23] ist eine Erweiterung der Sprache C, wohingegen UPC++[24, 25] (Unified Paralel C++), APGAS für Java [26, 20, 21] und PCJ [27, 28, 29] Beispiele für Bibliotheken sind, die das PGAS-Programmiermodell in C++ bzw. Java umsetzen. Von den genannten Umsetzungen bieten X10 (seit Version 2.4.1 [30]), APGAS (seit dem ersten Release) und PCJ (in naher Zukunft [31]) dem Programmierer die Möglichkeit, permanente Fehler abzufangen und zu behandeln. Elastizität unterstützen nur X10 und APGAS.

Entwurfsmuster sind eine weitere Möglichkeit, die Komplexität der Programmentwicklung und -wartung zu reduzieren. Durch den richtigen Einsatz von Entwurfsmustern wird der Abstraktionsgrad des Quellcodes erhöht [32]. Für parallele Programme wird oft das Entwurfsmuster des *Taskpools* [33] verwendet. Seine Kernidee ist die Aufteilung der zu verrichtenden Arbeit in Arbeitspakete – die *Tasks*. Die Implementierung wird hierbei unterteilt in

- die Berechnungsvorschrift, also den von den Tasks auszuführenden Code, sowie
- die Taskverwaltung.

Die Aufgabe der Taskverwaltung besteht in der dynamischen Verteilung der Tasks auf die zur Verfügung stehenden Ausführungseinheiten – die *Worker*. Worker können Prozessorkerne oder Rechenknoten sein. Es bietet sich an, die Verwaltung möglichst allgemeingültig in einem Framework zu implementieren, um sie für verschiedene Problemstellungen nutzen zu können. Ein Nutzer eines Taskpool Frameworks realisiert seine Berechnung durch Angabe der von den Tasks auszuführenden Berechnungsvorschrift. Die Tasks werden in einer geeigneten Datenstruktur gespeichert. Meist wird hierzu eine verteilte Datenstruktur verwendet, bei der jeder Worker einen lokalen Pool an lauffähigen Tasks verwaltet. Worker entnehmen Tasks aus dem lokalen Pool und arbeiten

diese durch Ausführung der Berechnungsvorschrift ab. Dabei können neue Tasks entstehen, die in den lokalen Pool eingefügt werden.

Ist der Arbeitsaufwand ungleich über die Worker verteilt, wird durch eine Umverteilung der Tasks zur Laufzeit eine Lastenbalancierung erreicht. Eine umfassend untersuchte Technik zur Lastenbalancierung ist das *Work-Stealing* [34]. Hat ein Worker keine lauffähigen Tasks mehr, so stiehlt er Tasks bei einem anderen Worker. Lifeline-based Global Load Balancing [35] ist ein Algorithmus zur Verwaltung eines verteilten Taskpools. Er ist im GLB-Framework der parallelen Programmiersprache X10 implementiert [36]. Beim Lifeline-Schema sind Dieb und Opfer aktiv in einen Stehlvorgang involviert: Der Dieb sendet eine Stehlanfrage an das Opfer und dieses antwortet auf die eingegangene Anfrage. Daher wird dieses Vorgehen *kooperatives Stehlen* genannt. Sendet das Opfer dem Dieb Tasks, so nennen wir diese nachfolgend *Beute*.

Taskpools bilden eine günstige Abstraktionsschicht zur Umsetzung von Fehlertoleranz: Fallen Worker aus, so müssen lediglich ihre Tasks auf die überlebenden Worker verteilt und die bisherigen Teilergebnisse übernommen werden.

Allerdings geht durch den Ausfall eines Workers auch der mit ihm assoziierte Speicher – und damit sein lokaler Pool und sein Teilergebnis – verloren. Eine Möglichkeit, diese Daten auch nach Ausfall des Workers verfügbar zu halten, ist ihr regelmäßiges Sichern in den Hauptspeicher eines anderen Workers. Alternativ kann die durch Stehlvorgänge entstehende Redundanz genutzt werden. Die explizite Datensicherung nennen wir nachfolgend *Checkpointing* und die entsprechende Sicherungskopie *Replika*. Checkpointing verursacht Overhead, sodass es in geeigneten Zeitabständen und mit möglichst geringen Kommunikationskosten realisiert werden muss. Hierzu kann blockierende oder nichtblockierende Kommunikation verwendet werden. Nichtblockierende Kommunikation ist effizienter, da sie Asynchronität erlaubt. Hierdurch können beispielsweise Berechnungen parallel zur Kommunikation erfolgen.

Die Replikas verschiedener Worker müssen stets konsistent gehalten werden. Diese Forderung ist insbesondere bei Stehlvorgängen schwierig zu gewährleisten, da hier Tasks aus dem lokalen Pool des Opfers entfernt, über das Netzwerk übertragen und zum lokalen Pool des Diebs hinzugefügt werden.

Um auf Ausfälle von Workern reagieren zu können, müssen sie zunächst erkannt werden. Dann muss sich eine Komponente des Programms um die Wiederherstellung eines korrekten Programmzustands kümmern. Beispielsweise kann diese Aufgabe von einem geeignet gewählten Worker übernommen werden.

 doi:10.17170/kobra-2018122577

Die Wiederherstellung gestaltet sich besonders dann schwierig, wenn vor Erreichen eines gültigen Programmzustands weitere Fehler auftreten. Falls sowohl die lokalen Daten eines Workers als auch seine Replika verloren gehen, liegt ein *endgültiger Datenverlust* vor, der die Wiederherstellung unmöglich macht. Auch solche Situationen müssen erkannt und sinnvoll behandelt werden.

Als weiteres Ziel neben der Fehlertoleranz sollten Taskpools in der Lage sein, neue Worker in laufende Berechnungen einzubinden, also Elastizität zu gewährleisten. Diese Worker sowie ihre Ansprechpartner unter den bisherigen Workern können ebenfalls von Ausfällen betroffen sein.

In einer asynchronen Implementierung kann ein Worker gleichzeitig in mehrere Vorgänge involviert sein. Beispielsweise kann er zu einem Zeitpunkt Dieb eines Stehlvorgangs, Opfer eines anderen Stehlvorgangs, für die Wiederherstellung eines ausgefallenen Workers verantwortlich und Ansprechpartner für einen neuen Worker sein. Derartige Situationen müssen angemessen behandelt werden, sodass es niemals zu falschen Ergebnissen oder Deadlocks kommen kann. Ferner sollen diese robust behandelt werden, sodass in möglichst vielen Fällen das korrekte Ergebnis ausgegeben wird.

1.2. Beitrag dieser Arbeit

Die vorliegende Arbeit löst die obige Problemstellung exemplarisch für das GLB-Framework von X10. Diese Umgebung wurde gewählt, da X10 Fehlertoleranz und Elastizität unterstützt sowie GLB als Teil seiner quelloffenen Klassenbibliothek zur Verfügung stellt [36].

In X10 bezieht sich die Unterstützung für Fehlertoleranz auf die Erkennung permanenter Placeausfälle (siehe Abschnitt 1.1). Durch einen solchen Ausfall gehen sowohl die Rechenresourcen als auch die zum Place gehörige Speicherpartition verloren. Nach derartigen Ausfällen wird eine Ausnahme ausgelöst. Die Lebendigkeit eines konkreten Places kann abgefragt werden. [30]

Bezüglich Elastizität unterstützt X10 das Hinzufügen von Places. Hierzu wird das Programm ein weiteres Mal mit zusätzlichen Kommandozeilenparametern gestartet. Die neuen Places können im Programm über Abfragefunktionen ermittelt werden. Ihre aktive Einbindung in die Berechnung obliegt dem Anwenderprogramm.

GLB setzt voraus, dass auf jedem Place genau ein Worker gestartet wird. Somit geht bei einem Placeausfall auch immer genau ein Worker verloren.

Die vorliegende Arbeit beschreibt ein Schema zur Gewährleistung von Fehlertoleranz und Elastizität für das GLB-Framwork. Wir beschreiben einerseits einen entsprechenden Algorithmus und andererseits seine Implementierung als Erweiterung von GLB. Der Algorithmus ist bzgl. seiner Herangehensweise nicht auf GLB beschränkt, sodass sich einige Konzepte auf andere Taskpools übertragen lassen.

Wir realisieren die Fehlertoleranz durch Checkpointing. Dazu werden die Places in einem Ring angeordnet. Wir setzen voraus, dass jedem Place eine eindeutige und fortlaufende ID, beginnend bei `0`, zugeordnet ist. In einem Programmlauf mit `N` Places erhalten die Places somit die IDs `0`, `1`, ..., `N-1`. Im Ring sendet Place `P` seine Replika an Place `P+1`, Place `N-1` sendet seine Replika an Place `0`. Somit sendet ein Place `P` seine Replika an seinen *Nachfolger* oder *Backup-Place* (kurz: `Back(P)`) und speichert die Replika seines *Vorgängers* (kurz: `Forth(P)`). Diesen Aufbau bezeichnen wir als *Backup-Ring*.

Um das Checkpointing effizient zu gestalten, wird weitestgehend auf Synchronisation verzichten. Jeder Place entscheidet selbstständig, wann seine Replika zu aktualisieren ist. Replikas werden nach der Abarbeitung einer gewissen Anzahl an Tasks und zusätzlich während Stehlvorgängen aktualisiert. Dabei haben wir mit verschiedene Techniken experimentiert. Beispielsweise können die Replikas durch einen aktuelleren Stand ersetzt werden. Alternativ werden lediglich die inkrementellen Änderungen seit dem letzten Checkpointing übertragen. Zusätzlich nutzen wir Datenredundanz, um die Kommunikationskosten während eines Stehlvorgangs gering zu halten.

Für den Stehlvorgang haben wir ein Fehlertoleranzprotokoll entwickelt. Dieses stellt sicher, dass die Beute zu jedem Zeitpunkt entweder zum Taskpool des Opfers oder zum Taskpool des Diebes gehört. Fallen Dieb und Opfer aus, so kann die Beute eindeutig einem der ausgefallenen Places zugeordnet und für diesen wiederhergestellt werden.

Places überwachen sich gegenseitig, um Ausfälle zeitnah zu erkennen. Jeder Place `P` überprüft regelmäßig, ob sein Nachfolger `Back(P)` noch lebendig ist. Zudem überwacht jeder Place diejenigen Places, von denen er noch Nachrichten erwartet. Fällt ein Place `P` aus, so ist sein Vorgänger `Forth(P)` für den Start der Wiederherstellung verantwortlich. Aufgrund der Definition des Backup-Rings gehen mit dem Ausfall eines Places `P` seine Daten und die Replika seines Vorgängers `Forth(P)` verloren. Das Programm wird in einen gültigen Zustand überführt, indem:

 doi:10.17170/kobra-2018122577

1. die Tasks in der Replika von `P` in den Taskpool von `Back(P)` aufgenommen und das Teilergebnis von `P` zum Teilergebnis von `Back(P)` hinzugefügt werden,
2. `Back(P)` eine Replika an seinen Nachfolger sendet, und
3. `Forth(P)` seine Replika auf `Back(P)` speichert.

Dabei ist ein Worker in der Lage, endgültigen Datenverlust durch den Ausfall eines Places `P` und dessen Backup-Place `Back(P)` zu erkennen und der Programmlauf wird abgebrochen.

Zur Umsetzung der Elastizität haben wir ein Protokoll entwickelt, welches den Backup-Ring um neu hinzugefügte Places erweitert. Werden einem mit `N` Places gestarteten Programmlauf `M` neue Places hinzugefügt, so vergibt X10 an die neuen Places die IDs `N`, `N+1`, ... `N+M-1`. Die Integration neuer Places wird über Place `0` gesteuert. Wir passen den Backup-Ring so an, das nach erfolgreicher Erweiterung Place `N-1` seine Replikas auf Place `N`, Place `N` seine Replikas auf Place `N+1`,..., Place `N+M-1` seine Replikas auf Place `0` speichert. Das Elastizitätsprotokoll wurde so entworfen, dass auch Ausfälle während einer Erweiterung abgefangen und korrekt behandelt werden.

X10 bietet die Möglichkeit, Programme nach C++ (genannt *Native X10*) oder nach Java (genannt *Managed X10*) zu compilieren. Elastizität wurde bisher nur für Managed X10 unterstützt. Wir haben die MPI-Netzwerkschnittstelle für Native X10 um Elastizität erweitert.

Die Performance von GLB sowie des vorgestellten Algorithmus und seiner Varianten wurde mittels dreier verschiedener Benchmarks (Unbalanced Tree Search (kurz *UTS*) [37], Betweenness Centrality (kurz *BC*) [38] und dem `N`-Damen Problem (kurz *NQueens*) experimentell ausgewertet. Alle Frameworks wurden mit Native X10 und MPI-Unterstützung kompiliert und ausgeführt. Mit 144 Places auf 12 Rechenknoten ergibt sich für die fehlertoleranten Algorithmen bei Messungen mit verschiedenen Benchmarks ein Overhead von $4,17$ bis $55,72\%$ gegenüber GLB. Der Wiederherstellungsaufwand bei einem Placeausfall ist vernachlässigbar, ebenso wie das elastische Hinzufügen von Places zur Programmlaufzeit.

1.3. Veröffentlichungen

Im Rahmen der Dissertation sind die folgenden Veröffentlichungen entstanden:

- Marco Bungart, Claudia Fohry und Jonas Posner. *Fault-Tolerant Global Load Balancing in X10*. In: *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2014. DOI: 10.1109/synasc.2014.69

  Eigener Beitrag: Algorithmenentwicklung; Implementierung; Durchführung und Auswertung der Experimente

- Claudia Fohry, Marco Bungart und Jonas Posner. *Towards an Efficient Fault-Tolerance Scheme for GLB*. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. 2015. DOI: 10.1145/2771774.2771779

  Eigener Beitrag: Implementierung; Durchführung und Auswertung der Experimente

- Claudia Fohry, Marco Bungart und Jonas Posner. *Fault Tolerance Schemes for Global Load Balancing in X10*. In: *Scalable Computing: Practice and Experience* 16.2 (2015). DOI: 10.12694/scpe.v16i2.1088

  Eigener Beitrag: Beteiligung an Algorithmenentwicklung; Implementierung; Anpassung der Benchmarks; Durchführung und Auswertung der Experimente

- Claudia Fohry und Marco Bungart. *A Robust Fault Tolerance Scheme for Lifeline-Based Taskpools*. In: *Proceedings of the 45th International Conference on Parallel Processing Workshops*. 2016. DOI: 10.1109/icppw.2016.40

  Eigener Beitrag: Implementierung; Durchführung und Auswertung der Experimente

- Marco Bungart und Claudia Fohry. *A Malleable and Fault-Tolerant Task Pool Framework for X10*. In: *Proceedings of the IEEE International Conference on Cluster Computing*. 2017. DOI: 10.1109/cluster.2017.27

  Eigener Beitrag: Algorithmenentwicklung; Implementierung; Durchführung und Auswertung der Experimente

- Marco Bungart und Claudia Fohry. *Extending the MPI Backend of X10 by Elasticity*. EuroMPI Poster. 2017. URL: https://www.mcs.anl.gov/eurompi2017/pics/posters/Fohry-EuroMPI2017-posterX10.pdf (besucht am 10. 09. 2018)

  Eigener Beitrag: Design und Implementierung

doi:10.17170/kobra-2018122577

- Claudia Fohry, Marco Bungart und Paul Plock. *Fault Tolerance for Lifeline-Based Global Load Balancing.* In: *Journal of Software Engineering and Applications* 10.13 (2017). DOI: 10.4236/jsea.2017.1013053

  Eigener Beitrag: Implementierung; Durchführung und Auswertung der Experimente

1.4. Struktur

Der Rest dieser Arbeit ist wie folgt gegliedert. Der aktuelle Stand der Forschung zu Fehlertoleranz und Elastizität wird in Kapitel 2 besprochen. Wir diskutieren Checkpoint-Restart als gängigen Ansatz für Fehlertoleranz und gehen auf verschiedene Grundansätze für Elastizität ein.

Kapitel 3 führt das benötigte Hintergrundwissen ein. Wir besprechen das PGAS- sowie das APGAS-Modell und die benötigten Sprachkonstrukte von X10. Des Weiteren führen wir GLB, sowie das von GLB verwendete Taskmodell ein.

In Kapitel 4 stellen wir unseren fehlertoleranten Algorithmus und seine Implementierung in GLB vor. Im Mittelpunkt stehen die konsistente Datenhaltung der Replikas sowie die zeitnahe Behandlung von Ausfällen. Wir entwickeln Protokolle zur Gewährleistung der Datenkonsistenz während Stehlvorgängen und zur Wiederherstellung ausgefallener Places. Zudem entwerfen wir ein Konzept zur zeitnahen Ausfallerkennung. Wir diskutieren, unter welchen Umständen eine Wiederherstellung möglich ist und wie zwischen wiederherstellbaren und nicht wiederherstellbaren Zuständen unterschieden werden kann.

Die Erweiterung um Elastizität ist Inhalt von Kapitel 5. Bevor die neuen Places in die Berechnung aufgenommen werden können, müssen auf jedem Place die benötigten Datenstrukturen – ein leerer Taskpool und ein leeres Teilergebnis – initialisiert werden. Danach werden die Places in den Backup-Ring integriert. Wir besprechen, wie Ausfälle, sowohl von alten wie auch von neuen Places, während dieses Prozesses behandelt werden.

In Kapitel 6 diskutieren wir einige Varianten, die den Stehlvorgang sowie das Schreiben der zu sichernden Daten beeinflussen. Für den Stehlvorgang entwickeln wir zwei Varianten. Für das Senden der Backups entwickeln wir einen inkrementellen Algorithmus, welcher nur Daten sendet, die sich seit dem letzten Checkpointing geändert haben.

Kapitel 7 beschreibt die Anpassungen, die wir am MPI-Backend von X10

vorgenommen haben, um Elastizität zu ermöglichen. Zunächst geben wir einen kurzen Einblick in die Funktionsweise von MPI und besprechen den Aufbau der abstrakten Netzwerkschnittstelle von X10 sowie ihrer MPI-Umsetzung. Darauf basierend beschreiben wir die nötigen Änderungen der MPI-Umsetzung.

Die experimentelle Bewertung der vorgestellten Frameworks erfolgt in Kapitel 8. Wir beschreiben dort die Testumgebung für die durchgeführten Experimente sowie ihre Zielsetzung. Wir erläutern und diskutieren die Messergebnisse und zeigen Stärken und Schwächen unserer Frameworks auf.

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 9.

 doi:10.17170/kobra-2018122577

2. Stand der Forschung

2.1. Fehlertoleranz

Im Bereich der Parallelverarbeitung ist die Fehlertoleranz zu einem Schwerpunkt der Forschung geworden [46, 47]. In Abschnitt 2.1.1 stellen wir die Fehlertoleranz-Unterstützung in verschiedenen Programmiersystemen vor.

Danach beschreiben wir in Abschnitt 2.1.2 verschiedene Verfahren zum Sichern und Wiederherstellen von Daten.

2.1.1. Unterstützung von Fehlertoleranz in Programmiersystemen

Im Folgenden beschreiben wir Programmiersysteme, die Fehlertoleranz in dem Sinne unterstützen, dass die Ausführung der Programme nicht abbricht, wenn ein Fehler auftritt. Die Fehlertoleranz der Applikation, also das Sichern und Wiederherstellen verloren gegangener Daten, erfolgt durch den Nutzer. Ausnahmen bilden Satin und Cilk-NOW.

Der MPI-Standard [10] unterstützt keine Fehlertoleranz. User Level Failure Mitigation [15, 14, 16] (kurz *ULFM*) erweitert den MPI-Standard um Konstrukte zur Behandlung von Fehlertoleranz. Die aktuelle Version 2.0 von ULFM ist kompatibel zur aktuellen Open MPI [12] Version 3.1.2. Fehler werden in ULFM erst durch Kommunikation mit einem ausgefallenen Prozess erkannt. Die Rekonstruktion der betroffenen Kommunikatoren muss explizit durch den Nutzer erfolgen. Wir gehen in Abschnitt 7.1 näher auf diesen Aspekt ein.

GASPI [48] (Global Address Space Programming Interface) ist eine API zur Umsetzung des PGAS-Modells, die in C und Fortran implementiert wurde. Sie bietet – ähnlich wie MPI – Routinen für asynchrone und einseitige Kommunikation an, arbeitet jedoch auf einer höheren Abstraktionsebene als MPI. Skalierbarkeit, Flexibilität und Fehlertoleranz waren die Hauptmotive beim Entwurf von GASPI. Anders als bei ULFM überwacht ein dedizierter

Fehlerdetektor-Prozess die Lebendigkeit aller anderen Prozesse [49]. Stellt der Fehlerdetektor fest, dass ein Knoten ausgefallen ist, informiert er mittels einseitiger Kommunikation alle anderen Knoten über den Ausfall. Die Rekonstruktion der Kommunikationsschicht wird ebenfalls über den Fehlerdetektor-Prozess realisiert und benötigt kein Mitwirken des Nutzers.

Die parallele PGAS Programmiersprache X10 [19, 20, 21] unterstützt Fehlertoleranz seit Version 2.4.1 [30]. Bei Zugriff auf einen ausgefallenen Place wird eine `DeadPlaceException` ausgelöst. Pro Place kann eine Methode als `PlaceRemoveHandler` registriert werden, die automatisch ausgeführt wird, wenn ein Place ausfällt. Über eine Methode `Place.isDead(...)` kann die Lebendigkeit eines Places geprüft werden. Wir besprechen diese drei Mechanismen in Abschnitt 3.1 im Detail. Für die mit X10 ausgelieferte MPI-Netzwerkschnittstelle wird Fehlertoleranz seit Version 2.5.4 [50] unterstützt. Sie basiert auf ULFM Version 1.1. Der API von X10 wurden vor kurzem zwei fehlertolerante Datenstrukturen hinzugefügt [51]. Eine Variante steht nur unter Managed X10 zur Verfügung und basiert auf der `IMap` von Hazelcast [52], einem System zur Verwaltung verteilter und fehlertoleranter Maps. Jedes in einer `IMap` gespeicherte Schlüssel-Wert-Paar wird durch Hazelcast auf mehrere Rechenknoten repliziert um Fehlertoleranz zu gewährleisten. Die zweite Variante ist eine in X10 geschriebene verteilte und fehlertolerante Map. Diese steht sowohl in Managed als auch in Native X10 zur Verfügung.

Die APGAS-Bibliothek für Java [26, 21, 20] unterstützt ebenfalls Fehlertoleranz. Zur Umsetzung dieser verwendet APGAS die `IMap` von Hazelcast als fehlertolerante Datenstruktur. Fehlertoleranz wird in ähnlicher Weise wie bei X10 unterstützt: der Zugriff auf einen ausgefallenen Place löst eine `DeadPlaceException` aus und pro Place kann eine Methode als `PlaceRemovedHandler` hinterlegt werden. Eine zu `Place.isDead(...)` vergleichbare Methode bietet die API von APGAS nicht an.

Die PGAS-Bibliothek PCJ [27, 28, 29] wird Fehlertoleranz in naher Zukunft unterstützten [31]. Die Fehlererkennung von PCJ nutzt einerseits TCP, andererseits eine regelmäßige Überwachung aller beteiligten Knoten, um Ausfälle festzustellen. Festgestellte Ausfälle werden gespeichert und erst an das Programm weitergegeben, wenn auf einen ausgefallenen Knoten zugegriffen werden soll oder aufgetretene Fehler abgefragt werden.

Satin [53] ist ein Programmiersystem für Task-basierte Programmierung. In Satin gibt es kein Konzept eines Rechenknotens. Die Tasks werden automatisch durch Satin verteilt, der Nutzer kann keinen Einfluss darauf nehmen, auf

 doi:10.17170/kobra-2018122577

welchem Knoten ein Task ausgeführt wird. Fällt ein Knoten aus, so gehen Tasks verloren. Zur Fehlerbehebung werden die verloren gegangenen Knoten neu gestartet. Hat ein Rechenknoten einen Task berechnet, der seinen Elterntask verloren hat, so teilt er allen anderen Knoten mit, dass er das Ergebnis für den entsprechenden Kindtask lokal gespeichert hat. Sobald ein Knoten den verloren gegangenen Elternknoten startet, fordert er das Ergebnis an.

Cilk-NOW [54] (Cilk-**N**etwork **O**f **W**orkstations) ist eine eigenständige parallele Programmiersprache, die auf Cilk [55] basiert. Sie erweitert Cilk um Fehlertoleranz und Elastizität. Ähnlich zu Satin arbeitet Cilk auf Tasks. Diese müssen jedoch unabhängig und frei von Seiteneffekten sein. Cilk-NOW realisiert Fehlertoleranz einerseits durch die Tasks, andererseits durch Checkpointing. Jede Workstation pflegt eine Liste an offenen Tasks, die sie an andere Workstations abgegeben hat. Fällt eine Workstation aus, prüfen die lebendigen Workstations, ob sie der ausgefallenen Workstation Tasks gesendet haben und führen diese neu aus. Somit wird Recovery realisiert. Cilk-NOW nutzt Checkpointing, um Totalausfälle abfangen zu können. Es wird vorausgesetzt, dass ein global zugängliches Dateisystem existiert. Für jeden Task wird eine Checkpointing-Datei angelegt. Der Zustand jedes Tasks wird regelmäßig in diese Datei geschrieben. Das Schreiben der Checkpoints erfolgt unkoordiniert. Die Begriffe „unkoordiniert“ „Recovery“ und „Checkpointing“werden weiter unten erklärt.

2.1.2. Fehlertoleranzverfahren

Das in Abschnitt 1.1 erwähnte Checkpointing, also das regelmäßige Sichern des Programmzustandes, bildet den de-facto Standard. Eine Übersicht über verschiedene Checkpointing-Realisierungen geben Elnozahy *et al.* in [56].

Die Checkpointing-Realisierungen lassen sich danach unterteilen, ob sie auf *System-Level* oder *User-Level* arbeiten [57]. System-Level Checkpointing benötigt keine Programmanpassungen und speichert den Zustand des gesamten verteilt ausgeführten Programms zu dem Zeitpunkt ab, zu dem der Checkpoint erstellt wird [58, 59]. User-Level Checkpointing basiert auf dem Ansatz, dass der Nutzer angibt, welche Daten in einem Checkpoint zu sichern sind [60, 61, 62]. Dies ermöglicht es, die Größe der Checkpoints zu reduzieren und somit die Laufzeit zu verbessern.

Werden die Checkpoints aller beteiligten Prozesse zum selben Zeitpunkt geschrieben, so wird dies auch *koordiniertes Checkpointing* genannt [63]. Durch

die benötigte Synchronisation entstehen zusätzliche Kosten. Die Checkpoints stellen dafür jedoch eine konsistente Sicht der Berechnung dar. Dem gegenüber steht das *unkoordinierte Checkpointing* [64, 65], bei dem jeder beteiligte Prozess autonom festlegt, wann er seinen Checkpoint aktualisiert.

Unkoordiniertes Checkpointing muss durch Message Logging ergänzt werden [66, 67]. Hierbei werden Nachrichten mitgeloggt, die zwischen zwei Checkpoints ausgetauscht werden. Diese werden im Fehlerfall erneut gesendet. Wird Message Logging in Verbindung mit unkoordiniertem Checkpointing genutzt, so kann es zu einem Domino-Effekt kommen. Der Ausfall eines Prozesses führt dann ggf. zum Rollback der gesamten Anwendung in den Startzustand [68]. Dieses Problem ist nicht allgemein, sondern nur für Applikationen mit bestimmten Voraussetzungen gelöst. Guermouche *et al.* [69] haben dies beispielsweise für Applikationen gelöst, die bei fehlerfreien Programmausführungen immer dieselben Nachrichten in derselben Reihenfolge austauschen. Koordiniertes Checkpointing kann um Message Logging ergänzt werden. Die Gefahr eines Domino-Effekts besteht dabei nicht. Wird Message Logging verwendet, so kann anstatt eines *Restart*s, der den Neustart der Applikation erfordert, ein *Recovery*, also eine Wiederherstellung ausgefallener Prozesse, durchgeführt werden.

Klassischerweise werden die Checkpoints auf die Festplatte geschrieben. Bei Verwendung von koordiniertem Checkpointing führt dies zu einer Vielzahl von zeitgleichen Schreibzugriffen [70]. Eine Alternative hierzu stellt das In-Memory Checkpointing dar [71]. Der benötigte Arbeitsspeicher limitiert diese Technik, da Checkpoints, vor allem bei Verwendung von System-Level Checkpointing, sehr groß werden können. Hybride Ansätze [4, 72, 73] schreiben Checkpoints sowohl in den flüchtigen Arbeitsspeicher als auch auf die Festplatte. Benoit *et al.* [74] verallgemeinern das Modell von Young/Daly [75, 76] auf ein k-stufiges Checkpointing-Verfahren.

Inkrementelles Checkpointing [77], also das Schreiben jener Daten, die sich seit dem letzten Checkpoint verändert haben, verringert das zu schreibenden Datenvolumen und damit die Laufzeit des Programms [3, 78].

Ist eine fehlertolerante Applikation in der Lage, nach einem Prozessausfall mit verminderter Anzahl an Prozessen die Ausführung fortzusetzen, sprechen wir von einer *schrumpfenden Wiederherstellung*, andernfalls von einer *nicht schrumpfenden Wiederherstellung* [79]. Bei einer nicht schrumpfenden Wiederherstellung muss der ausgefallene Prozess ersetzt werden, bevor die Berechnung fortgesetzt werden kann.

 doi:10.17170/kobra-2018122577

Algorithmen-basierte Realisierungen [6] können eine auf den Algorithmus zugeschnittene Fehlertoleranz bieten und haben meist den geringsten Overhead. Beispiele für Algorithmen-basierte Realisierungen, die gänzlich ohne Checkpoints auskommen, finden sich in [2, 80, 81, 82]. Die Entwicklung dieser Verfahren ist jedoch zeitintensiv und algorithmenspezifisch, sodass sie nicht für beliebige Algorithmen verwendet werden können.

Fehlertoleranz kann für eine bestimmte Gruppe von Anwendungen, beispielsweise für Taskpools mit einem bestimmten Taskmodell oder für Divide-And-Conquer-Algorithmen implementiert werden. Diese Implementierungen bieten die Verallgemeinerung der Fehlertoleranz auf die definierte Anwendungsgruppe und sind für diese optimiert. Wrzesinka *et al.* [83] haben ein solches System für Divide-And-Conquer Algorithmen in Satin realisiert. Jeder Task meldet sein Teilergebnis an seinen Elterntask zurück. Somit muss der Taskgraph explizit abgespeichert werden und kann zur Wiederherstellung von Tasks, die durch einen Ausfall verloren gegangen sind, genutzt werden. Die verloren gegangenen Tasks werden auf den überlebenden Knoten neu gestartet; Kindtasks der verloren gegangenen Tasks werden darüber informiert, auf welchen Knoten der Elterntask neu gestartet wurde. Einen ähnlichen Ansatz verfolgen Kestor *et al.* [84] für Fork-Join Algorithmen in Cilk-NOW. Posner und Fohry [85, 86] haben Fehlertoleranz für Taskpools mit unabhängigen Tasks durch unkoordiniertes Checkpointing mit APGAS und Hazelcast realisiert. Hierbei nutzen sie die von Hazelcast bereitgestellte `IMap` zum Sichern der Backups.

Weitere Programmiersysteme, die Fehlertoleranz unterstützen sind beispielsweise AllScale [87, 88], CHARM++ [89, 90], Erlang [91, 92] und OmpSs [93, 94].

2.2. Elastizität

Abschnitt 2.2.1 zeigt, dass der Nutzen elastischer Programm umfassend untersucht wurde. Dieser Abschnitt motiviert die Entwicklung elastischer Programme.

Die Unterstützung von Elastizität durch verschiedene Bibliotheken und Programmiersystem besprechen wir in Abschnitt 2.2.2.

2.2.1. Nutzen für Workload Manager

Es ist bekannt, dass die Suche nach einem optimalen Scheduling für Jobs ein NP-vollständiges Problem ist [95]. Das Problem vereinfacht sich, wenn elastische Jobs verwendet werden. Dies haben Feitelson und Rudolph [9] bereits 1995 festgestellt. Experimentelle Auswertungen zeigen, dass Job Scheduler die Auslastung eines Clusters um bis zu 25% steigern können, wenn der Scheduler mit elastische Jobs arbeitet [8]. Einige Batchsysteme, beispielsweise Torque [96, 97], Koala [98, 99] und ReSHAPE [100], können die Elastizität der auszuführenden Programme ausnutzen, um die Auslastung des Clusters zu verbessern.

2.2.2. Unterstützung durch Bibliotheken und Programmiersysteme

Der MPI-Standard [10] bietet über die Routinen `MPI_Port_open(...)`, `MPI_Comm_accept(...)` und `MPI_Comm_connect(...)` die Möglichkeit an, neue Prozesse in einen Programmlauf einzubinden. Eine Freigabe von Prozessen zur Laufzeit ist nicht möglich. Ein Beispiel für eine Implementierung geben Leopold, Süß und Breitbart [101, 102].

Compres *et al.* [103] haben eine Erweiterung des MPI-Standards vorgeschlagen, um Unterstützung für Elastizität zu erlauben. Cores *et al.* [104, 105] zeigen, wie für iterative MPI-Applikationen Checkpoint/Restart (siehe Abschnitt 2.1.2) genutzt werden kann, um Elastizität zu realisieren. Die Anzahl der MPI-Prozesse bleibt hierbei fest, die Anzahl der verwendeten Rechenknoten darf sich ändern. Werden Knoten der Berechnung hinzugefügt oder aus der Berechnung entfernt, werden die entsprechenden MPI-Prozesse migriert. Um neue Rechenknoten in die Berechnung aufnehmen zu können, müssen zu Programmstart mehrere MPI-Prozesse pro Knoten gestartet werden, da den neuen Knoten sonst keine Prozesse zugewiesen werden können. Ebenso kann es durch eine Verkleinerung dazu kommen, dass weitere Prozesse zu einem Rechenknoten hinzugefügt werden. Während der Migration wird der MPI-Kommunikator neu konfiguriert, bevor die Daten des Prozesses über den Checkpoint wiederhergestellt werden.

FLEX-MPI [106] bietet Elastizität für iterative MPI-Programme. Es besteht aus zwei Komponenten: einem Workload Manager sowie einer MPI-Erweiterung zum Hinzufügen neuer und Entfernen bestehender Prozesse. Der Workload

 doi:10.17170/kobra-2018122577

Manager fordert die Programme zum Hinzufügen bzw. Entfernen von Prozessen auf. Die MPI-Erweiterung stellt hierfür Methoden wie beispielsweise `XMPI_Spawn(...)` und `XMPI_Remove(...)` bereit. FLEX-MPI pflegt einen konsistenten Kommunikator `XMPI_Comm_World` über alle Prozesse in der Berechnung. Für die elastische Verteilung von Daten werden weitere Routinen wie `XMPI_Register_dense(...)` und `XMPI_Get_wsize(...)` bereitgestellt. FLEX-MPI ist nicht mit ULFM kompatibel.

Satin [53] unterstützt Elastizität über das verwendete Taskmodell. Hierbei wird das gleiche Konzept genutzt, dass auch die Fehlertoleranz in Satin ermöglicht (siehe Abschnitt 2.1.1): Tasks werden automatisch durch den Task Scheduler von Satin auf neu hinzugefügte Knoten verteilt.

Managed X10 unterstützt Elastizität seit Version 2.5.0. Durch das Setzen der Umgebungsvarialbe `X10_JOIN_EXISTING=[Hostname|IP-Adresse]:Port` vor Programmstart wird ein Programm zu einem bereits existierenden Programmlauf hinzugefügt. Managed X10 bietet die Möglichkeit, auf jedem Place eine Methode als `PlaceAddedHandler` zu registrieren, welche ausgeführt wird, sobald neue Places dem Programmlauf hinzugefügt wurden.

Buchwald, Mohr und Zwinkau [107] haben eine Sprache auf Grundlage von X10 entwickel. Die Sprache erlaubt das Schreiben von invasiven Applikationen. Solche Applikationen können neue Rechenressourcen per `invade` anfordern, die angeforderten Ressourcen über einen `infect` einbinden und mittels `retreat` wieder freigeben. Der Austausch der benötigten Daten geschieht für den Nutzer transparent im Hintergrund.

3. Grundlagen

Im folgenden Abschnitt 3.1 besprechen wir das PGAS-Modell sowie die auf dem PGAS-Modell basierende parallele Programmersprache X10. Eine Besprechung des GLB-Frameworks folgt in Abschnitt 3.2.

3.1. PGAS und X10

Das Partitioned Global Address Space (kurz *PGAS*) Modell ist ein paralleles Programmiermodell. Es betrachtet den Speicher aller an einer Berechnung beteiligten Rechenknoten als einen globalen, partitionierten Addressraum. Jeder Rechenknoten kann auf den gesamten globalen Addressraum zugreifen, jedoch ist der Zugriff auf die lokale Partition effizienter. Das PGAS-Modell trifft keine Aussagen darüber, wie die Kommunikation zwischen den Rechenknoten realisiert wird.

Die seit 2004 von IBM entwickelte Programmiersprache X10 setzt eine Erweiterung des PGAS-Modells, das Asynchronous Partitioned Global Address Space (kurz *APGAS*) Modell [108], um. Der Name X10 ist aus der Motivation heraus entstanden, die Produktivität der parallelen Programmentwicklung um das Zehnfache zu steigern („times ten“). Das APGAS-Modell erweitert das PGAS-Modell um Asynchronität. Bezüglich Syntax und Semantik ähnelt X10 der Programmiersprache Java. X10 ist objektorientiert und nutzt einen Garbage Collector. Darüber hinaus bietet X10 Sprachkonstrukte zur Unterstützung von placeinterner und placeübergreifender Parallelität.

Über das Schlüsselwort `val` kann eine Variable als Konstante deklariert werden. Die Semantik folgt hierbei dem Schlüsselwort `final` in Java: Der Variable darf nur einmalig ein Wert zugewiesen werden. Ist der zugewiesene Wert eine Objektreferenz, so kann das entsprechende Objekt über Methoden weiterhin verändert werden.

In X10 werden Rechenressourcen mit ihrem assoziierten Speicher als *Place* bezeichnet. Pro Rechenknoten können mehrere Places gestartet werden, je-

doch teilen diese sich die Speicherpartition nicht, sodass ein Zugriff auf einen Place auf demselben Rechenknoten Kommunikation bedingt. Zu Programmstart wird über die Umgebungsvariablen `X10_NPLACES` und `X10_NTHREADS` festgelegt, mit wievielen Places die Berechnung gestartet wird und wieviele Threads maximal pro Place parallel ausgeführt werden dürfen. Die ausführbaren Einheiten eines Programms werden in X10 *Aktivitäten* genannt. Aktivtäten sind leichtgewichtige, ressourcenunabhängige Threads und somit zueinander asynchron.

Die Ausführung eines X10-Programms beginnt immer mit der Ausführung der `main`-Methode durch eine Aktivität auf Place `0`. Diese Aktivität wird als *Wurzelaktivität* bezeichnet. Sie ist die einzige Aktivität zu Beginn des Programmlaufs. Der Programmlauf endet, sobald die Wurzelaktivität ausgeführt wurde. Laufen zu diesem Zeitpunkt noch andere Aktivitäten, so werden diese abgebrochen.

Mit dem Schlüsselwort `async`, gefolgt von einem Codeblock, kann eine Aktivität erzeugt werden. Der Programmierer hat keine Kontrolle darüber, welcher Thread des aktuellen Places die Aktivität ausführen wird. Die Zuweisung der Aktivitäten auf die dem Place zur Verfügung stehenden Threads übernimmt die Laufzeitumgebung von X10. Hierbei findet eine placeinterne Lastenbalancierung statt. Ein Aufruf von `async` kehrt immer sofort zurück, unabhängig davon, ob die erzeugte Aktivität bereits gestartet wurde. Ferner bietet X10 über die Methode `Runtime.probe()` die Möglichkeit, die Ausführung der momentanen Aktivität zu unterbrechen, alle ausstehenden Aktivitäten auszuführen und dann zur unterbrochenen Aktivität zurückzukehren.

Um auf die Beendigung aller gestarteten Aktivitäten zu warten, wird das Schlüsselwort `finish`, gefolgt von einem Codeblock, verwendet. Der `finish`-Block wird erst dann verlassen, wenn alle innerhalb des Blocks gestarteten Aktivitäten beendet wurden. Dies gilt auch für verschachtelt gestartete Aktivitäten.

Quellcodeabbildung 3.1 zeigt, wie `finish` (Zeile 5) und `async` (Zeile 7) gemeinsam verwendet werden können, um auf die Beendigung asynchron gestarteter Aktivitäten zu warten. Das Programm startet 10 Aktivitäten in einer `for`-Schleife (Zeile 6). Nach Verlassen des `finish`-Blocks (Zeile 11) ist garantiert, dass alle 10 Aktivitäten ausgeführt wurden. Die Ausführungsreihenfolge der Aktivitäten ist hierbei nicht festgelegt.

Aktivitäten können mit dem Schlüsselwort `at(<Place>)`, gefolgt von einem Codeblock, synchron auf einen anderen Place verschoben werden. Der im

 doi:10.17170/kobra-2018122577

```
1  public class HelloActivities {
2
3    public static def main(val args:Rail[String]) {
4      Console.OUT.println("Say hello!");
5      finish {
6        for (i:Long in 0..9) {
7          async {
8            Console.OUT.println("Hello!");
9          }
10       }
11     }
12     Console.OUT.println("Thank you.");
13   }
14
15 }
```

Quellcodeabbildung 3.1: `HelloActivities.x10`

```
1  public class HelloPlaces {
2
3    public static def main(val args:Rail[String]) {
4      Console.OUT.println("Say hello!");
5      for (place in Place.places()) {
6        at (place) {
7          Console.OUT
8                 .println("[" + place + "]: Hello!");
9        }
10     }
11     Console.OUT.println("Thank you!");
12   }
13
14 }
```

Quellcodeabbildung 3.2: `HelloPlaces.x10`

```
1  public class HelloPlacesAsync {
2
3    public static def main(val args:Rail[String]) {
4      Console.OUT.println("Say hello!");
5      finish {
6        for (place in Place.places()) {
7          at (place) async {
8            Console.OUT
9                   .println("[" + place + "]: Hello!");
10         }
11       }
12     }
13     Console.OUT.println("Thank you!");
14   }
15
16 }
```

Quellcodeabbildung 3.3: `HelloPlacesAsync.x10`

Block definierte Codeabschnitt wird auf dem entfernten Place ausgeführt. Alle Werte der im `at`-Block verwendeten Variablen werden an den entfernten Place gesendet. Diese Variablen müssen als `val` definiert sein. Erst nach Ausführung des Blocks kehrt die Ausführung auf den ursprünglichen Place zurück. Quellcodeabbildung 3.2 zeigt, wie eine Aktivität einmal auf jeden Place verschoben werden kann. Da `at` (Zeile 6) ein synchrones Konstrukt ist, kann auf einen `finish`-Block verzichtet werden. Der Aufruf `Place.places()` (Zeile 5) liefert eine Liste aller Places zurück. Die Places werden in der Reihenfolge der Place-IDs durch die Aktivität besucht.

Die Schlüsselworte `async` und `at` können gemeinsam genutzt werden, um auf einem entfernten Place eine neue Aktivität zu starten. Quellcodeabbildung 3.3 zeigt ein Beispiel, in dem `finish` (Zeile 5), `at` und `async` (beides Zeile 7) gemeinsam genutzt werden, um pro Place eine Aktivität zu starten. Im Unterschied zu dem Programm aus Quellcodeabbildung 3.2 ist bei diesem Programm nicht gewährleistet, dass die Ausgabe in geordneter Reihenfolge erfolgt, da das Programm Asynchronität verwendet.

Neben `at(<Place>) async {...}` kann auch `async at (<Place>) {...}` genutzt werden. Die beiden Varianten unterscheiden sich jedoch geringfügig in ihrer Semantik:

 doi:10.17170/kobra-2018122577

- `at(<Place>) async {...}` wechselt zuerst auf den entfernten Place und startet dort eine neue Aktivität.
- `async at (<Place>) {...}` startet eine neue Aktivität auf dem lokalen Place. Sobald sie ausgeführt wird, wechselt sie auf den entfernten Place.

Wird `async at (<Place>) {...}` zu einem Zeitpunkt ausgeführt, zu dem keine lokale Rechenresourcen zur Verfügung stehen, um die erstellte Aktivität auszuführen, so wird die Ausführung der Aktivität – und somit der Wechsel auf den entfernten Place – verzögert, bis die Aktivität ausgeführt werden kann. Diese Verzögerung tritt bei Verwendung von `at(<Place>) async {...}` nicht ein.

Arrays werden in X10 über die Klasse `Rail[Type]` dargestellt. Hierbei ist `Type` der Elementtyp des Arrays. Darüber hinaus bietet X10 die Klasse `DistArray[Type]` an, um ein Array über mehrere Places zu verteilen. Die Verteilung der Daten wird über Konstruktorparameter gesteuert.

X10 bietet über die Klasse `PlaceLocalHandle[<Type>]` eine Möglichkeit, auf jedem Place eine Instanz des gegebenen Typs zu initialisieren. Wird ein `PlaceLocalHandle` auf einem Place ausgewertet, so liefert dieser die auf dem Place gespeicherte Instanz zurück. Hierdurch kann jeder Place eine andere lokale Instanz des gleichen Typs abspeichern.

Wird eine Instanz einer Klasse nur auf einem einzigen Place benötigt, bietet X10 hierfür die Klasse `GlobalRef[<Type>]` an. Die Referenz kann an andere Places gesendet werden. Ein Zugriff auf die durch die `GlobalRef` gespeicherten Daten ist nur auf dem Ursprungs-Place möglich. Der Ursprungs-Place ist im Attribut `home` der `GlobalRef` gespeichert.

Places können in `Team`s organisiert werden. Auf einer Instanz von `Team` können kollektive Operationen – beispielsweise `allreduce(...)`, `bcast(...)`, `gather(...)` und `scatter(...)` – ausgeführt werden.

Die Fehlertoleranz von X10 wird durch das Setzen der Umgebungsvariable `X10_RESILIENT_MODE` auf den Wert `1` vor Programmstart aktiviert. Ein Ausfall von Place `0` wird nicht unterstützt und führt zu sofortigem Programmabbruch. Zur Behandlung von Placeausfällen bietet X10 drei Mechanismen an:

- Wird auf einen ausgefallenen Place mittels `at(...)` zugegriffen, so wird eine `DeadPlaceException` ausgelöst.
- Die Lebendigkeit eines spezifischen Places kann über eine Methode `Place.isDead(<Place>)` abgefragt werden. Ebenso kann die Anzahl der seit Programmstart ausgefallenen Places über den Aufruf `Place.numDead()` abgefragt werden.

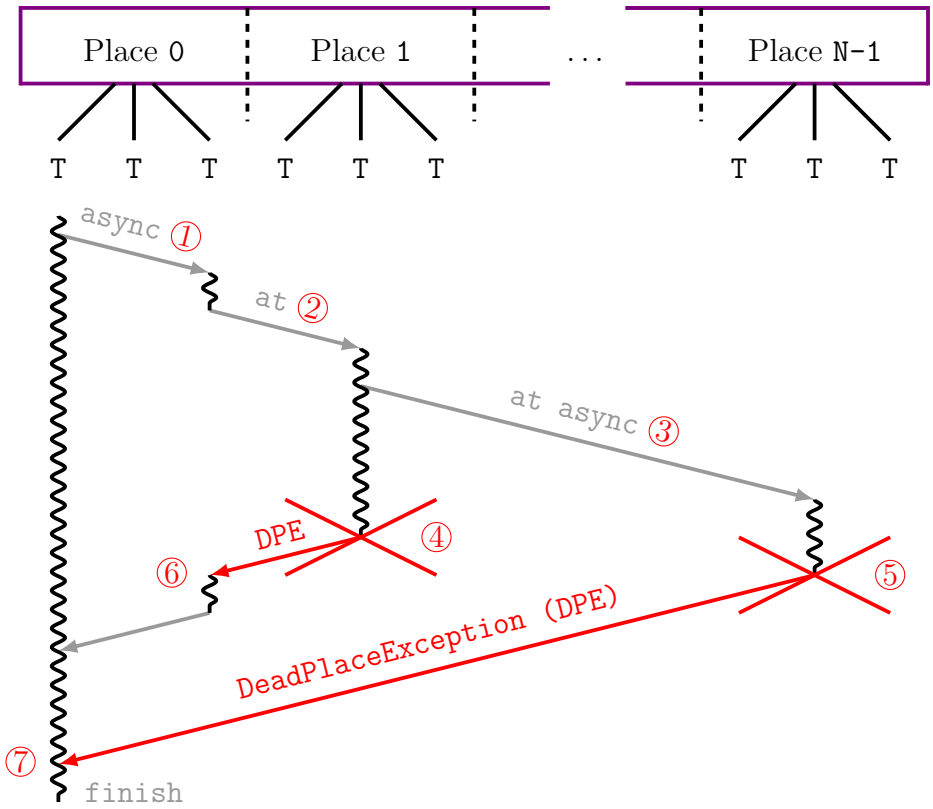


Abbildung 3.1.: Synchronisationspunkte zur Behandlung auftretender `DeadPlaceExceptions` bei Verwendung von `async` und `at`

- Auf jedem Place kann eine Methode als `PlaceRemovedHandler` registriert werden. Diese Methode wird von der X10-Laufzeitumgebung automatisch ausgeführt, wenn ein Place ausgefallen ist.

Wir besprechen nachfolgend diese drei Mechanismen, wie sie zu verwenden sind und die daraus folgenden Einsatzgebiete für jeden Mechanismus.

Tritt in X10 eine `Exception` innerhalb einer Aktivität auf, so kann diese nicht von der startenden Aktivität gefangen werden. Dies liegt daran, dass es keinen Synchronisationspunkt gibt, an dem die `Exception` behandelt werden könnte. Abbildung 3.1 zeigt ein Beispiel mit zwei Placeausfällen bei der Verwendung von `async` und `at`. Place `0` greift über `async` ①, gefolgt von einem `at` ② auf Place `1` zu, welcher wiederum mit einem `at async` ③ auf Place `N-1` zugreift. Da Place `1` ausfällt ④, bevor er die durch das `at` ② verschobene Aktivität

doi:10.17170/kobra-2018122577

von Place 0 beenden konnte, kann die DeadPlaceException nach dem at auf Seiten von Place 0 ⑥ behandelt werden. Fällt später Place N-1 aus ⑤, so kann die DeadPlaceException nicht an das at von Place 1 weitergeleitet werden. Zum einen wurde nach dem Placewechsel eine neue Aktivität durch das async gestartet ③, zum anderen ist Place 1 ausgefallen. Diese Ausnahme kann auch nicht an das at von Place 0 ⑥ weitergeleitet werden, da die Aktivität auf Place N-1 durch das at async von Place 1 ③ asynchron zu dem at auf Place 0 ② ausgeführt wird. Somit muss die Exception an das nächste finish weitergeleitet werden ⑦, falls ein solches existiert. Existiert kein solches finish, so kann die Exception nicht behandelt werden und geht stillschweigend verloren. Wir sehen somit, dass zum Abfangen von Exceptions ein Synchronisationspunkt benötigt wird. Synchronisationspunkte können durch vorangegangene Placeausfälle weggefallen sein.

Die Verwendung von Place.isDead(<Place>) erlaubt die spezifische Lebendigkeitsüberprüfung eines Places. Für eine zeitnahe Reaktion auf Ausfälle muss die Überprüfung in regelmäßigen Abständen erfolgen.

Wird die Mechanik des PlaceRemovedHandler zur Fehlerbehandlung verwendet, so wird der als Methode hinterlegte Handler bei einem Placeausfall als neue Aktivität ausgeführt. Diese Aktivität wird von der X10-Laufzeitumgebung gestartet und bindet sich nicht an im Programm vorhandene finish-Blöcke. Der PlaceRemovedHandler kann nur in Verbindung mit Managed X10 verwendet werden.

Für die Unterstützung von Elastizität bietet X10 einen PlaceAddedHandler an. Dieser folgt der Funktionsweise des PlaceRemovedHandler und unterliegt denselben Beschränkungen.

Die Methoden Place.places() und Place.numPlaces() berücksichtigen elastisch hinzugefügte Places. Bei diesen Methoden ist zu beachten, dass ausgefallene Places weiterhin berücksichtigt werden. Werden einem Programmlauf neue Places hinzugefügt, so kann dies durch einen regelmäßigen Aufruf von Place.places() oder Place.numPlaces() erkannt werden.

3.2. GLB

Wie in Abschnitt 1.1 erwähnt, ist GLB ein in X10 implementiertes Framework zur Verwaltung eines verteilten Taskpoolols, welches kooperatives Stehlen zur Lastenbalancierung nutzt. GLB verwendet das folgende Taskmodell:

- Die Ausführung jedes Tasks berechnet ein Teilergebnis.
- Durch die Ausführung von Tasks können zur Laufzeit neue Tasks entstehen.
- Das Gesamtergebnis der Berechnung kann durch Reduktion mit einem assoziativen und kommutativen Operator über alle Teilergebnisse berechnet werden.
- Tasks verursachen keine Seiteneffekte.

Darüber hinaus fordert das GLB-Framework, dass zu jedem Zeitpunkt höchstens eine Aktivität pro Place ausgeführt wird. Dies vereinfacht die Implementierung des Frameworks, da somit placeinterne Parallelität ausgeschlossen wird. Anstatt dessen können pro Rechenknoten mehrere Places gestartet werden.

GLB setzt voraus, dass eine Datenstruktur zur Speicherung der lokalen Taskpools durch den Nutzer implementiert wird. Diese Datenstruktur muss die folgenden Operationen unterstützen:

- `process(n:Int)`: arbeitet bis zu `n` Tasks aus dem Taskpool ab, enthält also Code der Anwendung. Die Methode gibt `true` zurück, wenn nach der Abarbeitung der `n` Tasks der Taskpool nicht leer ist und `false` sonst.
- `split()`: Löst eine Menge von Tasks aus dem Taskpool heraus und gibt diese als `TaskBag` (siehe unten) zurück. Hierbei ist es der Implementierung überlassen, wieviele und welche Tasks herausgelöst werden. Es muss jedoch mindestens ein Task im lokalen Pool verbleiben.
- `merge(bag:TaskBag)`: Fügt die in `bag` enthaltene Beute in den lokalen Taskpool ein. Es ist der Implementierung überlassen, wie die Beute in den Taskpool eingefügt wird.
- `getResult()`: gibt das Ergebnis des Workers als `GLBResult` zurück.

Die Klassen `TaskBag` sowie `GLBResult[T]` sind Hilfsdatenstrukturen, die ebenfalls vom Benutzer zu implementieren sind. Die Klasse `TaskBag` wird zum Übertragen von Beute während eines Stehlvorgangs verwendet. Sie enthält eine Menge von Tasks, jedoch kein Teilergebnis. Sie verfügt über eine Methode `size()`, welche die Anzahl an Tasks im `TaskBag` zurückgibt. Die Klasse

 doi:10.17170/kobra-2018122577

```
1 do {
2   while(queue.process(n)) {
3     Runtime.probe();
4     distribute();
5     reject();
6   }
7 } while (steal(st));
```

Quellcodeabbildung 3.4: Hauptarbeitsschleife eines Workers (siehe [40])

`GLBResult[T]` stellt das berechnete Teilergebnis eines Workers in Form eines `Rail`s dar, wobei der Elementtyp `T` des Arrays durch den Benutzer definiert wird. In GLB wurde ein `Rail` gewählt, da das Ergebnis einer Berechnung mehrere Werte sein können. Ein Beispiel hierfür ist die Betweenness Centrality [38] auf Graphen, die für jeden Knoten des Graphen eine Kenngröße berechnet.

Sprechen wir fortan von einem Taskpool, so meinen wir den lokalen Taskpool eines Workers.

Das GLB-Framework verwendet zur Realisierung sämtlicher Kommunikation `at(<Place>) async { ... }`. Wir bezeichnen so gestartete Aktivitäten auf entfernten Places nachfolgend als *Nachricht*.

Quellcodeabbildung 3.4 zeigt die Hauptarbeitsschleife, die von jedem Worker ausgeführt wird. Nachfolgend erklären wir die Funktionsweise von GLB anhand dieser Hauptarbeitsschleife. Die Variable `queue` stellt den Taskpool des Workers dar. Der GLB-Parameter `n` bestimmt, wie viele Tasks bearbeitet werden, bevor die Taskprozessierung unterbrochen wird, um Nachrichten zu empfangen und Stehlanfragen zu bearbeiten. Die Aktivität, welche die Hauptarbeitsschleife ausführt, nennen wir *Workeraktivität*. Führt ein Worker die Hauptarbeitsschleife aus, so ist er *aktiv*, sonst *inaktiv*. Der Taskpool eines inaktiven Workers ist normalerweise leer.

Wir beginnen die Betrachtung bei der `while`-Bedingung in Zeile 2. Durch den Aufruf von `queue.process(n)` werden bis zu `n` Tasks des Taskppol abgearbeitet. Enthält der Taskpool nach der Ausführung von `n` Tasks noch weitere Tasks, so werden über die Aufrufe `Runtime.probe()` (Zeile 3) Nachrichten empfangen. Eingegangene Stehlanfragen werden aufgezeichnet und über `distribute()` (Zeile 4) bzw. `reject()` (Zeile 5) bearbeitet. Auf die genaue Funktionsweise diesesr Methoden gehen wir weiter unten ein.

Bei einigen Problemen ist die Ausführungszeit eines einzelnen Tasks so hoch, dass selbst mit `n=1` zu selten `Runtime.probe()` aufgerufen wird. Hierdurch entstehen hohe Latenzen in der Bearbeitung von Stehlanfragen und infolge dessen eine ungleichmäßige Lastenbalancierung und eine hohe Ausführungszeit. GLB erlaubt daher, dass die Methode `queue.process(n)` die Abarbeitung eines Tasks unterbricht, um in die Hauptarbeitsschleife zurückzukehren, Nachrichten zu empfangen und Stehlanfragen zu beantworten. Als Alternative bietet GLB eine Methode `yield()` an, die innerhalb von `queue.process(...)` aufgerufen werden kann. Diese Methode unterbricht die Berechnung, um ausstehende Nachrichten zu empfangen und zu bearbeiten. In beiden Fällen muss die Implementierung von `queue.split()` sicherstellen, dass ein partiell bearbeiteter Task nicht aus dem Taskpool entfernt wird. Ein Beispiel für ein solches Problem ist die Betweenness Centrality Bewertung eines jeden Knoten in einem Graphen [38].

Ist der Taskpool nach Ausführung von bis zu `n` Tasks leer, so beginnt ein Worker in Zeile 7 Arbeit von anderen Workern zu stehlen. Der Stehlvorgang wird in zwei Phasen unterteilt: Ein Worker `T` (**T**hief, deutsch Dieb) versucht zuerst, bei `w` zufälligen Workern Tasks zu stehlen, wobei `w` ebenfalls ein GLB-Parameter ist. Hierzu sendet `T` eine Nachricht an das erste, zufällig gewählte Opfer `V` (**V**ictim, deutsch Opfer) und wartet dann mittels busy waiting auf eine Antwort. `T` ruft hierbei `Runtime.probe()` auf, um weiterhin Nachrichten zu empfangen, wird aber aufgezeichnete Nachrichten erst abarbeiten, sobald er in die Hauptarbeitsschleife zurückkehrt. Sollten alle `w` zufälligen Stehlversuche von `T` erfolglos sein, so beginnt `T`, bei seinen *Lifeline-Buddies* zu stehlen. Hierzu werden zu Programmstart alle Worker in einem Graphen angeordnet, der einem gerichteten zyklischen Hypercubus der Dimensionalität `z` mit `l` Knoten in jeder Dimension ähnelt. Die `z` Nachfolger eines Workers bilden dabei seine Lifeline-Buddies. Der Wert `l` ist ein GLB-Parameter, die Dimensionalität `z` wird aus `l` automatisch berechnet. Abbildung 3.2 zeigt einen Lifeline-Graphen der Dimensionalität $z = 2$, mit $l = 4$ und 10 Workern. Eine Kante von einem Place `P` zu einem Place `Q` bedeutet, dass `Q` ein Lifeline-Buddy von `P` ist.

Wir nennen Stehlanfragen von Lifeline-Buddies nachfolgend *Lifeline-Anfragen*. Ein zufällig gewähltes Opfer weist eine Stehlanfrage ab, wenn es keine Tasks hat, die es teilen kann. Zusätzlich speichert ein Lifeline-Buddy erfolglose Stehlanfragen ab. Erhält dieser Lifeline-Buddy zu einem späteren Zeitpunkt Tasks, so teilt er diese mit `T`. Ist eine Lifeline-Anfrage von `T` zu seinem Lifeline-Buddy `V` (noch) nicht mit Tasks beantwortet worden, so be-

 doi:10.17170/kobra-2018122577

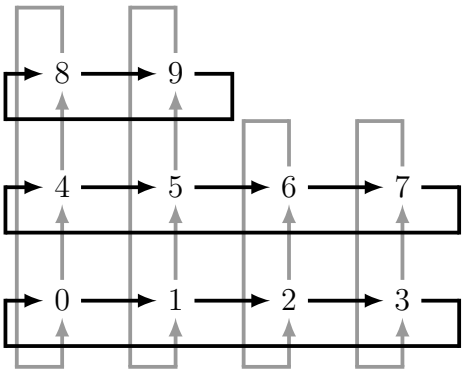


Abbildung 3.2.: Lifeline-Graph der Dimensionalität `z` = 2, mit `l` = 4 und 10 Knoten.

zeichnen wir die entsprechende *Lifeline* als *offen*, andernfalls als *geschlossen*. `T` wird nur eine Lifeline-Anfrage an einen Lifeline-Buddy senden, wenn die entsprechende Lifeline geschlossen ist.

Führt das Opfer der Stehlanfrage zu dem Zeitpunkt, zu dem die Nachricht bei ihm eingeht, die Hauptarbeitsschleife aus, so wird die Stehlanfrage erst mit dem nächsten Aufruf von `Runtime.probe()` in Zeile 3 bzw. innerhalbe der Methode `steal(...)` in Zeile 7, wenn es selber gerade auf die Beantwortung einer gesendeten Stehlanfrage wartet, empfangen. Durch die Ausführung der Stehlanfrage wird geprüft, ob der Taskpool von `V` leer ist. Hat `V` keine Arbeit, so teilt er dies `T` mit einer entsprechenden Nachricht mit und `T` wird einen neuen Stehlvorgang beim nächsten Opfer starten. Wenn `V` jedoch Tasks im Taskpool hat, so wird `T` als Dieb in eine Liste von Dieben aufgenommen und die Stehlanfrage mit dem nächsten Aufruf von `distribute()` in Zeile 4 oder `reject()` in Zeile 5 beantwortet. Die Methode `distribute()` ruft iterativ für jeden aufgezeichneten Dieb `queue.split()` auf, um die Beute für den entsprechenden Dieb zu bestimmen und sendet diese mit einer Nachricht an den Dieb. Hierbei werden zuerst neu eingegangene Stehlanfragen und danach vorher schon einmal abgelehnte Stehlanfragen von Lifeline-Buddies bearbeitet. Sollte während dieser Ausführung der Taskpool des Opfers so klein werden, dass das Opfer keine Tasks mehr abgeben kann, so endet die Methode `distribute()` und alle noch ausstehenden Stehlanfragen werden durch den Aufruf der Methode `reject()` in Zeile 5 abgelehnt.

Erhält ein Dieb Beute, so fügt er diese mit Hilfe der Methode `queue.merge(...)` in seinen Taskpool ein.

Wenn `w` zufällige Stehlanfragen in Folge sowie alle Lifeline-Anfragen eines Workers erfolglos waren, so verlässt der Worker die Hauptarbeitsschleife und wird inaktiv. Die auf dem Worker gespeicherten Daten bleiben jedoch erhalten und er kann nach Reaktivierung wieder auf diese zugreifen. Der Worker kann nur dann wieder aktiv werden, wenn er Tasks von einem Lifeline-Buddy erhält. Nach dem Einfügen der Tasks führt der Worker dann wieder die Hauptarbeitsschleife aus.

Die Berechnung endet, sobald die Taskpools aller Worker leer sind. In diesem Zustand sind alle Worker inaktiv. Das GLB-Framework erkennt diesen Zustand dadurch, dass alle initialen Workeraktivitäten in einem äußeren `finish`-Block gestartet werden, der dann verlassen wird. Das Endergebnis wird dann mittels Reduktion über die Teilergebnisse aller Worker berechnet. Hierzu wird die Methode `getResult()` der Taskpool-Datenstruktur verwendet.

Es bleibt zu betrachten, wie eine Berechnung beginnt. GLB verwendet zur Initialisierung der Worker (einer pro Place) den von X10 bereitgestellten `PlaceLocalHandle`. Hierdurch wird unter anderem der initiale Taskpool jedes Workers angelegt. GLB unterscheidet zwischen zwei möglichen Startvarianten:

1. Die Berechnung startet mit einem Task im Taskpool von Place `0`, oder
2. die Berechnung startet mit mehreren Tasks.

Den ersten Fall nennen wir im Weiteren *dynamischen Start*. Bei dieser Startvariante hat zu Beginn der Berechnung der Worker auf Place `0` den einzigen ausführbaren Task. Die Lifelines aller anderen Worker sind offen. Hierdurch verteilt Worker `0` Tasks an seine Lifeline-Buddies, nachdem er die ersten `n` Tasks, die aus dem ursprünglichen Task entstanden sind, abgearbeitet hat. Seine Lifeline-Buddies reichen wiederum Tasks an ihre Lifeline-Buddies weiter, nachdem diese `n` Tasks abgearbeitet haben. Dieser Vorgang wiederholt sich. Somit verteilt sich die Arbeit dynamisch über den Lifeline-Graphen.

Startet das Programm mit mehr als einem Task, so wird ein *statischer Start* durchgeführt: Jeder Worker bekommt zu Beginn eine Menge an Tasks zugewiesen und beginnt, diese zu prozessieren. Die Verteilung der initialen Tasks auf die Worker wird durch eine benutzerdefinierte Methode gesteuert. Bei dieser Startvariante sind alle Lifelines zu Beginn der Berechnung geschlossen.

 doi:10.17170/kobra-2018122577

4. Fehlertoleranz

Wir stellen nachfolgend den Fehlertoleranzalgorithmus vor. Die Entwicklung des Algorithmus erfolgte iterativ. Teile des Algorithmus wurden während der Promotion in [39, 40, 41, 42, 43, 45] veröffentlicht. In dieser Arbeit erfolgt erstmalig eine ganzheitliche Beschreibung aller Aspekte des Algorithmus.

Wir stellen zuerst die Grundidee des Algorithmus in Abschnitt 4.1 dar.

Der Algorithmus umfasst drei Grundkonzepte: Die Datensicherung, die Erkennung von Ausfällen und die Wiederherstellung nach aufgetretenen Ausfällen. In Abschnitt 4.2 wird die Funktionsweise der Datensicherung beschrieben unter der Voraussetzung, dass keine Fehler auftreten. Insbesondere besprechen wir die verschiedenen Aspekte zum Schreiben regelmäßiger und stehlinduzierter Backups.

Die Erkennung von Ausfällen diskutieren wir in Abschnitt 4.3. Wir zeigen drei verschiedene Techniken auf, mit denen wir zeitnah über Placeausfälle informiert werden.

Abschnitt 4.4 führt das Wiederherstellungsprotokoll zur Behandlung von Placeausfällen ein. Wir betrachten zuerst Situationen, in denen ein Place ausfällt. Anschließend betrachten wir Situationen, in denen mehrere Places vor der Wiederherstellung eines konsistenten Zustandes ausfallen.

4.1. Kernidee

Wir übernehmen alle Voraussetzungen von GLB, insbesondere:

- die Bedingung, dass pro Place nur ein Thread und höchstens eine Workeraktivität läuft,
- das Taskmodell, sowie
- das Lifeline-Schema und die daran gekoppelte Terminierungserkennung.

Unsere Implementierung übernimmt die Einschränkung von Resilient X10, dass Place `0` nicht ausfallen darf. Im Falle eines Ausfalls von Place `0` bricht das Programm mit einer Fehlermeldung ab.

Wie wir in Abschnitt 1.1 bereits besprochen haben, realisieren wir die Datensicherung durch Checkpointing der relevanten Daten in den Hauptspeicher eines anderen Workers. Die Verteilung der Backup-Daten folgt dem Schema des Backup-Rings. Jeder Worker speichert sich seinen Vorgänger und Nachfolger im Backup-Ring in Variablen `Forth` und `Back`.

Die Backup-Daten, die in den Speicher des *Backup-Places* `Back` repliziert werden, bestehen aus dem lokalen Pool inklusive Teilergebnis sowie der `Forth`-Variable des Workers. Abhängig vom Zustand des Workers können weitere Informationen auf dem Backup-Place gespeichert werden. Wir führen diese an den entsprechenden Stellen in Abschnitt 4.2 ein.

Ein Entwurfsziel des Fehlertoleranzalgorithmus ist eine gleichmäßige Redundanz. Das heißt, dass alle für die Berechnung relevanten Daten möglichst auf genau zwei Workern vorgehalten werden. Jeder Worker speichert genau ein Backup eines anderen Workers ab. Der Algorithmus ist so entworfen, dass Zeiträume, in denen Daten auf einem oder mehr als zwei Workern vorgehalten werden, möglichst kurz gehalten sind.

Fällt ein Worker während der Berechnung aus, so übernimmt sein Backup-Place dessen lokales Teilergebnis und fügt die Tasks aus dem Backup des ausgefallenen Workers in seinen Taskpool ein. Ein Neustart des Programms ist somit nicht erforderlich.

Wir verwenden ein aktorenartiges Modell zur Realisierung des Fehlertoleranzalgorithmus. Hierbei ist jeder Worker eine Entität, die Tasks aus ihrem lokalen Pool abarbeitet. Treten Ereignisse auf, so verändert ein Worker sein Verhalten, um auf diese zu reagieren. Ereignisse können entweder lokal generiert werden (beispielsweise die Feststellung, dass der lokale Pool leer ist) oder als Nachrichten von anderen Workern empfangen werden (beispielsweise der Erhalt einer Stehlanfrage). Reden wir nachfolgend über den Erhalt einer Nachricht, so meinen wir damit den Zeitpunkt, zu dem die Nachricht auf Seiten des Empfängers ausgeführt wird (beim nächsten Aufruf von `Runtime.probe()`).

Das GLB-Framework, welches wir in Abschnitt 3.2 besprochen haben, kennt drei Ereignisse:

- die Feststellung, dass der lokale Pool eines Workers leer ist,
- das Eingehen einer Stehlanfrage und

 doi:10.17170/kobra-2018122577

- die Antwort auf eine vorangegangene zufällige oder Lifeline-basierte Stehlanfrage.

Für den fehlertoleranten Algorithmus führen wir weitere Ereignisse ein, beispielsweise für das Senden und Empfangen von Backups, das fehlertolerante Stehlen sowie für das Überwachen und Wiederherstellen von Workern. Reagiert ein Worker auf ein Ereignis, so können hierdurch weitere Ereignisse ausgelöst werden. Beispielsweise kann das Ereignis, dass der lokale Taskpool leer ist, dazu führen, dass einem Opfer eine Stehlanfrage gesendet wird. Wir halten den Worker reaktionsbereit, indem er weiter rechnet, während er ausstehende Ereignisse von anderen Workern erwartet. Hierdurch lassen sich durch Kommunikation verursachte Latenzen verstecken.

Die Reaktion auf ein Ereignis kann entweder *sofort* bei Erhalt des Ereignisses erfolgen oder *aufgezeichnet* werden. Zum Aufzeichnen von Nachrichten werden alle notwendigen Informationen in einer geeigneten Datenstruktur festgehalten und die Reaktion erfolgt zu einem späteren Zeitpunkt. Das Aufzeichnen von Ereignissen bietet den Vorteil, dass die aufgezeichneten Ereignisse in einer wohldefinierten Reihenfolge bearbeitet werden können. Hierdurch können bestimmte Ereignisse bevorzugt behandelt werden.

Wir fordern von der genutzten Kommunikationsschicht lediglich, dass Nachrichten zuverlässig, aber nicht zwingend in der gesendeten Reihenfolge übertragen werden. Das Senden einiger Nachrichten von einem Sender zu einem Empfänger setzt voraus, dass bestimmte andere Nachrichten vormals vom Empfänger ausgeführt wurden. Beispielsweise darf ein Worker erst dann ein neues Backup senden, wenn sein Backup-Place den Erhalt des vorigen Backups bestätigt hat. Andernfalls könnten die Backup-Nachrichten in umgekehrter Reihenfolge vom Backup-Place empfangen werden und somit zu Dateninkonsistenzen führen. An den notwendigen Stellen wird der Erhalt von Nachrichten durch das Senden von Bestätigungsnachrichten vom Empfänger zurück an den Sender gewährleistet. Die Namen dieser Nachrichten enden typischerweise mit dem Kürzel `ack` (`acknowledgement`). Kann eine Nachricht zu einem Zeitpunkt aufgrund einer noch ausstehenden Nachricht nicht bearbeitet werden, so wird die Bearbeitung der Nachricht bis zum nächsten Aufruf von `processRecorded(...)` *verschoben*.

Der Umbau auf das aktorenartige Schema führte zu einer Veränderung der Hauptarbeitsschleife. Quellcodeabbildung 4.1 zeigt eine vereinfachte Form der

```
1 while ((queue.size() > 0) && !stealsFailed) {
2     if (queue.size() > 0) {
3         processUpToN();
4     }
5     Runtime.probe();
6     processRecorded();
7 }
```

Quellcodeabbildung 4.1: Vereinfachte Darstellung der Hauptarbeitsscheife eines fehlertoleranten Workers (siehe [45])

neuen Hauptarbeitsschleife. Mit dem Aufruf von `Runtime.probe()` in Zeile 5 werden alle ausstehenden Nachrichten, wie in Abschnitt 3.1 besprochen, ausgeführt. Die Verarbeitung aller aufgezeichneten und verschobenen Ereignisse erfolgt durch den Aufruf `processRecorded()` in Zeile 6. Ferner ist diese Methode für die Generierung lokaler Ereignisse verantwortlich. Eine Auflistung aller Ereignisse und der zugehörigen Reaktionen findet sich in Tabelle 4.1. Aktionen, die in der Tabelle *kursiv* dargestellt sind, werden sofort ausgeführt. Nicht-kursiv geschriebene Aktionen werden aufgezeichnet und somit beim nächsten Aufruf von `processRecorded()` ausgeführt. Für sofort ausgeführte Aktionen gibt es keine wohlgeordnete Ausführungsreihenfolge. Aufgezeichnete Nachrichten werden gemäß ihrer Priorität ausgeführt. Die Ereignisse in Tabelle 4.1 sind gemäß ihrer Priorität sortiert. Die Tabelle umfasst alle Nachrichten, insbesondere auch jene, die erst in Kapitel 5 behandelt werden. Die Spalten **NM** (**N**otfall**M**odus) und **TK** (**T**imeout-**K**ontrolle) geben an, ob die Nachrichten im Notfallmodus ausgeführt werden bzw. ob diese Nachrichten über eine Timeout-Kontrolle überwacht werden. Wir gehen in den Abschnitten 4.3 und 4.4 näher auf diese Spalten ein. Zusätzlich sind in Tabelle 4.2 alle Nachrichten (mit den zugehörigen Zuständen) aufgelistet, die verschoben werden können.

Führt ein Place `P` keine Workeraktivität aus, so führt er auch `processRecorded()` nicht aus. Somit kann `P` keine aufgezeichneten Nachrichten beantworten. Wir diskutieren und lösen dieses Problem in Abschnitt 4.3.

 doi:10.17170/kobra-2018122577

Nachr. / Ereignis	Bedeutung	Reaktionen auf Seiten des Empfängers	NM	TK
`REGack`	reguläres Backup abgeschlossen	*Zustand aktualisieren*	x	(-)
`STLack`	Stehl-Backup abgeschlossen	`give` *Tasks an alle Diebe senden*	x	(-)
`TOack`	Übernahme-Backup abgeschlossen, siehe Abbildung 4.2	*sende* `RSTack`	x	x
`IAack`	Einweihungs-Backup abgeschlossen, siehe Abbildung 4.2	*setze* `Back` *auf neuen Wert*	x	-
`RSTack`	handshaking, siehe Abbildung 4.2	*sende* `IAreq`	x	x
`BTack`	handshaking, siehe Abbildung 4.1	*sende* `Tend`; *wenn letzter Dieb, sende* `BVend`	x	x
`Tend`	Stehlvorgang für Dieb beendet	*aktualisiere* `openTends`	x	x
`BVend`	Transaktion auf Seiten des Diebes abgeschlossen	*aktualisiere Zustand*	x	x
`victimLink`	Sende V's Link, siehe Abbildung 4.1	*speichere Link, sende* `BTack`	x	-
`delOpen`	`Back(T)` hat Kopie übernommen, siehe Abbildung 4.1	*lösche alle Tasks bis zu* `tan` *aus* `Open(T)`	x	-
`linkResolve`	`Back(T)` fordert auf `V` gespeicherte Tasks an	*sende* `linkTasks`	x	-
`linkTasks`	sende angefragte Tasks	*wenn* (R3) *erreicht, sende* `TOreq`	x	x
`GOTcheck`	`Back(V)` prüft, ob Tasks bei `T` eingetroffen sind, siehe Abbildung 4.2	*sende* `GOTok`, *breche Programm ab* oder warte	x	-
`GOTok`	Tasks sind bei `T` eingegangen, siehe Abbildung 4.2	*wenn* (R3) *erreicht, sende* `TOreq`	x	x
`deathNotice`	Backup-Place ausgefallen	*sende* `restore`, *markiere* `Back` *als ungültig*	x	-
`REGreq`	regelmäßiges Backup empfangen	*ersetze Backup-Daten, sende* `REGack`	x	-
`STLreq`	Stehl-Backup empfangen, siehe Abbildung 4.1	*ersetze Backup-Daten, sende* `STLack`	x	-
`TOreq`	Übernahme-Backup empfangen, siehe Abbildung 4.2	*ersetze Backup-Daten, sende* `TOack`	x	-
`IAreq`	Einweihungs-Backup erhalten, siehe Abbildung 4.2	*Ersetze Backup-Daten, sende* `IAack`	x	x
`monitor`	Ghost-Aktivität, siehe Abschnitt 4.3	*starte Ghost, sende* `monitor` *zu* `Back`	x	-
`startGrowth`	Einbindung neuer Places, siehe Abbildung 5.1	*Beginne Ausführung von Phase 2 des Elastizitätsprotokolls*	-	x
`migrated`	Place `N` hat das Backup von Place `N-1` übernommen	*Sende* `referral`, *falls ein Placeausfall gemeldet wurde*	x	-
`referral`	Place `N` ist für die Wiederherstellung des ausgefallenen Places verantwortlich	*sende* `restore(...)` *an Place* `N`	x	-
`extendDone`	Phase 2 des Elastizitätprotokolls ist abgeschlossen, siehe Abbildung 5.1.	*Sende* `startup` *an alle Places*	-	x
`startup`	Lifeline-Graph Rekonstruktion und Phase 3 des Elastizitätprotokolls, siehe Abschnitt 4.4.3 und Abbildung 5.1	*Speichere Lifline-Graphen ab, starte Workeraktivität falls inaktiv*	-	-
`restore`	Wiederherstellung angefragt, siehe Abbildung 4.2	sende `linkResolve`/ `GOTcheck`, prüfe Ring	x	-
`give`	Taskzustellung, siehe Abbildung 4.1	*speichere* `tan` *in* `tanGOT`, füge Tasks in lokalen Pool ein, sende `victimLink`	-	x
isDead(`Back`)	regelmäßiger `isDead()` Aufruf liefert `true`	sende `restore(...)`	-	n/a
`trySteal`	Stehl-Anfrage erhalten, siehe Abbildung 4.1	*sende* `noTasks` oder bilde eine Transaktion und starte mit Abbildung 4.1	-	-
timeout	Zeitüberschreitung erkannt	Prüfe Lebendigkeit, sende `deathNotice`, Spezifisches (siehe Abschnitt 4.3)	x	n/a
k Iterationen vorbei	Backup-Intervall vorbei	sende `REGreq`	-	-
`noTasks`	Angefragtes Opfer hat keine Tasks	sende nächstes `trySteal` oder setze `stealsFailed`	-	x
lokaler Pool leer	`process(n)` hat `false` zurückgeliefert	sende erstes `trySteal`	-	n/a

Tabelle 4.1.: Ereignisse und die zugehörigen Reaktionen (siehe [45]). Die Reihenfolge in der Tabelle gibt die Ausführungsreihenfolge bei zeitgleichem Eintreffen an. In der ersten Spalte sind Nachrichten in `TrueType` geschrieben. Lokal generierte Ereignisse sind in normaler Schriftart geschrieben. *Rekursiv* geschriebene Reaktionen werden sofort ausgeführt, normal geschriebene Reaktionen werden aufgezeichnet. Abkürzungen: NM = Notfallmodus, TK = Timeout-Kontrolle, n/a = nicht anwendbar, (-) = wird nur im Notfallmodus ausgeführt.

doi:10.17170/kobra-201812257

Aktion	Zustand	Reaktion
Versand `(REG|STL|TO|IA)req`	vorangegangenes `(REG|STL|TO|IA)ack` steht aus	Verschiebe Verstand bis `(REG|STL|TO|IA)ack` eingetroffen ist.
	Zwischen (T1) und (T4) in Abbildung 4.1	Verschiebe Versand bis (T4) in Abbildung 4.1 erreicht wurde.
Abarbeitung aufgezeichneter Diebe	zwischen (T1) und (T4) in Abbildung 4.1	Verschiebe Abarbeitung bis (T4) in Abbildung 4.1 erreicht wurde.
	(V1) und (V5) in Abbildung 4.1	Verschiebe die Abarbeitung bis (V5) in Abbildung 4.1 erreicht wurde.
Verstand `victimLink`	vorangegangenes `(REG|STL|TO|IA)ack` steht aus	Verschiebe Versand bis `(REG|STL|TO|IA)ack` eingetroffen ist.
	zwischen (V1) und (V5) in Abbildung 4.1	Verschiebe Versand bis (V5) in Abbildung 4.1 erreicht wurde.
Versandt `IAReq`	zwischen (R1) und (R3) in Abbildung 4.2	Verschiebe Versand, bis (R3) in Abbildung 4.2 erreicht wurde, sende dann ein Backup als Einweihungs- und Übernahme-Backup.
Verstand `TOReq`	zwischen (C1) und (C3) in Abbildung 4.2	Verschiebe Versand bis (C3) in Abbildung 4.2 erreicht wurde, sende dann ein Backup als Einweihungs- und Übernahme-Backup.
Versand `startGrowth`	zwischen (C1) und (C3) in Abbildung 4.2	Verschiebe Versand bis (C3) in Abbildung 4.2 erreicht wurde.
Ausführung `restore`	zwischen (E1) und (E5) in Abbildung 5.1	Sende Nachricht `referral`, wenn (E5) in Abbildung 5.1 erreicht wurde.

Tabelle 4.2.: Auflistung aller verschiebbaren Aktionen. In der linken Spalte ist die zu verschiebende Aktion aufgeführt. Die mittlere Spalte nennt den Zustand, in der die entsprechende Aktion verschoben wird. Die rechte Spalte gibt an, bis wann die Aktion verschoben wird.

doi:10.17170/kobra-2018122577

4.2. Backups und Stehlprotokoll

Wie in GLB startet jeder Worker initial mit einer ggf. leeren Menge an zu bearbeitenden Tasks sowie einem leeren Teilergebnis. Vor dem Start der Berechnung sendet jeder Worker ein initiales Backup an seinen Backup-Place. Die Berechnung startet erst, wenn sichergestellt ist, dass das initiale Backup jedes Workers auf dem jeweiligen Backup-Place gespeichert wurde.

Die Aktualisierung der Backup-Daten erfolgt zum einen periodisch, zum anderen wenn Tasks von einem Worker zu einem anderen im Zuge eines Stehlvorgangs verschoben werden. Den zweiten Fall bezeichnen wir fortan als *stehlinduzierte Backups*. Die periodische Aktualisierung der Backups führen wir in Abschnitt 4.2.1 ein.

Stehlinduzierte Backups beschreiben wir in Abschnitt 4.2.2. Hierzu modifizieren wir den Stehlvorgang aus GLB, sodass während des Stehlvorgangs die Datenkonsistenz gewährleistet wird.

4.2.1. Reguläre Backups

Um die Anzahl der abermals zu berechnenden Tasks nach einer Wiederherstellung zu begrenzen, aktualisiert ein Worker sein Backup regelmäßig. Wir haben dazu einen Parameter `k` eingeführt, sodass jeder Worker nach der Abarbeitung von $\texttt{n} \cdot \texttt{k}$ Tasks ein Backup sendet. Hierbei ist `n` der aus Abschnitt 3.2 bekannte GLB-Parameter. Das Senden des Backups wird über ein lokal generiertes Ereignis ausgelöst, welches durch die Methode `processRecorded()` (Zeile 6 in Quellcodeabbildung 4.1) erstellt und abgearbeitet wird. Das Senden eines Backups geschieht mit einer Nachricht `REGreq` (**REG**ular**B**ackup**req**uest) von einem Workers `P` an seinen Backup-Place `Back(P)`. Ist eine vorher gesendete Backup-Nachricht noch nicht durch eine entsprechende `ack`-Nachricht beantwortet worden, so wird das Senden des regulären Backups verschoben. Ein reguläres Backup beinhaltet den Taskpool sowie das Teilergebnis des sendenden Workers.

Nach Ausführung der Nachricht `REGreq` sendet `Back(P)` eine Bestätigungsnachricht `REGack` (**REG**ular**B**ackup**ack**nowledgement) an `P`.

4.2.2. Stehlinduzierte Backups

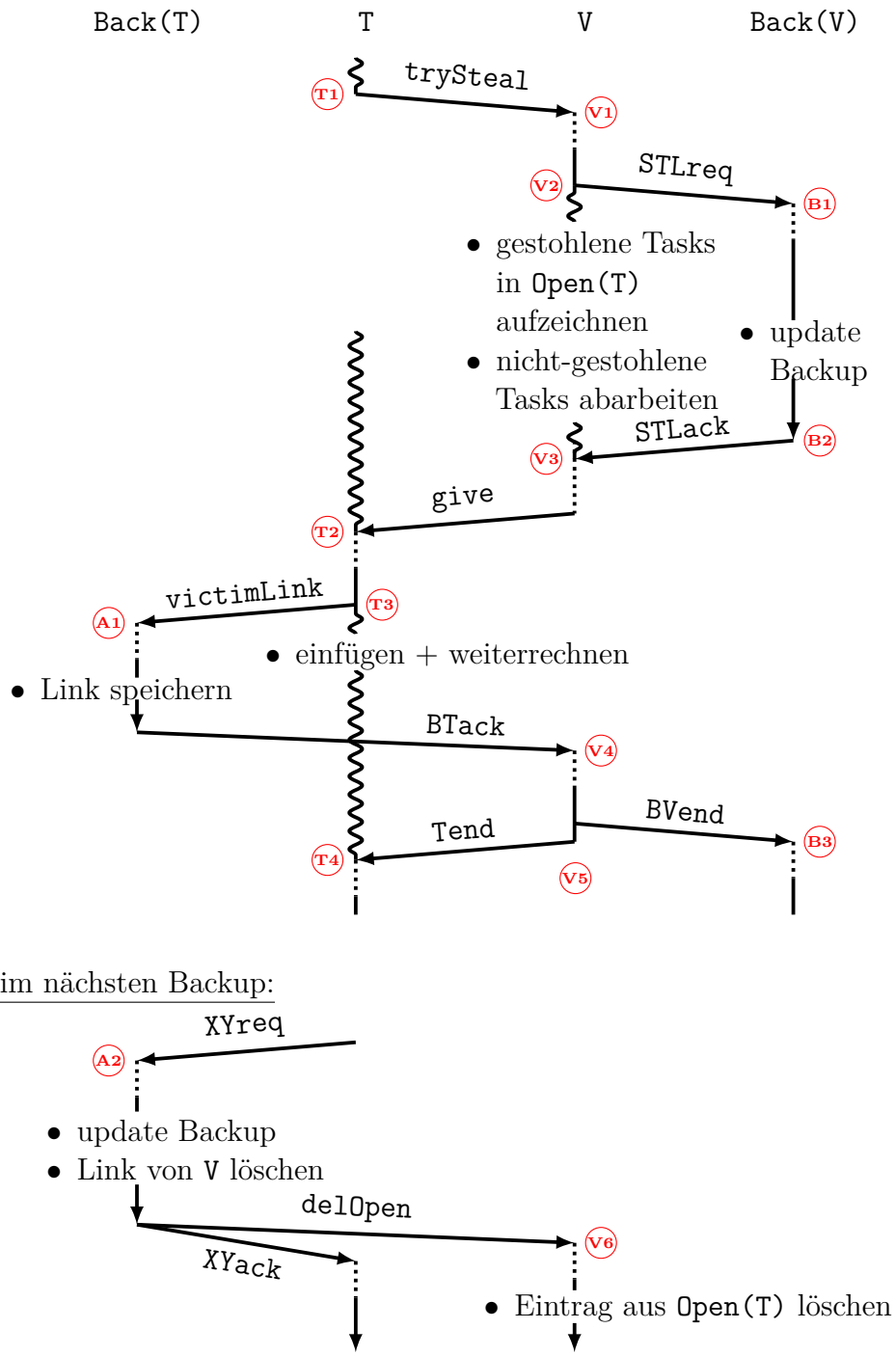
Werden im Zuge eines Stehlvorgangs Tasks als Beute von einem Opfer `V` zu einem Dieb `T` gesendet, so muss während des Transfers sichergestellt sein, dass genau ein Worker für die Wiederherstellung der Beute verantwortlich ist. Würde die Beute von keinem oder mehr als einem Worker wiederhergestellt, so würde das Programm möglicherweise ein falsches Ergebnis berechnen. Um dies zu vermeiden, ist ein Eingriff in den Stehlprozess notwendig. Abbildung 4.1 zeigt unser fehlertolerantes Stehlprotokoll. Stehen Zeitpunkte an Pfeilspitzen (Beispielsweise (V1) an der Pfeilspitze der Nachricht `trySteal`), so sind diese als jener Zeitpunkt zu verstehen, zu dem die Nachricht auf Seiten des Empfängers ausgeführt wird, also der nächste Aufruf von `Runtime.probe()`. Die Abbildung zeigt ein Opfer `V` sowie einen Dieb `T`. Wie in GLB bündeln Opfer alle Stehlanfragen zwischen zwei Aufrufen von `Runtime.probe()` und beantworten diese gemeinsam in einem Aufruf von `processRecorded()` (Zeile 5 bzw. 6 in Quellcodeabbildung 4.1). Somit kann ein Opfer zu einem Zeitpunkt Stehlanfragen mehrerer Diebe bearbeiten. Im Unterschied zu GLB kann ein Worker jedoch entweder als Dieb oder als Opfer in einen Stehlvorgang involviert sein. Wir haben diese Entscheidung getroffen, um das Design des Protokolls einfach zu halten.

Eine Stehlanfrage beginnt immer aufseiten des Diebes `T`, wenn dieser erkennt, dass sein lokaler Pool leer ist. Der Dieb `T` sendet eine Nachricht `trySteal` an das Opfer `V` ((T1) in Abbildung 4.1) und kehrt danach in seine Hauptarbeitsschleife zurück. Dies ist ein gravierender Unterschied zu einem GLB-Worker, welcher innerhalb der Methode `steal(...)` mittels busy waiting auf eine Antwort zu seiner Stehlanfrage wartet. Für einen Dieb gilt eine Stehlanfrage als offen, solange er keine entsprechende Nachricht `noTasks` bzw. `Tend` (`Thiefend`) (T4) erhalten hat. Da der Dieb nun eine offene Stehlanfrage gesendet hat, wird er in diesem Zeitraum die Abarbeitung aufgezeichneter Diebe verschieben, bis die offene Stehlanfrage abgeschlossen ist. Ein Place kann, während er auf die Beantwortung der Stehlanfrage wartet, andere Aufgaben erledigen, wie beispielsweise Backup-Daten seines Vorgängers abspeichern.

Ist `V` inaktiv oder ist sein Taskpool leer, sendet es sofort bei Erhalt von `trySteal` eine Nachricht `noTasks` zurück an `T`. Die Behandlung von Lifeline-Anfragen geschieht hierbei identisch zu GLB, wie in Abschnitt 3.2 beschrieben.

Hat `V` Tasks im Taskpool, so zeichnet es die Stehlanfrage auf und bearbeitet sie mit dem nächsten Aufruf von `processRecorded()` in einer Subroutine

 doi:10.17170/kobra-2018122577



Beim nächsten Backup:

Abbildung 4.1.: Stehlprotokoll (siehe [45])

`processSteals()`. Da der Stehlvorgang auch das Schreiben eines Backups umfasst, kann es vorkommen, dass die Bearbeitung der Stehlanfrage verschoben werden muss. Dies ist der Fall, wenn `V` auf eine ausstehende Nachricht `(REG|STL|TO|IA)ack` wartet. Darüber hinaus verschiebt ein Opfer die Bearbeitung von Stehlanfragen auch, wenn es bereits die Rolle des Opfers oder Diebes in einer offenen Stehltransaktion einnimmt. Wir erklären den Begriff Stehltransaktion nachfolgend.

Bearbeitet das Opfer Stehlanfragen, so formt es eine *Stehltransaktion*, um die Stehlanfragen aller aufgezeichneten Diebe zu beantworten. Eine Stehltransaktion wird durch einen *Transaktionsdeskriptor* dargestellt. In diesem werden die nachfolgenden Daten der Stehltransaktion abgespeichert.

- Eine Transaktionsnummer (kurz *TAN*). Diese ist placeintern fortlaufend. Ein Opfer kann erst eine neue Stehltransaktion eröffnen, wenn die vorherige Stehltransaktion abgeschlossen ist. Zwei Opfer können zum selben Zeitpunkt dieselbe TAN nutzen, jedoch sind Place-ID und TAN zusammen im System einzigartig.

  Die TAN wird bei jeder stehlbezogenen Nachricht mitgesendet und wird im Fehlerfall genutzt, um verloren gegangene Beute zu identifizieren.

- Eine Datenstruktur **`lootToThieves`** in der gespeichert wird, welcher Dieb welche Beute erhalten soll. Hierzu ruft das Opfer für jeden Dieb die Methode `split()` des lokalen Pools auf. Es sei an dieser Stelle noch einmal daran erinnert, dass die vom Benutzer zu implementierende Methode `split()` einen Teil der Tasks aus dem lokalen Pool entnimmt und als `TaskBag` zurückgibt. Hierbei muss mindestens ein Task im Taskpool von `V` verbleiben. Sollte das Opfer nicht genug Beute für alle Diebe haben, so sendet es den restlichen Dieben eine Nachricht `noTasks`.

Nach dem Abarbeiten aller Diebe enthält der Transaktionsdeskriptor die TAN sowie `lootToThieves`. Durch das Herauslösen der Beute hat sich der Taskpool des Opfers geändert. Das Opfer `V` sendet seinen aktuellen Taskpool (ohne die als Beute herausgelösten Tasks), sein aktuelles Teilergebnis, die TAN und eine Liste aller Diebe mittels der Nachricht `STLreq` (**`STeaLBackuprequest`**) (V2) als *Stehl-Backup* an seinen Backup-Place `Back(V)`. `Back(V)` erhält keinerlei Informationen darüber, welcher Dieb welche Beute erhalten soll. Wir bezeichnen den Zeitraum zwischen dem Erhalt der Nachricht `STLreq` (B1) und dem Erhalt

 doi:10.17170/kobra-2018122577

der Nachricht `BVend` (**`BackupVictimend`**) (B3) als *heikel*, da `Back(V)` nicht weiß, ob alle Diebe ihre Beute erhalten haben.

Das Opfer `V` speichert die Einträge aus `lootToThieves` mit der TAN in einer Datenstruktur **`Open`**. Diese Datenstruktur speichert zu einer TAN und einem Dieb die in dieser Transaktion an diesen Dieb gesendeten Tasks ab. Die Einträge in `Open` bleiben gespeichert, bis sichergestellt ist, dass der Backup-Place `Back(T)` eines Diebes `T` die Beute in das Backup aufgenommen hat. Dies wird durch eine Nachricht `delOpen` von `Back(T)` an `V` (V4) signalisiert. Hiernach verlässt `V` die Subroutine `processSteals()`.

`Back(V)` speichert bei Erhalt der Nachricht `STLreq` (B1) die erhaltenen Backup-Daten ab und bestätigt dies mit einer Nachricht `STLack` (**`STeaLBackupacknowledgement`** (B2) an `V`. Nach Erhalt dieser Nachricht (V3) verteilt `V` die Beute mittels `give`-Nachrichten an die Diebe der Stehltransaktion.

Ein Dieb `T` reagiert auf eine Nachricht `give` (T2), indem er die erhaltenen Tasks sofort in seinen Taskpool einfügt und die TAN in einer Datenstruktur **`tanGOT`** speichert. Diese hat einen Eintrag pro Worker und enthält pro potentiellen Sender die aktuellste TAN, die durch eine `give`-Nachricht von diesem Sender eingetroffen ist. Da ein Dieb durch Lifeline-Nachrichten zeitgleich Tasks von mehreren Opfern erhalten kann, pflegt jeder Worker eine Liste **`openTEnds`**, in der er festhält, welche Stehltransaktionen aus seiner Sicht als Dieb noch nicht abgeschlossen sind. Abgespeichert werden TAN und ID des Opfers. Einträge aus dieser Liste werden erst gelöscht, wenn `T` eine entsprechende Nachricht `Tend` von `V` erhält (T4). Zwischen (T2) und (T4) darf `T` keine Backups senden, da er noch nicht sicher sein kann, dass `Back(T)` den Link erhalten hat. Wir diskutieren diesen Fall später in diesem Kapitel.

Wenn `T` bei Ausführung der Nachricht `give` ein Backup senden kann, also auf keine ausstehende Nachricht `(REG|STL|IA|TO)ack)` wartet, sendet er die TAN sowie die ID von `V` über eine Nachricht `victimLink` an `Back(T)`. Andernfalls sendet `T` die Nachricht, sobald er die ausstehende Nachricht `(REG|STL|IA|TO)ack)` erhalten hat. An dieser Stelle nutzen wir aus, dass die Beute von `T` noch auf `V` in der Datenstruktur `Open` abgespeichert ist. Die gewünschte Redundanz liegt also bereits vor, sodass eine weitere Kopie der Daten auf `Back(T)` nicht erforderlich ist. Der Eintrag in `Open` auf `V` sowie der Link auf `Back(T)` werden mit dem Erhalt des nächsten Backup von `T` (A2) überflüssig, da die gestohlenen Tasks dann implizit im Backup enthalten sind. `Back(T)` prüft vor dem Abspeichern des Links die Lebendigkeit von `V` und wird

den Link nur abspeichern, wenn V noch lebendig ist. Ist V ausgefallen, so sendet Back(T) eine deathNotice an Forth(V). Fallen T und V aus, so reicht die Diebesliste auf Back(V) aus um zu überprüfen, ob Daten endgültig verloren gegangen sind. Details zu beiden Fällen besprechen wir in Abschnitt 4.4.

Nach Abspeichern dieser Links sendet Back(T) eine Nachricht BTack an V und signalisiert somit, dass T die Beute übernommen hat. Erhält V diese Nachricht (V4), so sendet es eine Nachricht Tend an T. Diese Nachricht hat für T zwei Bedeutungen:

1. Die Stehltransaktion ist aus seiner Sicht abgeschlossen.
2. Er kann sicher sein, dass Back(T) den Link zu V abgespeichert hat.

Der zweite Punkt ist wichtig, da T zwischen dem Senden des Links (T3) und dem Erhalt von Tend (T4) keine neuen Backups senden darf. Andernfalls könnte es sein, dass eine neuere Backup-Nachricht die victimLink-Nachricht überholt und somit eine Inkonsistenz entsteht.

Das Opfer V pflegt eine Liste OpenBTack aller Diebe, von deren Backup-Places es noch eine Nachricht BTack erwartet. Erhält das Opfer V die letzte ausstehende BTack-Nachricht, so ist auch für das Opfer die Stehltransaktion abgeschlossen (V5). Mit der Nachricht BVend signalisiert es Back(V) dass alle Diebe ihre Beute erhalten haben und der heikle Zustand verlassen werden kann. Bei Erhalt der Nachricht BVend löscht Back(V) die Liste der Diebe sowie die TAN (B3).

Mit dem Erhalt des nächsten Backups von T nach Beendigung der Stehltransaktion (A2) löscht, wie vorstehend erwähnt, Back(T) den abgespeicherten Link zu V. Back(T) sendet zusätzlich eine Nachricht delOpen an V, welches bei Erhalt dieser Nachricht (V6) den noch abgespeicherten Eintrag von T in Open entfernt. Zudem bestätigt Back(T) den Erhalt des Backups über eine entsprechende ack-Nachricht an T.

4.3. Lebendigkeitsüberwachung

Wir haben in Abschnitt 3.1 drei Möglichkeiten kennengelernt, um Placeausfälle in X10 zu erkennen. Da sowohl GLB als auch unsere fehlertolerante Erweiterung nur ein einziges finish benutzen, welches die Terminierungserkennung des Programmlaufes steuert, würde eine Fehlererkennung über Ausnahmebehandlung an diesem Punkt zu spät kommen. Die Möglichkeit, Placeausfälle

 doi:10.17170/kobra-2018122577

über einen `PlaceRemovedHandler` zu erkennen steht in Native X10 nicht zur Verfügung und würde die Nutzbarkeit des Frameworks somit auf Managed X10 beschränken. Zudem würde der `PlaceRemovedHandler` in einer separaten Aktivität ausgeführt.

Hier sei an die Grundstruktur von GLB (siehe Abschnitt 3.2) erinnert: Die Terminierungserkennung findet über ein äußeres `finish` statt. Die Aktivität, welche den `PlaceRemovedHandler` ausführt, bindet sich nicht an dieses äußere `finish` und erschwert die Terminierungserkennung. Posner und Fohry [85] haben dieses Problem durch einen sogenannten *Restart Daemon* umgangen. Dieser muss jedoch als separate Aktivität gestartet werden.

Der vorgestellte Algorithmus realisiert daher die Lebendigkeitsüberprüfung individueller Places mithilfe der Methode `Place.isDead(<Place>)`. Um zu gewährleisten, dass jeder Place in ausreichend kurzen Zeitabständen auf seine Lebendigkeit überwacht wird, werden die drei folgenden Überwachungstechniken verwendet.

Regelmäßige Überwachung des Backup-Places. Die erste Technik basiert auf dem Backup-Ring. Jeder aktive Worker überprüft regelmäßig die Lebendigkeit seines Backup-Places. Dies geschieht implizit bereits durch das Senden regulärer Backups, welche wir in Abschnitt 4.2.1 besprochen haben.

Ghost-Aktivität. Ist ein Worker inaktiv, so ruft er `processRecorded()` nicht auf. Dies führt zu dem Problemen, dass ein inaktiver Worker `P` seinen Backup-Place `Back(P)` nicht auf Lebendigkeit überprüft. Dieses Problem lässt sich vermeiden, indem jeder inaktive Worker regelmäßig aufgeweckt wird, um die Lebendigkeit seines Backup-Places zu überprüfen. Jeder aktive Worker `P` sendet hierzu in jeder `g`-ten Ausführung von `processRecorded()` eine Nachricht `monitor` an `Back(P)`. Hierbei ist `g` ein Parameter des fehlertoleranten Frameworks. Durch die Nachricht `monitor` wird `processRecorded()` auf `Back(P)` ausgeführt, falls `Back(P)` inaktiv ist. Diese Nachricht wird auch von einem Worker gesendet, wenn der Worker `processRecorded()` ausführt, aber inaktiv ist. Somit werden alle inaktiven Worker regelmäßig reaktiviert, arbeiten aufgezeichnete Nachrichten ab und senden eine Nachricht `monitor` an ihren Backup-Place. Erreicht eine `monitor`-Nachricht den Place, der sie ursprünglich abgeschickt hat, so wird dieser die `monitor`-Nachricht nicht weiterleiten. Dies verhindert, dass das Netzwerk mit `monitor`-Nachrichten geflutet wird.

Sender	Gesendete Nachricht	Empfänger	Erwartete Antwort
T	trySteal	V	give oder noTasks
P	(REG|STL|TO)Req	Back(V)	XYack
Back(V)	STLack	V	BVend
V	give	T	BTack
T	victimLink	Back(T)	Tend
Forth(P)	restore	Back(P)	RSTack
Back(P)	linkResolve	V	linkTasks
Back(P)	GOTcheck	T	GOTok
Back(P)	RSTack	Forth(P)	IAreq

Tabelle 4.3.: Nachrichten, die zu Überwachung des Empfängers durch den Sender führen.

Nachrichten-Timeout. Bei vielen Nachrichten weiß der Sender einer Nachricht bereits, dass im weiteren Verlauf des Protokolls eine Antwort auf seine Nachricht bei ihm eingehen muss. Der Sender speichert sich ab, welche Antworten ausstehen und wann die ursprüngliche Nachricht gesendet wurde. Ist nach einer gewissen Zeitspanne, dem *Timeout-Intervall*, zu der gesendeten Nachricht keine Antwort eingegangen, so überprüft der Sender, ob der Empfänger noch lebendig ist. Die nächste Lebendigkeitsüberprüfung findet erst statt, nachdem das nächste Timeout-Intervall abgelaufen ist. Tabelle 4.3 zeigt alle Nachrichten aus Tabelle 4.1, die zu einer Überwachung des Empfängers durch den Sender führen. Hinzuweisen ist hierbei auf eine asymmetrische Überwachung: Ein Opfer überwacht nach dem Senden der Nachricht `give` einen Dieb `T`, bis es die entsprechende `BTack`-Nachricht von Backup-Place `Back(T)` erhält.

Sollte eine Nachricht über mehrere Timeout-Intervalle hinweg unbeantwortet bleiben, so wird der Programmlauf präventiv abgebrochen. Wir nennen dieses Vorgehen den *ultimativen Timeout*. Dies ist notwendig, da es durch bestimmte Ausfallkonstellationen dazu kommen kann, dass das Wiederherstellungsprotokoll nicht terminiert. Wir besprechen diesen Aspekt in Abschnitt 4.4.2 genauer.

 doi:10.17170/kobra-2018122577

```
1 while (!active && !received(RSTack)) {
2   Runtime.probe ();
3   record (if (!active(Back(P)) {
4     start (regular) ghost on Back(P)
5   })
6   processRecorded ();
7 }
```

Quellcodeabbildung 4.2: Langlebige Ghost-Aktivität auf `Forth(P)` (siehe [45])

4.4. Wiederherstellung

Abbildung 4.2 zeigt das Wiederherstellungsprotokoll unseres fehlertoleranten Algorithmus. Um die Funktionsweise zu erklären, besprechen wir einzelne Placeausfälle zu verschiedenen Zeitpunkten in Abschnitt 4.4.1.

Ausfälle mehrerer Places betrachten wir in Abschnitt 4.4.2. Wir diskutieren für alle möglichen Situationen, was geschieht, wenn während der Wiederherstellung eines Places ein weiterer Place ausfällt und zeigen, wie die Wiederherstellung verschachtelt ausgeführt wird.

Um die Tasks eines ausgefallenen Workers übernehmen zu können, haben wir die Benutzerschnittstelle `Queue` um eine Methode `merge(queue:Queue)` erweitert. Ebenso haben wir die Benutzerschnittstelle `GLBResult` um eine Methode `setResult(...)` erweitert.

4.4.1. Ausfall eines Places

Wir gehen nachfolgend davon aus, dass der Place `P` ausgefallen ist und dass `Forth(P)` dies durch die Überwachung oder eine Nachricht `deathNotice(P)` bemerkt hat. Diese Nachricht wird an `Forth(P)` gesendet, falls ein anderer Place as `Forth(P)` den Ausfall von `P` feststellt.

Sobald `Forth(P)` den Ausfall von `P` bemerkt oder durch eine Nachricht `deathNotice(P)` gemeldet bekommt, wechselt dieser in den *Notfallmodus*. Ein Place im Notfallmodus stoppt die Bearbeitung von Tasks und bearbeitet nur noch Nachrichten, die in Tabelle 4.1 als Notfallnachrichten markiert sind („x“ in der Spalte **NM**). Ebenso werden nur noch Timeouts von Nachrichten

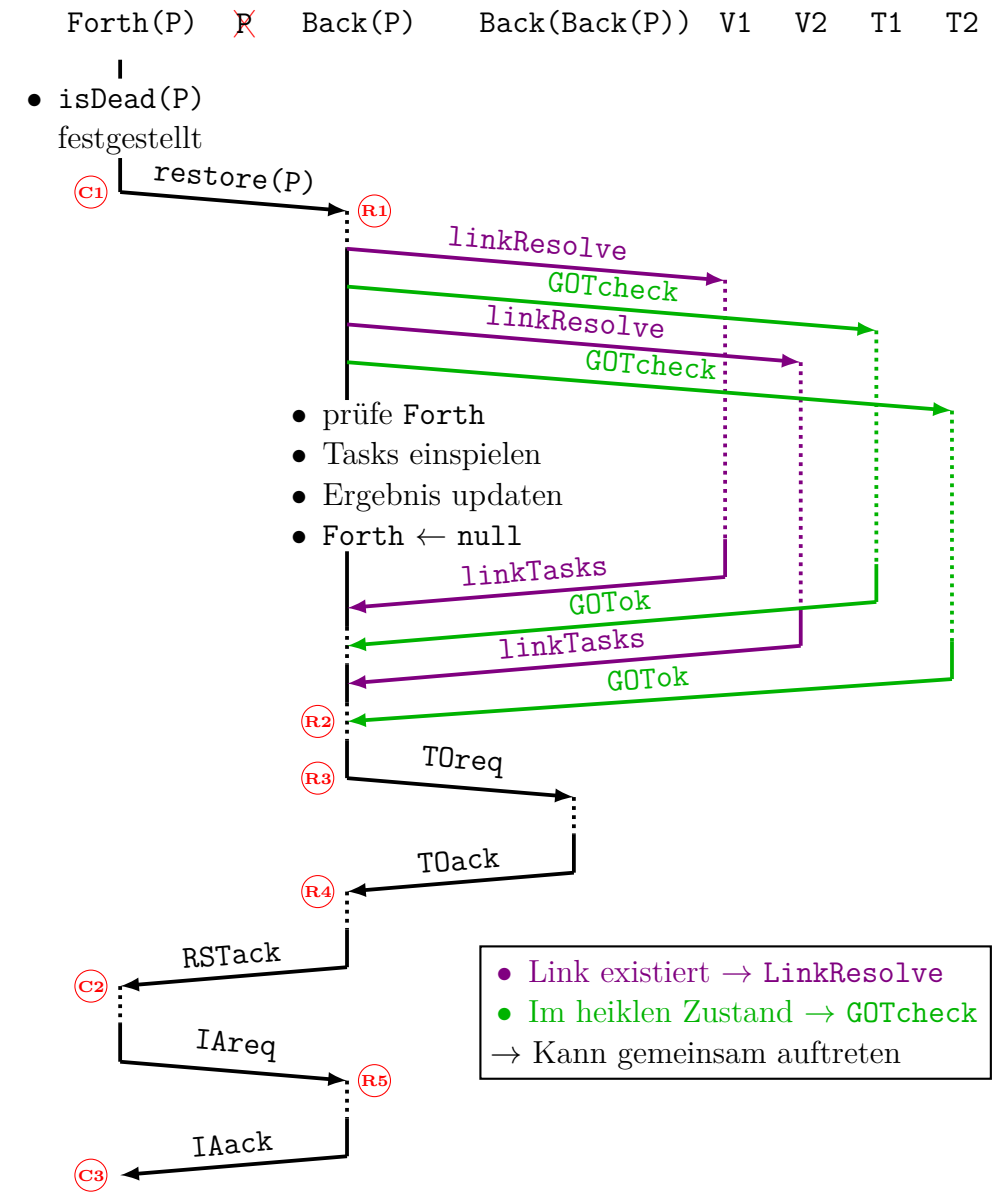


Abbildung 4.2.: Wiederherstellungsprotokoll (siehe [45]).

doi:10.17170/kobra-2018122577

überprüft, die für die Wiederherstellung relevant sind. Sollte `Forth(P)` inaktiv sein, wenn er den Ausfall von `P` feststellt, so startet er eine *langlebige Ghost-Aktivität*, die bis zum Erhalt der Nachricht `RSTack` (**`ReSToreacknowledgement`**, (C2)) ausgeführt wird. Quellcodeabbildung 4.2 zeigt den Pseudocode dieser langlebigen Ghost-Aktivität auf `Forth(P)`. Hierbei steht `record(...)` für das Aufzeichnen einer Nachricht. `Forth(P)` startet auf `Back(P)` eine Ghost-Aktivität (siehe Abschnitt 4.3), falls `Back(P)` inaktiv ist (Zeilen 3 bis 5). Diese Ausführung der Ghost-Aktivität ist im Gegensatz zur in Abschnitt 4.3 erklärten Ghost-Aktivität nicht rekursiv. Durch die langlebige Ghost-Aktivität auf `Forth(P)` sowie das wiederholte Starten einer Ghost-Aktivtät auf `Back(P)`, falls dieser inaktiv ist, wird gewährleistet, dass alle wiederherstellungsbezogenen Nachrichten zeitnah ausgeführt werden.

Die Wiederherstellung startet mit einer Nachricht `restore(P)` von `Forth(P)` an `Back(P)` (C1). `Back(P)` ist als nächster lebendiger Nachfolger von `P` im Backup-Ring zu verstehen.

Mit dem Erhalt der Nachricht `restore(P)` (R1) wechselt `Back(P)` ebenfalls in den Notfallmodus. `Back(P)` überprüft, ob er die Daten von `P` wiederherstellen kann. Dies ist notwendig, da zwischen `Forth(P)` und `Back(P)` zwei oder mehr Places liegen könnten, die zeitgleich abgestürzt sind. In diesem Fall sind Daten endgültig verloren gegangen. Um zu entscheiden, ob eine Wiederherstellung möglich ist, betrachtet `Back(P)` den Absender `Forth(P)` sowie die Werte seiner `Forth`-Variable und der `Forth`-Variable im Backup von `P`. Eine Wiederherstellung ist möglich, wenn `P` gleich `Back(P)`s `Forth` und `Forth(P)` gleich dem in `P`s Backup gespeicherten `Forth` ist.

`Back(P)` kann aufgrund der ihm vorliegenden Daten erkennen, ob `P` als Opfer und/oder Dieb in Stehltransaktionen involviert war:

1. Zwischen (B1) und (B3) ist `Back(P)` im heiklen Zustand und kennt die potentiellen Diebe von `P`.

2. Zwischen (A1) und (A2) hat `Back(P)` einen Link zum Opfer `V`.

Zu 1.: Stellt `Back(P)` den Ausfall von `P` durch das Senden der Nachricht `STLack` (B2) fest, so kann `P` die Beute noch nicht an die Diebe verteilt haben. Die Beute aller Diebe ist somit endgültig verloren gegangen. Daher bricht `Back(P)` den Programmlauf sofort ab.

Stellt `Back(P)` den Ausfall von `P` zwischen (B1) und (B3) fest, so ist unklar, ob

alle Diebe ihre Beute erhalten haben. Um dies zu prüfen, sendet `Back(P)` an jeden der Diebe eine Nachricht `GOTcheck(P, TAN)`.

Erhält ein Dieb `T` eine Nachricht `GOTcheck(P, TAN)`, so folgt er den folgenden drei Regeln:

1. `TAN` $\in$ `tanGOT` $\wedge$ `TAN` $\notin$ `openTends`: Sende `GOTok`.
2. `TAN` $\notin$ `tanGOT`: Verschiebe die Bearbeitung dieser Nachricht, um Verzögerungen im Netzwerk zu tolerieren. Trifft diese Regel danach immernoch zu, so breche den Programmlauf ab.
3. `TAN` $\in$ `tanGOT` $\wedge$ `TAN` $\in$ `openTends`: Entferne den entsprechenden Eintrag aus `openTends`. Da `Tend` nicht mehr gesendet werden kann, übertrage den Link mittels synchroner Kommunikation an `Back(T)` und sende `GOTok`. Auf Seiten von `Back(T)` wird der Link gespeichert, obwohl `V` ausgefallen ist. Wir begründen dieses Verhalten, wenn wir den zweiten Fall besprechen. Ist auch `Back(T)` ausgefallen wenn `T` die Nachricht `GOTcheck` erhält, so sendet `T` sofort eine Nachricht `GOTok`. Wir begründen dieses Verhalten in Abschnitt 4.4.2.

Wenn mindestens ein Dieb seine Beute nicht erhalten hat, so sind Tasks endgültig verloren gegangen und der Programmlauf muss abgebrochen werden. Dies ist die einzige Situation, in der ein einzelner Placeausfall zu einem Programmabbruch führen kann.

Zu 2.: War `P` Dieb einer Stehltransaktion, so fordert `Back(P)` die Beute über den Link beim entsprechenden Opfer an. Dazu sendet er eine Nachricht `linkTasks(P, TAN)`. Da `P` (beispielsweise durch Lifeline-Anfragen) in mehrere Stehltransaktionen als Dieb involviert sein kann, muss dieser Nachrichtentyp ggf. an mehrere Opfer gesendet werden.

Ein Opfer `V` sendet als Antwort eine Nachricht `linkResolve` mit der Beute, die es in `Open` zu `P` abgespeichert hat. Es kann sein, dass `V` den Ausfall von `P` bemerkt hat, bevor es die `linkTasks`-Nachricht erhält (siehe Abschnitt 4.3). In diesem Fall hat `V` die zu `P` in `Open` abgespeicherten Tasks bereits in seinen Taskpool eingespielt und sein Backup mittels einer Nachricht `TOreq` (`TakeOverBackuprequest`) aktualisiert. Die entsprechende `linkTasks`-Nachricht wird in diesem Fall mit `null` beantwortet.

Sobald `Back(P)` eine Nachricht `linkTasks` erhält, fügt er die erhaltene Beute in seinen Taskpool ein.

 doi:10.17170/kobra-2018122577

Sollte Back(P) eine GOTcheck- oder linkResolve-Nachricht an einen ausgefallenen Place senden, so wird der Programmlauf ebenfalls wegen möglichem endgültigen Datenverlust abgebrochen. Hiermit können wir auch begründen, warum Back(T) nur Links zu lebendigen Opfern speichern muss: Ist ein Opfer V bereits tot, wenn Back(T) den Link erhält (Ⓐ1 in Abbildung 4.1), kann das Opfer BVend nicht an seinen Backup-Place gesendet haben und dieser ist noch im heiklen Zustand. Somit wird Back(V), wenn er versucht die Daten von V wiederherzustellen, eine Nachricht GOTcheck an T senden. Ist T ausgefallen, bricht Back(V) den Programmlauf ab.

Ist T lebendig, wird dieser den Link synchron an Back(T) übertragen und den Erhalt der Tasks mit einer Nachricht GOTok bestätigen. Es sei noch einmal erwähnt, dass synchron übertragene Links auf Back(T) immer gespeichert werden, unabhängig davon ob das Opfer lebendig ist oder nicht. Sollte Back(T) eine Nachricht restore(T) erhalten, bevor er das nächste Backup von T ausgeführt hat (Ⓐ2 in Abbildung 4.1), so sendet er eine linkResolve-Nachricht an den abgestürzten Place V, da ein offener Link zu V vorliegt. Dies führt dann zu einem Programmabbruch.

Nachdem Back(P) alle linkResolve- und GOTcheck-Nachrichten versendet hat, fügt dieser den als Backup gespeicherten Taskpool und die darin enthaltenen Tasks in seinen Taskpool ein und übernimmt das Teilergebnis von P mittels Reduktion in sein Teilergebnis. Zudem setzt Back(P) seine Forth-Variable auf null, da er ab diesem Zeitpunkt bis zum Erhalt eines Einweihungs-Backups Ⓡ5 keine Backup-Daten mehr speichert.

Mit Bearbeitung der letzten ausstehenden linkResolve- bzw. GOTok-Nachricht sind alle Daten von P eingeholt. Der Taskpool von Back(P) beinhaltet nun alle Tasks und Teilergebnisse von P Ⓡ2. Back(P) sendet ein Übernahme-Backup durch eine Nachricht TOreq an seinen Backup-Place Back(Back(P)). Dieses Backup beinhaltet neben dem Taskpool auch Forth(P) als neuen Wert für die Forth-Variable innerhalb des Backups. In Abschnitt 4.4.2 werden wir sehen, warum diese verfrühte Aktualisierung der Forth-Variablen im Backup notwendig ist. Sobald Back(Back(P)) das Backup abgespeichert hat, signalisiert er dies durch eine Nachricht TOack an Back(P). Dieser wiederum signalisiert die erfolgreiche Wiederherstellung von P mit einer Nachricht RSTack (ReSToreacknowledgement) an Forth(P).

Forth(P) reagiert auf den Erhalt dieser Nachricht Ⓒ2 mit dem Senden eines Einweihungs-Backups mittels der Nachricht IAreq

(`InAugurationBackuprequest`). Diese Backup-Nachricht kann auch eine TAN sowie eine Diebesliste enthalten, wenn `Forth(P)` zum Zeitpunkt der Wiederherstellung zwischen (V2) und (V5) des Stehlprotokolls (Abbildung 4.1) war.

Führt `Back(P)` diese Nachricht aus (R5), so speichert er auch `Forth(P)` in seiner `Forth`-Variable ab und verlässt den Notfallmodus. Ist `Back(P)` zu diesem Zeitpunkt inaktiv, so wechselt er in den aktiven Zustand und führt die Hauptarbeitsschleife aus (siehe Quellcodeabbildung 4.1). Dies ist gerechtfertigt, da `Back(P)` durch die Wiederherstellung ggf. Tasks erhalten hat. Er bestätigt den Erhalt des Backups durch die Nachricht `IAack`. Mit dieser Nachricht (C3) verlässt auch `Forth(P)` den Notfallmodus. Die Wiederherstellung von `P` ist abgeschlossen.

4.4.2. Ausfall mehrerer Places

Der nachfolgende Abschnitt orientiert sich in seiner Darstellung an [45].

Es kann passieren, dass gleichzeitig mehrere Places ausfallen, bzw. dass während der Wiederherstellung eines Places weitere Places ausfallen. Wir betrachten nur Ausfälle von Places, die an der Wiederherstellung des ersten ausgefallenen Places beteiligt sind. Wiederherstellungen von Places, die nicht an der gerade betrachteten Wiederherstellung beteidigt sind, laufen parallel und unabhängig zu der betrachteten Wiederherstellung ab und sind daher für diese Betrachtung uninteressant.

Einen Teil dieser Betrachtung haben wir bereits in Abschnitt 4.4.1 begonnen, wo wir beschrieben haben, dass der Programmlauf abgebrochen wird, sobald eine `GOTcheck`- oder `linkResolve`-Nachricht an einen ausgefallenen Place geschickt wird. Neben den Dieben und Opfern von `P` sind drei weitere Worker an der Wiederherstellung beteiligt: `Forth(P)`, `Back(P)` und `Back(Back(P))`. Tabelle 4.4 zeigt alle möglichen Konstellationen für Ausfälle von zwei Places. Wir besprechen diese Konstellationen nachfolgend. Wir verwenden die folgende Nomenklatur: `Back`→`Forth` bedeutet, dass ein Place in der ersten Wiederherstellung die Rolle von `Back` und in der zweiten Wiederherstellung die Rolle von `Forth` eingenommen hat.

`Forth` → `Forth` Dieser Fall ist in Abbildung 4.3 dargestellt. Eine Wiederherstellung ist möglich, wenn die `restore(P)`-Nachricht der zweiten Wiederherstellung nach der `TOreq`-Nachricht der ersten Wiederherstellung bei

 doi:10.17170/kobra-2018122577

1. Protokoll \ 2. Protokoll	`Fort`	`Back`	`Back(Back)`
`Forth`	(+), `B`	+, `F(F)`	+, `F(F(F))`
`Back`	+, `B(B)`	-, `F`	-, `F(F)`
`Back(Back)`	+, `B(B(B))`	(+), `B`	-, `F`

Tabelle 4.4.: Wiederherstellung nach zwei Ausfällen (siehe [45]). Die Zeilen- und Spaltenüberschriften zeigen die Rolle des Workers an, die er in beiden Wiedherstellungsprotokollausführungen einnimmt. Die Einträge benennen die Rolle des Workers im ersten Protokoll, welcher durch den 2. Ausfall ausgefallen ist.
Abkürzungen: `B`: `Back`, `F`: `Forth`, +: Wiederherstellung möglich, (+): Wiederherstellung hängt vom genauen Ausfallzeitpunkt des 2. Ausfalls ab, -: Programmlauf muss abgebrochen werden.
Beispielsweise bedeutet Zeile `Forth`, Spalte `Back`, dass der Worker im 1. Protokoll der `Forth` des ersten ausgefallenen Places und im 2. Protokoll der `Back` des ausgefallenen Places ist. Der entsprechende Eintrag `+, F(F)` gibt zum einen durch das `+` an, dass eine Wiederherstellung möglich ist und zum anderen, dass während der ersten Wiederherstellung der Worker in der Rolle `Forth(Forth)` ausgefallen ist.

`Back(Back(P))` eintrifft. Für eben diesen Fall ist es notwendig, dass `Back(P)` mit dem Übernahme-Backup über eine Nachricht `TOreq` verfrüht den `Forth`-Wert auf `Back(Back(P))` aktualisiert. Wird die Nachricht `restore` vor der Nachricht `TOreq` ausgeführt, so bricht `Back(Back(P))` die Programmausführung ab, da ein endgültiger Verlust der Daten von `P` vorliegt.

`Forth` → `Back` Wird die zweite Wiederherstellungsnachricht erst nach Ⓒ2, Abbildung 4.2 empfangen, so sind die Wiederherstellungen unabhängig. Wird die zweite Wiederherstellungsnachricht jedoch vor Ⓒ2 empfangen, so kommt `Forth` seiner Rolle als `Back` nach, um die Wartezeit bis zum Eintreffen von `RSTack` zu nutzen. Das Senden einer Nachricht `IAreq` im ersten Protokoll wird verzögert, bis das zweite Protokoll Ⓡ3 erreicht hat. Das Einweihungs-Backup des ersten Protokolls ist dann zeitgleich das Übernahme-Backup des zweiten Protokolls.

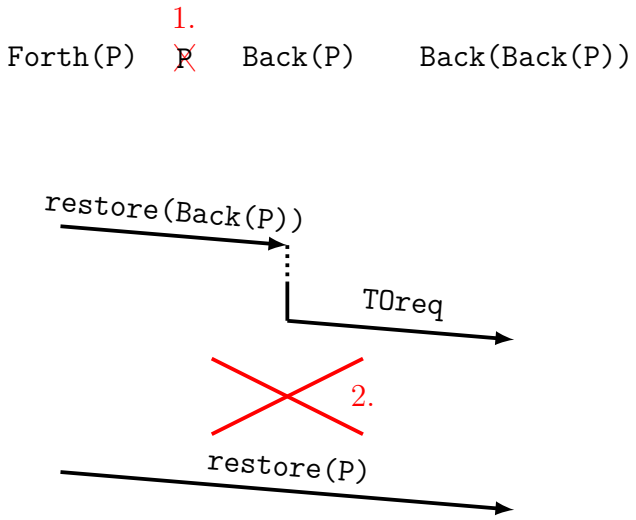


Abbildung 4.3.: Wiederherstellung im Fall `Forth` → `Forth` (siehe [45])

`Forth` → `Back(Back)` Die beiden Wiederherstellungen sind unabhängig und können parallel ausgeführt werden.

`Back` → `Forth` Ist `Back` bereits hinter (R4), so sind die Wiederherstellungen unabhängig. Andernfalls ist dieser Fall ähnlich zu `Forth` → `Back`. Während der Wiederherstellung von `P` fällt `Back(Back(P))` aus. Somit muss die Nachricht `TOreq` des ersten Protokolls bis zum Erhalt der Nachricht `RSTack` im zweiten Protokoll verzögert werden. Es wird dann nur eine Nachricht `IAReq` gesendet, die zeitgleich auch das Übernahme-Backup repräsentiert.

`Back` → `Back` Ein Worker kann zu einem Zeitpunkt das Backup von höchstens einem anderen Worker speichern. Da die erste Wiederherstellung nicht abgeschlossen wurde, kann `Back` noch kein neues Einweihungs-Backup empfangen haben. Somit muss die zweite `restore(...)`-Nachricht zu einem Abbruch des Programmlaufs führen. `Back` erkennt diesen Fall, da seine `Forth`-Variable zu diesem Zeitpunkt den Wert `null` hat.

An dieser Stelle reichen wir die in Abschnitt 4.4.1 (Seite 48) angesprochene Begründung zur 3. Regel bei der Reaktion eines Diebes `T` auf eine `GOTok`-Nachricht nach. Hierzu schauen wir uns den relevanten Teil der 3. Regel nochmals an:

doi:10.17170/kobra-2018122577

3. `TAN` $\in$ `tanGOT` $\wedge$ `TAN` $\in$ `openTends`: [...] Ist auch `Back(T)` ausgefallen wenn `T` die Nachricht `GOTcheck` erhält, so sendet `T` sofort eine Nachricht `GOTok`.

`T` muss im Laufe der Wiederherstellung von `Back(T)` ein Einweihungs-Backup zu seinem neuen Backup-Place `Back(Back(T))` senden. Mit diesem Backup sind die durch `GOTok` bestätigten Tasks gesichert.

Sollte `Back(Back(T))` eine Nachricht `restore(T)` erhalten, bevor er das Einweihungs-Backup von `T` erhalten hat, so sind wir in dem gerade besprochenen Fall `Back` $\rightarrow$ `Back` und das Programm wird wegen endgültig verloren gegangener Daten abgebrochen.

`Back` $\rightarrow$ `Back(Back)` Dieser Fall ist derselbe wie `Forth` $\rightarrow$ `Back`, nur aus der Sicht von `Back`.

`Back(Back)` $\rightarrow$ `Forth` Wie auch schon bei `Forth` $\rightarrow$ `Back` sind diese Wiederherstellungen unabhängig.

`Back(Back)` $\rightarrow$ `Back` Dieser Fall ist derselbe wie `Forth` $\rightarrow$ `Forth`. Eine Wiederherstellung kann nur dann erfolgreich sein, wenn die `restore`-Nachricht der zweiten Wiederherstellung nach der `TOreq`-Nachricht der ersten Wiederherstellung eintrifft.

`Back(Back)` $\rightarrow$ `Back(Back)` Dies ist derselbe Fall wie `Back` $\rightarrow$ `Back` und führt somit zu einem Abbruch des Programmlaufs.

Ausfall von mehr als zwei Places Wir betrachten nun was geschieht, wenn ein Worker in mehr als zwei Wiederherstellungen involviert ist. Von der obigen Betrachtung wissen wir, dass wann immer ein Worker als `Back(Back)` in die Wiederherstellung involviert ist, die Wiederherstellungen entweder unabhängig ablaufen können oder dies zeitgleich zu einem anderen Fall gehört. Wir betrachten, in wie vielen `Forth`- und `Back`-Rollen ein Worker zeitgleich sein kann. Aus `Forth` $\rightarrow$ `Forth` wissen wir, dass ein Worker in mehreren Wiederherstellungen als `Forth` involviert sein kann, aber eine Wiederherstellung muss bis mindestens $\textcircled{c2}$ gekommen sein, bevor die nächste begonnen werden kann. Wir bezeichnen die aktuell in der Ausführung befindliche Wiederherstellung als die *aktive Wiederherstellung*. Aus dem Fall `Back` $\rightarrow$ `Back` wissen wir zudem,

dass ein Worker in höchstens eine Wiederherstellung als `Back` involviert sein darf. Andernfalls wird der Programmlauf aufgrund unwiderruflich verloren gegangener Daten abgebrochen. Somit kann ein Worker in höchstens einer Wiederherstellung als `Back` und in höchstens einer aktiven Wiederherstellung als `Forth` involviert sein.

Wir reichen an dieser Stelle die Begründung für die Einführung des ultimativen Timeouts aus Abschnitt 4.3 nach: Fallen Places in ungünstiger Konstellation aus, so können Ausführungen des Wiederherstellungsprotokolls zu einem Livelock führen. Wir verdeutlichen das Problem an dem folgenden Beispiel.

Betrachtet wird eine Programmausführung mit den vier Places `0`, `1`, `2` und `3`. Während der Programmausführung fallen Place `1` und Place `3` aus, bevor eine dieser Wiederherstellungen abgeschlossen ist. Place `0` bemerkt den Ausfall von Place `1` und sendet eine Nachricht `restore(1)` an Place `2` (C1). Place `2` bemerkt den Ausfall von Place `3` spätestens dann, wenn er versucht die Nachricht `TOreq` an Place `3` zu senden (R3). Place `2` wird eine Nachricht `restore(3)` an Place `0` senden, die Place `0` niemals beantworten wird, da Place `0` auf die Nachricht `RSTack` von Place `2` wartet (C2). Die zur Wiederherstellung benötigten Daten sind vorhanden. Essentiell hierbei ist, dass die Wiederherstellung eines Ausfalls auf die Wiederherstellung eines anderen Ausfalls wartet, und umgekehrt. Somit kommt es zu einem Livelock. Der Programmlauf wird über den ultimativen Timeout abgebrochen.

Eine Anpassung des Stehl- und Wiederherstellungsprotokolls wäre möglich, jedoch besteht die Möglichkeit weiterer Livelock-Situationen. Um alle Livelock-Situationen aufzudecken wäre eine umfassende Analyse des Algorithmus nötig. Wir haben hier den ultimativen Timeout als pragmatischen Lösungsansatz gewählt. In unseren Tests und Experimenten wurden Programmläufe nie aufgrund eines ultimativen Timeouts abgebrochen.

4.4.3. Rekonstruktion des Lifeline-Graphen

Placeausfälle können einen ungünstigen Verlust von Lifelines verursachen. Abbildung 4.4 (links) zeigt eine solche Situation. Hierbei handelt es sich um den Lifeline-Graphen aus Abbildung 3.2, jedoch sind die Places `2` und `6` ausgefallen. Ein Pfeil von einem Place `P` zu einem andere Place `Q` bedeutet, dass `Q` ein Lifeline-Buddy von `P` ist. Place `P` sendet also Lifeline-Anfragen an `Q`. Place `3` hat die Places `1` und `7` als Lifeline-Buddies. Place `7` hat die Places `3`

 doi:10.17170/kobra-2018122577

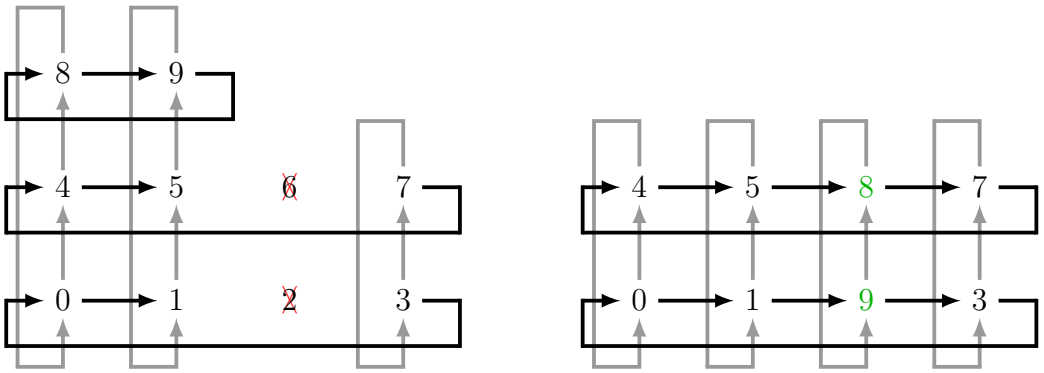


Abbildung 4.4.: Degenerierter Lifeline-Graph der Dimensionalität 2, mit `l`= 4 und ursprünglich 10 Knoten; links: vor der Rekonstruktion, rechts: nach der Rekonstruktion

und `4` als Lifeline-Buddies. Induziert durch Lifeline-Anfragen können Tasks von Place `0` bzw. `4` an Place `3` bzw. `7` verteilt werden. Place `3` ist nur noch Lifeline-Buddy von `7`, und `7` nur noch von `3`. Durch den Erhalt von zufälligen Stehlanfragen können die Places `3` und `7` an andere Places Tasks abgeben. Sind jedoch nur noch Place `3` und `7` aktiv, so werden die Places `0`, `1`, `4`, `5`, `8` und `9` im weiteren Programmlauf keine weiteren Tasks bearbeiten, da keiner dieser Places eine offene Lifeline zu den einzig aktiven Places `3` und `7` besitzt. Wir sprechen davon, dass der Lifeline-Graph *degeneriert* ist. Ein degenerierter Lifeline-Graph beeinflusst nicht die Korrektheit, da weiterhin jeder Task bearbeitet wird. Er kann aber zu einer schlechten Lastenbalancierung und somit zu schlechter Performance führen. Im Extremfall hat ein Place alle Tasks und alle andere Places sind inaktiv.

Um dieses Problem zu erkennen, überprüft Place `0` den Lifeline-Graphen. Hierzu prüft er, ob im Graphen noch ein Pfad von jedem Knoten zu jedem anderen Knoten existiert. Im Beispiel aus Abbildung 4.4 (links) haben die Knoten `3` und `7` keinen Pfad mehr zu den übrigen Knoten. Da diese Überprüfung vor allem bei Berechnungen mit vielen Places rechenintensiv sein kann, wird sie nur in größeren Zeitabständen ausgeführt und auch nur dann, wenn sich seit der letzten Überprüfung die Anzahl an ausgefallenen Places verändert hat.

Zur Rekonstruktion des Graphen wird die erste Lücke (Place `2`) mit dem letzten lebendigen Place (Place `9`) gefüllt. Dieser Prozess wird wiederholt, bis alle Lücken geschlossen sind. Die vordersten Lücken werden mit den hintersten Places gefüllt, um möglichst wenig strukturelle Änderungen am Lifeline-Graphen

vorzunehmen. Würden beispielsweise die Lücken durch Aufrücken geschlossen, würden sich die Lifeline-Buddies von mehr Places ändern. Der Graph aus Abbildung 4.4 (links) wird durch diesen Prozess zu dem Graphen aus Abbildung 4.4 (rechts) rekonstruiert. Die Places `9` und `8` haben die ausgefallenen Places `2` und `6` ersetzt. Die Verteilung erfolgt mithilfe einer asynchronen Nachricht `startup` von Place `0` an alle anderen Worker. Diese Nachricht ist ebenfalls Teil des Elastizitätsprotokolls und wird in Kapitel 5 in diesem Kontext nochmals behandelt. Ist ein Worker inaktiv, so wird er durch die Nachricht `startup` aktiviert, da sich seine Lifeline-Buddies geändert haben könnten, und beginnt erneut, Tasks von Lifeline-Buddies zu stehlen.

doi:10.17170/kobra-2018122577

5. Elastizität

Im Nachfolgenden besprechen wir das Protokoll, das das Hinzufügen neuer Places zu einem Programmlauf regelt. Die Inhalte dieses Kapitels wurden bereits in [43] publiziert. Das Kapitel ist analog zu Kapitel 4 aufgebaut, d. h. wir beginnen mit der Beschreibung der Kernidee (Abschnitt 5.1), gefolgt von der Einführung des Protokolls unter der Voraussetzung, dass keine Fehler auftreten (Abschnitt 5.2). Abschließend betrachten wir die Funktionsweise des Elastizitätsprotokolls, falls Fehler auftreten (Abschnitt 5.3).

5.1. Kernidee

Aus Abschnitt 3.1 wissen wir, dass Elastizität in X10 auf zwei Weisen unterstützt wird. Zum einen kann auf jedem Place eine Methode als `PlaceAddedHandler` registriert werden. Wird ein Place zu einem Programmlauf hinzugefügt, wird diese Methode automatisch ausgeführt. Die grundlegende Funktionsweise gleicht der des `PlaceRemovedHandler`s. Auch diese Implementierung steht nur in Managed X10 zur Verfügung und die ausgeführte Aktivität bindet sich nicht an bestehende `finish`-Blöcke.

Zum anderen bietet X10 die API-Methoden `Place.places()` und `Place.numPlaces()` an, welche alle Places – auch elastisch hinzugefügte – berücksichtigen. Der Aufruf der Methode `Place.places()` gibt eine Liste aller Places (inklusive abgestürzter Places) zurück, wohingegen `Place.numPlaces()` nur deren Anzahl zurückgibt. Speichert das Programm die Anzahl der zu Programmstart vorhandenen Places in einer Variablen ab, so kann es durch regelmäßiges Aufrufen der Methode `Place.numPlaces()` feststellen, ob Places dem Programmlauf hinzugefügt wurden.

Das vorgestellte Elastizitätsschema nutzt das zweite Verfahren. Wir nennen diese Technik *Beobachten*. Im Elastizitätsschema übernimmt Place `0` das Beobachten und koordiniert alle weiteren Schritte. Die Zentralisierung dieser Aufgaben ist gerechtfertigt, da ein Ausfall von Place `0` weder von X10 noch

von GLB verkraftet werden kann. Durch die zentrale Koordination vermeiden wir eine inkonsistente Sicht der verschiedenen Places.

Wurden Places elastisch hinzugefügt, gilt es, die vier folgenden Punkte abzuarbeiten:

1. Auf jedem hinzugefügten Place muss ein Worker mit leerem Taskpool und leerem Teilergebnis initialisiert werden.

2. Die neuen Worker müssen in den Backup-Ring aufgenommen werden, und jeder neue Worker muss ein initiales Backup schreiben.

3. Der bestehende Lifeline-Graph muss um die neuen Worker erweitert und alle Worker (alt wie neu) müssen über die Änderung informiert werden.

4. Die neuen Worker müssen in die Berechnung eingebunden werden, indem sie mit Tasks versorgt werden.

Wie wir in Abschnitt 3.2 besprochen haben, nutzt die X10-Implementierung von GLB einen `PlaceLocalHandle`, um auf jedem Place einen Worker zu initialisieren. Für die Initialisierung der hinzugefügten Worker bietet X10 entsprechende Methoden an, um den bestehenden`PlaceLocalHandle` zu erweitern.

Zur Integration der neuen Places wird der Backup-Ring erweitert. Hierzu werden die `M` hinzugefügten Places `N`, ..., `N+M- 1` zwischen Place `N-1` und Place `0` eingefügt. Das Elastizitätsprotokoll stellt hierbei sicher, dass während der Erweiterung des Backup-Rings jeder hinzugefügte Place `P` auf seinem Backup-Place `P+1` ein initiales Backup schreibt und dass das Backup von Place `N-1` auf seinen neuen Backup-Place `N` migriert wird. Hierbei wird der Ring von links nach rechts erweitert: zuerst wird das Backup des Places `N-1` übertragen, dann sendet Place `N` sein initiales Backup an Place `N+1`, ..., bis schließlich Place `N+M-1` sein Backup an Place `0` sendet.

Nach erfolgreicher Integration der elastisch hinzugefügten Places berechnet Place `0` den Lifeline-Graphen wie in Abschnitt 4.4.3 beschrieben neu und verteilt diesen an alle Worker (alt wie neu) in der Berechnung. Erst danach werden die elastisch hinzugefügten Worker für die restlichen Worker sichtbar. Zeitgleich werden die Workeraktivitäten auf den elastisch hinzugefügten Places gestartet. Da diese keine Tasks in ihrem Pool haben, beginnen sie, Stehlanfragen zu senden.

 doi:10.17170/kobra-2018122577

Um den Overhead für das Hinzufügen von Places möglichst gering zu halten wird nicht sofort jeder neu hinzugekommene Place in die Berechnung aufgenommen. Erkennt Place `0`, dass mindestens ein Place der Berechnung hinzugefügt wurde, so wartet er eine gewisse Zeit auf weitere, neue Places. Diese Zeitspanne ist standardmäßig auf 1 Sekunde eingestellt. Alle in dieser Zeit eintreffenden Places werden dann gemeinsam in einer Ausführung des Elastizitätsprotokolls in die Berechnung eingebunden. Werden neue Places zum Programmlauf hinzugefügt während das Elastizitätsprotokoll ausgeführt wird, so wird zunächst die vorherige Instanz des Elastizitätsprotokolls abgeschlossen, bevor es ein zweites Mal gestartet wird. Zu jedem Zeitpunkt wird das Elastizitätsprotokoll also höchstens einmal ausgeführt. Ebenso wird die Ausführung des Elastizitätsprotokolls verschoben, wenn Place `0` zu diesem Zeitpunkt in das Wiederherstellungsprotokoll involviert ist.

Im Unterschied zum Fehlertoleranzprotokoll verwendet das Elastizitätsprotokoll größtenteils synchrone Nachrichten. Dies erleichtert das Erkennen sowie Abarbeiten von Placeausfällen während einer Protokollausführung, da wir `try-catch`-Blöcke zum Abfangen von `DeadPlaceException`s nutzen können. Wir besprechen diesen Aspekt näher in Abschnitt 5.3. Um Unklarheiten zu vermeiden, werden wir im Folgenden immer angeben, ob es sich um synchrone oder asynchrone Nachrichten handelt.

5.2. Elastizitätsprotokoll

Für die Beschreibung des Elastizitäsprotokolls gehen wir davon aus, dass in einem Programmlauf bereits `N` Places `0`, ..., `N-1` in die Berechnung eingebunden wurden und `M` Places `N`, ..., `N+M-1` elastisch zum Programmlauf hinzugefügt wurden.

Abbildung 5.1 zeigt das Elastizitätsprotokoll. Die verschiedenen Hintergrundfarben unterteilen das Protokoll in drei Phasen. Ziel von Phase 1 ist die Initialisierung aller benötigten Datenstrukturen, inklusive eines leeren Taskpools und eines leeren Teilergebnisses auf jedem neuen Place.

In Phase 2 wird der Backup-Ring um die neuen Places erweitert. Insbesondere wird das Backup des Places `N-1` von Place `0` auf Place `N` migriert. Parallel hierzu berechnet Place `0` den Lifeline-Graph neu.

Das Hinzufügen der Places endet mit Phase 3. Der neu berechnete Lifeline-Graph wird von Place `0` an alle Places verteilt, auf den neuen Places wird dann die Workeraktivität gestartet.

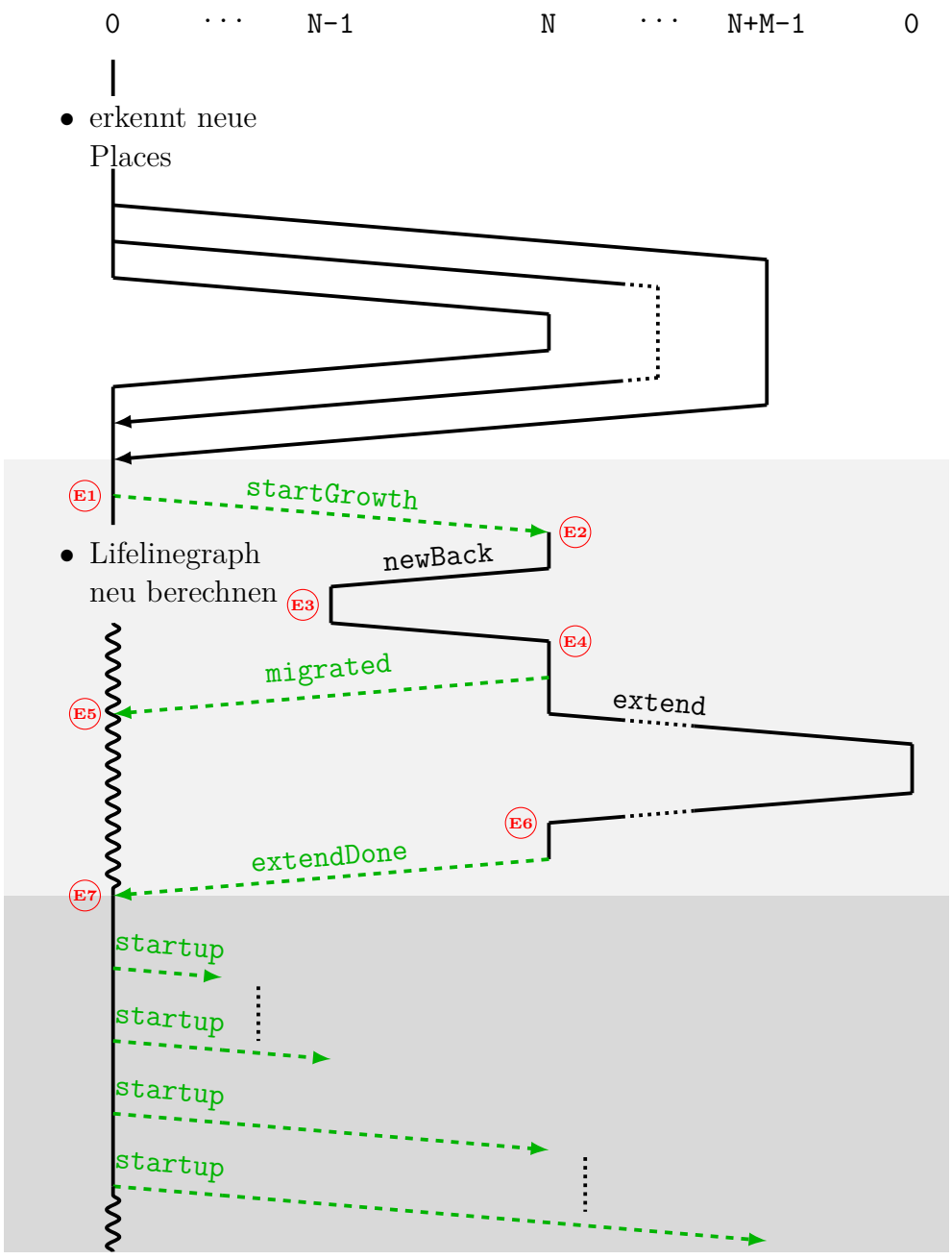


Abbildung 5.1.: Elastizitätsprotokoll (siehe [43]). Schwarze Linien stellen synchrone, grüne gestrichelte Pfeile stellen asynchrone Nachrichten dar.

doi:10.17170/kobra-2018122577

Die Ausführung des Protokolls beginnt immer auf Place `0`, nachdem dieser erkannt hat, dass neue Places zu Berechnung hinzugefügt wurden. Place `0` erweitert den genutzten `PlaceLocalHandle` durch die von X10 bereitgestellte Methode `PlaceLocalHandle.addPlace(...)`. Diese Methode arbeitet synchron. Sind die benötigten Datenstrukturen auf allen neuen Places initialisiert, so endet Phase 1.

Phase 2 startet mit einer asynchronen Nachricht `startGrowth` von Place `0` an den ersten neuen Place `N` (E1). Diese Nachricht enthält alle Backup-Daten, die Place `0` von Place `N-1` abgespeichert hat. Sobald Place `0` diese Nachricht gesendet hat, ist er nicht länger der Backup-Place von `N-1` und setzt dem entsprechend seine `Forth`-Variable auf `null`. Place `0` wird eintreffende `[REG|STL|IA|TO]req`-Nachrichten von `N-1` nicht beantworten. Nach Senden von `startGrowth` löscht Place `0` die von `N-1` gespeicherten Backup-Daten, berechnet den Lifeline-Graphen neu und kehrt dann zur Taskabarbeitung zurück. Place `0` überwacht die Lebendigkeit von `N` über die in Abschnitt 4.3 vorgestellte Timeout-Kontrolle bis er eine Nachricht `extendDone` (E7) von `N` erhalten hat. Wie auf einen Ausfall von Place `N` reagiert wird besprechen wir in Abschnitt 5.3.

Bei Erhalt der Nachricht `startGrowth` (E2) speichert `N` die Backup-Daten von `N-1` ab und setzt seine `Forth`-Variable auf `N-1`. Von hier an steuert Place `N` die Ausführung von Phase 2.

Place `N` sendet eine synchrone Nachricht `newBack` an Place `N-1` (E3). Diese Nachricht hat zwei Effekte: Zum einen wird mit dieser Nachricht Place `N-1` darüber informiert, dass Place `N` sein neuer Backup-Place ist. Zum anderen überträgt `N-1` aktuelle Backup-Daten an `N`. Diese Indirektionalität anstatt einer direkten Nachricht `0` $\rightarrow$ `N-1` wurde gewählt, damit `N` frühestmöglich seine Aufgabe als Backup-Place von `N-1` wahrnehmen kann. Mit dieser Nachrichtenstruktur kann `N` bereits ab (E2) anstatt erst ab (E4) die Daten von `N-1` wiederherstellen. Wir besprechen diesen Punkt näher in Abschnitt 5.3.

Wir erinnern uns daran, dass Place `0` seit dem Senden der Nachricht `startGrowth` eventuelle Backupnachrichten von `N-1` ignoriert. Durch die Nachricht `newBack` werden alle ausstehenden `XYack`-Nachrichten von Place `0` für `N-1` obsolet.

Sobald die Übertragung der aktuellen Backup-Daten von `N-1` an `N` abgeschlossen ist (E4), ist `N` als Backup-Place von `N-1` eingeweiht. Dies meldet `N` mit einer asynchronen Nachricht `migrated` an Place `0`. Sobald Place `0` diese Nachricht ausgeführt hat (E5), sind sich Place `0` und Place `N` einig, dass Place `N` der Backup-Place von `N-1` ist.

Um Phase erfolgreich abzuschließen, muss jeder Place ein initiales Backup an seinen Backup-Place senden. Dies geschieht mit einer Kette synchroner `extend`-Nachrichten. Mit jeder Nachricht übermittelt Place `P` seinen (leeren) Taskpool und sein (leeres) Teilergebnis an seinen neuen Backup-Place `P+1`. Die Übermittlung eines leeren Taskpools und eines leeren Teilergebnisses ist notwendig, da die Repräsentation des Taskpools und des Teilergebnisses Teil des Benutzercodes ist (siehe Abschnitt 3.2). Bei Erhalt dieser Nachricht setzt der Place den Sender als seinen Vorgänger (`Forth`-Variable). Kehrt eine gesendete `extend`-Nachricht zurück, so setzt der Place den Absender als seinen Nachfolger (`Back`-Variable). Die Kette startet mit einer `extend`-Nachricht von `N` an `N+1` und endet, sobald Place `0` eine `extend`-Nachricht erhält (in Abbildung 5.1 ganz rechts). Wir haben diese Nachrichten in X10 als rekursive `at(<Place>)`-Blöcke implementiert, sodass die Ausführung von Place `0` (in Abbildung 5.1 ganz rechts) zu `N+M-1`, ..., zu `N` zurückkehrt (E6). Wir bezeichnen diesen Vorgang als `extend`-*Subprotokoll*. Wir begründen die Wahl von synchroner Kommunikation an dieser Stelle mit zwei Argumenten. Zum einen haben die elastisch hinzugefügten Places zu diesem Zeitpunkt keine Tasks, die sie prozessieren könnten und somit wird der Fortschritt der Berechnung nicht blockiert. Zum anderen erleichtert die Nutzung von `at(<Place>) {...}` die Ausfallerkennung in X10, da wir `try-catch`-Blöcke verwenden können, um `DeadPlaceException`s zu behandeln.

Nach Abschluss des `extend`-Subprotokolls sendet Place `N` eine Nachricht `extendDone` an Place `0` und gibt die Verantwortung für die restliche Ausführung des Elastizitätsprotokolls wieder an Place `0` ab (E7). Hiermit ist Phase 2 erfolgreich abgeschlossen.

Bevor Place `0` den Lifeline-Graphen verteilt, repariert er ihn, falls nötig, wie in Abschnitt 4.4.3 beschrieben. Die Neuverteilung des Lifeline-Graphen geschieht über asynchrone Nachrichten `startup` von Place `0` an alle anderen Places. Erhält ein neu hinzugefügter Place die Nachricht `startup`, so speichert er nicht nur den mitgesendeten Lifeline-Graphen ab, sondern startet auch die Workeraktivität. Da der Taskpool aller neuen Worker leer ist, beginnen diese, Tasks zu stehlen und werden somit in die Berechnung eingebunden. Places könnten offene Lifelines zu alten Lifeline-Buddies haben. Dies ist nicht störend, da höchstens die noch offnenen Lifeline-Anfragen der alten Lifeline-Buddies beantwortet werden. Für neue Stehlanfragen werden ausschließlich die neuen Lifeline-Buddies verwendet.

 doi:10.17170/kobra-2018122577

5.3. Ausfallerkennung und Wiederherstellung

Das Elastizitätsprotokoll ist so gestaltet, dass weite Teile des Protokolls nur von neuen Places ausgeführt werden. Kommunikation zwischen alten und neuen Places während einer Ausführung des Protokolls lässt sich nicht verhindern, ist jedoch minimal gehalten.

Einen Ausfall von Place `0` betrachten wir nicht, da wir voraussetzen, dass Place `0` nicht ausfallen darf (siehe Abschnitt 3.1). Fällt einer der Places `1`, ..., `N-3` (und weder `N-2` noch `N-1`) während einer Ausführung des Elastizitätsprotokolls aus, so läuft die Wiederherstellung über das in Abschnitt 4.4 vorgestellte Wiederherstellungsprotokoll ohne Beteiligung elastisch hinzugefügter Places ab. Wir betrachten nachfolgend Ausfälle der Places `N-2`, `N-1` sowie von neuen Places `N`,..., `N+M-1` zu verschiedenen Zeitpunkten des Elastizitätsprotokolls. Wir beginnen mit der Betrachtung von einzelnen Placeausfällen.

Ausfälle der Places `N-2` und `N-1` werden über die in Abschitt 4.3 vorgestellten Überwachungsschemen festgestellt. Ein Ausfall von Place `N` wird, wie vorstehend beschrieben, durch Place `0` erkannt.

Place `N` erkennt einen Ausfall von `N-1` durch das Senden der synchronen Nachricht `newBack`. Sollte `N-1` nach erfolgreicher Abarbeitung von `newBack` ausfallen, so wird dies über die in Abschnitt 4.3 vorgestellten Techniken erkannt und durch das Wiederherstellungsprotokoll aus Abschnitt 4.4 behandelt.

Da das `extend`-Subprotokoll synchrone Kommunikation in Form von `at`-Blöcken verwendet, werden Ausfälle neuer Places während der Ausführung des `extend`-Subprotokolls über das Abfangen von `DeadPlaceException`s erkannt. Fallen neue Places nach der Ausführung des `extend`-Subprotokolls aus, so wird ihr Ausfall durch die in Abschnitt 4.3 vorgestellten Techniken nach Neuverteilung des Lifeline-Graphen erkannt und gemäß dem Wiederherstellungsprotokoll behandelt.

Ausfall des Places `N-2` Ein Ausfall von Place `N-2` wird durch die in Abschnitt 4.3 vorgestellten Techniken erkannt. Der Ausfall wird gemäß dem Wiederherstellungsprotokoll behandelt. Place `N-1` sendet in seiner Rolle als Backup-Place bei der Ausführung des Wiederherstellungsprotokolls ein Übernahme-Backup an seinen Backup-Place (Ⓡ3 in Abbildung 4.2). Bis zum Erhalt der entsprechenden `TOack`-Nachricht (Ⓡ4 in Abbildung 4.2) wird die weitere Ausführung des Wiederherstellungprotokolls verzögert. Für die weitere Ausführung

der Wiederherstellung ist entscheidend, zu welchem Zeitpunkt die Nachricht `TOreq` versendet bzw. vom Empfänger ausgeführt wird.

Sendet `N-1` die Nachricht `TOreq` an Place `0` und führt Place `0` die Nachricht `TOreq` vor (E1) aus, so kommt Place `0` seiner Rolle als `Back(Back(P))` nach. Führt Place `0` diese Nachricht erst nach (E1) aus, so wird Place `0` diese ignorieren und die Ausführung des Wiederherstellungsprotokolls verzögert sich bis das Elastizitätsprotokoll (E3) erreicht hat. Das über `newBack` an Place `N` gesendete Backup ersetzt das Übernahme-Backup.

Sendet `N-1` eine Nachricht `TOreq`, nachdem (E3) im Elastizitätsprotokoll erreicht wurde, so ist `N` bereits als neuer Backup-Place eingeweiht und wird das Backup entgegennehmen.

Ausfall des Places `N-1` Wie im vorangegangenen Fall wird der Ausfall über das Wiederherstellungsprotokoll behandelt. Im Laufe der Ausfallbehandlung sendet `Forth(N-1)` eine Nachricht `restore(N-1)` an Place `0`. Die `restore(...)`-Nachricht wird nicht an Place `N` gesendet, da `Forth(N-1)` noch nicht weiß, dass neue Places in die Berechnung aufgenommen wurden.

Führt Place `0` diese Nachricht vor (E1) aus, so wird die Ausführung von Phase 2 des Elastizitätsprotokolls verschoben, bis Place `0` den Notfallmodus verlässt.

Führt Place `0` die `restore(N-1)`-Nachricht zwischen (E1) und (E5) aus, so kann er sie nicht bearbeiten, aber speichern. Sobald Place `0` die Nachricht `migrated` ausführt (E5), sendet er eine Nachricht `referral(N)` an `Forth(N-1)`, um zu signalisieren, dass `N` für diese Wiederherstellung verantwortlich ist. `Forth(N-1)` sendet daraufhin eine Nachricht `restore(N-1)` an `N`.

Stellt `N` durch die synchrone Nachricht `newBack` fest, dass `N-1` ausgefallen ist, so spielt er sofort das Backup von `N-1` in seinen Taskpool ein und sendet ggf. `linkResolve`- und `GOTcheck`-Nachrichten an die Opfer und Diebe von `N-1`. Danach wird das Elastizitätsprotokoll normal weitergeführt. Die Ausführung des restlichen Wiederherstellungsprotokolls kann erst erfolgen, sobald `N` eine Nachricht `migrated` an Place `0` gesendet hat und dieser ggf. aufgezeichnete Wiederhestellungsanfragen mittels `referral(...)` beantwortet.

Erhält Place `N` eine Nachricht `restore(N-1)` und hat die Daten von `N-1` noch nicht wiederhergestellt, so kommt er seiner Tätigkeit als Backup-Place gemäß dem Wiederherstellungsprotokoll nach, muss jedoch das Senden des Übernahme-Backups ((R3) in Abbildung 4.2) bis (E6) verschieben. Solange das `extend`-Subprotokoll nicht erfolgreich beendet wurde, darf `N` kein neues Backup senden.

 doi:10.17170/kobra-2018122577

Ausfall neu hinzugefügter Places Fällt ein neuer Place vor (E1) aus, so wird sein Ausfall ignoriert. Diese Reaktion ist gerechtfertigt, da der Taskpool sowie das Teilergebnis neu initialisierter Worker leer ist. Somit können keine für die Berechnung relevanten Daten verloren gehen. Fällt `N` vor (E1) aus, so sendet Place `0` die Nachricht `startGrowth` an den nächsten lebendigen neuen Place. Fallen alle neuen Places vor (E1) aus, so bricht Place `0` die Ausführung des Elastizitätsprotokolls ab.

Fällt `N` nach (E1) und vor (E7) aus, wird Place `0` dies über die Timeout-Kontrolle feststellen und sendet eine Nachricht `startGrowth` an den nächsten lebendigen Nachfolger `N+1` von `N`. Im Gegensatz zur ursprünglichen `startGrowth`-Nachricht kann Place `0` nun nicht mehr die Backup-Daten von `N-1` mitsenden, wodurch `N+1` unbedingt die Backup-Daten von `N-1` über eine Nachricht `newBack` erhalten muss. Gelingt dies nicht, wird der Programmlauf aufgrund von endgültigem Datenverlust abgebrochen. Sollte auch `N+1` ausfallen, so sendet Place `0` eine Nachricht `startGrowth` an Place `N+2`. Dieser Vorgang wird so lange wiederholt, bis entweder der Abschluss von Phase 2 durch eine Nachricht `extendDone` bestätigt wird oder alle neuen Places ausgefallen sind. Im zweiten Fall fordert Place `0` aktuelle Backup-Daten von `N-1` und bricht die Ausführung des Elastizitätsprotokolls ab. Sollte `N-1` ausgefallen sein, so bricht Place `0` den Programmlauf aufgrund von endgültigem Datenverlust ab.

Fällt ein neuer Place $\texttt{P} \neq \texttt{N}$ während der Ausführung des `extend`-Subprotokolls aus, so wird der Ausfall sofort durch seinen Vorgänger `P-1` bemerkt. Der Vorgänger reagiert, indem er eine neue `extend`-Nachricht an den nächsten lebendigen Nachfolger von `P` sendet. Ist das `extend`-Subprotokoll schon einmal erfolgreich zu einem Place `P` zurückgekehrt, so muss es nicht nochmals ausgeführt werden, da die vorige Ausführung zugesichert hat, dass die Places `P`, `P+1`, ...,`N+M-1` erfolgreich ein initiales Backup gesendet haben. Somit wird unnötige Kommunikation vermieden. Wir erkennen diesen Zustand daran, dass der Nachfolger eines neuen Places nur dann gesetzt ist, wenn seine synchron gesendete `extend`-Nachricht wieder zu ihm zurückgekehrt ist.

Fällt ein neuer Place nach (E7) aus, so wird sein Ausfall durch die Ausführung des Wiederherstellungsprotokolls behandelt.

6. Varianten

In den Abschnitten 6.1 und 6.2 stellen wir zwei Varianten vor, welche das Stehlverhalten der Worker verändern. Unser fehlertoleranter Algorithmus lässt sich in offensichtlicher Weise auf beide Varianten übertragen. Die Varianten wurden bereits in [45] vorgestellt.

In Abschnitt 6.3 stellen wir zusätzlich eine Variante des fehlertoleranten Algorithmus vor, welche bereits in [40] veröffentlicht wurde. Wir geben ein detailliertere Beschreibung des inkrementellen Algorithmus. Statt bei der Aktualisierung der Backups den gesamten lokalen Pool eines Workers zu versenden, überträgt diese Variante, wann immer möglich, lediglich inkrementelle Änderungen.

Alle Varianten sind mit dem in Kapitel 5 vorgestellten Elastizitätsalgorithmus kompatibel.

6.1. Vorausschauendes Stehlen

Normalerweise beginnen Worker in GLB Tasks zu stehlen, sobald ihr Taskpool leer ist. Zwischen dem Versenden einer Stehlanfrage und dem Erhalt von Tasks ist der Worker ohne Arbeit.

Um den Leerlauf zu vermeiden, beginnt ein Worker in dieser Variante bereits dann Tasks zu stehlen, wenn die Anzahl der Tasks in seinem Taskpool einen Schwellwert `f` unterschreitet. Hierbei ist `f` ein Parameter, der bei Programmstart übergeben werden kann. Um die Variante nutzen zu können, muss der vom Benutzer implementierte Taskpool `queue` eine Methode `size()` zur Abfrage der aktuellen Poolgröße bereitstellen. Das Verfahren wurde von Prell [109] erstmals vorgestellt. Es ließ sich mit wenigen Änderungen auf unseren fehlertoleranten und elastischen Algorithmus übertragen.

Ein Worker erhält in der Regel Tasks, obwohl er noch eigene Arbeit hat. Diese Situation ist jedoch auch schon in GLB über Lifeline-Anfragen gegeben. Daher ist eine Anpassung unseres fehlertoleranten Algorithmus nicht notwendig.

6.2. Weiterleiten von Stehlanfragen

Diese Variante ist ebenfalls eine Adaption des Steal-Forwarding von Prell [109] auf unseren fehlertoleranten Algorithmus. Die Grundidee des Steal-Forwardings ist, dass ein Opfer `V` eine gescheiterte Stehlanfrage nicht ablehnt, sondern zu einem weiteren Opfer weiterleitet und sich selber als Dieb in diese Anfrage einträgt. Hierdurch wird anstatt zweier Nachrichten (vom Opfer zum Dieb und vom Dieb zum nächsten Opfer) nur noch eine Nachricht gesendet.

Wir setzen dieses Konzept für die zufälligen Stehlanfragen in unserem fehlertoleranten Algorithmus um. Wir können im Unterschied zu [109] aufgrund der Fehlertoleranz die Anzahl der Nachrichten nicht vermindern. Unsere Erwartung ist, dass die Anzahl der Diebe pro Stehltransaktion steigt und somit eine bessere Performance des Algorithmus erreicht wird.

Um Steal-Forwarding umzusetzen, fügen wir der Nachricht `trySteal` die ID des ursprünglichen Diebes `T`, eine Stehl-ID zur Identifikation der Stehlanfragen, eine Liste an Opfern `victims` und eine Liste zusätzlicher Diebe `additionalThieves` hinzu. Wir besprechen die einzelnen Bestandteile der Nachricht nachfolgend.

Bevor ein Dieb `T` die Stehlanfrage versendet, befüllt er die Liste `victims` mit `w` zufällig gewählten, paarweise verschiedenen Opfern $\texttt{V}_1$, ..., $\texttt{V}_\texttt{w}$. Dadurch, dass der ursprüngliche Dieb alle Opfer wählt, können wir paarweise Verschiedenheit gewährleisten, welche nachfolgend gebraucht wird. Die Liste `additionalThieves` ist zu Beginn leer. Diese Nachricht sendet `T` an das erste Opfer $\texttt{V}_1$. Von diesem Zeitpunkt an überwacht `T` das Opfer $\texttt{V}_1$ bis zum Eintreffen einer Nachricht `give` oder `noTasks` über die Timeout-Kontrolle aus Abschnitt 4.3 auf Lebendigkeit.

Erhält ein Opfer $\texttt{V}_i$ eine Nachricht `trySteal` und hat keine Arbeit, so schreibt es sich als zusätzlicher Dieb in `additionalVictims`. Ein Opfer hängt sich bei jeder `trySteal`-Anfrage, die es erhält als zusätzlicher Dieb an. Danach leitet es die Anfrage an das nächste Opfer $\texttt{V}_{i+1}$ weiter und sendet eine Nachricht `forwardedTo` mit der ID von $\texttt{V}_{i+1}$ und der Stehl-ID an `T`. Ist kein weiteres Opfer vorhanden, so sendet das letzte Opfer $\texttt{V}_\texttt{w}$ eine Nachricht `noTasks` an `V`.

Erhält ein Opfer $\texttt{V}_i$ eine Nachricht `trySteal` und kann Arbeit abgeben, so zeichnet $\texttt{V}_i$ die Stehlanfrage auf. Bei der Bearbeitung in `processRecorded()` wird Dieb `T` vor den zusätzlichen Dieben `additionalThieves` mit Tasks versorgt.

Es ist möglich, dass ein Worker $\texttt{V}_i$ mehrfach bei einem Opfer $\texttt{V}_j$ als Dieb

 doi:10.17170/kobra-2018122577

eingetragen ist. Dies wird von V_j bei der Bearbeitung der Stehlanfragen durch `processRecorded()` festgestellt und V_i erhält nur einmal Beute.

Erhält `T` eine Nachricht `forwardedTo` von V_i, so hört er auf, V_i auf Lebendigkeit zu überprüfen und überwacht stattdessen V_{i+1}.

In einem Programmlauf ohne Placeausfälle sendet ein Dieb keine zweite Stehlanfrage, solange die erste Stehlanfrage noch nicht beantwortet wurde. Fällt jedoch ein Opfer V_i nach dem Weiterleiten einer Anfrage an V_{i+1} und bevor es `forwardedTo` an `T` senden kann, aus, bemerkt Dieb `T` den Ausfall von V_i und sieht alle zufälligen Stehlanfragen als gescheitert an. Dem entsprechend beginnt `T`, Lifeline-Anfragen zu stellen. Erhält `T` über die Lifeline-Anfragen Tasks, so kann es dazu kommen, dass `T` diese Tasks abarbeitet und danach erneut mit zufälligen Stehlanfragen beginnt. Hat V_{i+1} die ursprüngliche Stehlanfrage noch nicht bearbeitet, so ist `T` in zwei zufällige Stehlanfragen involviert. Die Stehl-ID ermöglicht es festzustellen, ob eine Nachricht `forwardedTo` zur letzten gestarteten Stehlanfrage oder einer älteren Stehlanfrage gehört. Eine Überwachung des aktuellen Opfers der zuletzt gestarteten Stehlanfrage ist an dieser Stelle ausreichend, sodass ältere `forwardedTo`-Nachrichten verworfen werden. Die Stehl-ID wird von jedem Place fortlaufend, beginnend bei `0`, vergeben.

6.3. Inkrementelle Backups

Wie in Abschnitt 4.1 beschrieben, sendet jeder Worker als Teil seines Backups immer den vollständigen Taskpool mit. Bei großen Taskpools kann dies zu hohen Kommunikationskosten führen. Nachfolgend stellen wir ein inkrementelles Backup-Schema vor, das die gesendete Datenmenge und somit die Kommunikationskosten minimiert.

Wir erweitern den fehlertoleranten Algorithmus um *inkrementelles* Sichern *stabiler* Tasks. Dabei bedeutet inkrementell, dass nur jene Änderungen des Taskpools, die seit dem Senden des letzten Backups aufgetreten sind, übertragen werden. Stabil bedeutet, dass der seit dem letzten Backup minimal erreichte (am wenigsten Tasks beinhaltende) Zustand des Taskpools als Backup gesendet wird.

Zur Umsetzung dieser Variante ist es notwendig, dass der Taskpool als Dequeue implementiert wird: Die Entnahme von Tasks durch `process(n:Int)` sowie das Einfügen neu generierter Tasks müssen

am Anfang der Dequeue arbeiten, während die Methoden `split()`, `merge(bag:TaskBag)` und `merge(queue:Queue)` Tasks am Ende der Deque entnehmen bzw. einfügen. Darüber hinaus sind folgende neue Methoden zu der Taskpool-Schnittstelle `Queue` (siehe Abschnitt 3.2) hinzugekommen:

- `removeFromTop(n:Int)`, `removeFromBottom(n:Int)`: Entfernt die `n` obersten (vom Anfang) bzw. untersten (vom Ende) Tasks aus der `queue`. Werden diese Methoden mit einem Wert < 0 aufgerufen, so geben sie `null` zurück und entfernen keine Tasks.
- `getTopmostTask()`: Gibt den obersten Task des Taskpools als `Queue` zurück. Hierbei erhält die neu erzeugte `Queue` eine Kopie des momentanen Ergebnisses des Workers. Der Task wird nicht aus der ursprünglichen `Queue` entfernt.
- `getFromBottom(n:Int)`: Gibt die `n` untersten Tasks als `Queue` zurück, ohne sie aus der ursprünglichen `Queue` zu löschen. Wird diese Methode mit einem Wert < 0 aufgerufen, so gibt sie `null` zurück und entfernt keine Tasks.
- `pushBottom(q:Queue)`: Fügt die übergebene Queue `q` am Ende der aktuellen `Queue` ein.
- `size()`: Gibt die aktuelle Anzahl an Tasks in der `Queue` zurück.

Wir erläutern die Funktionsweise der inkrementellen Variante zunächst anhand der regulären Backups in Abschnitt 6.3.1. Die Erweiterung auf stehl- und wiederherstellungsinduzierte Backups besprechen wir in Abschnitt 6.3.2.

6.3.1. Reguläre Backups

Wir gehen vorerst davon aus, dass zwischen zwei aufeinander folgenden Backup-Zeitpunkten $t-1$ und t weder Tasks empfangen noch gestohlen werden.

Abbildung 6.1 illustriert das inkrementelle Backup-Schema. Das Schema ermittelt den *minimalen Zustand* des Taskpools in dem Zeitintervall zwischen zwei Backups. Hierunter verstehen wir den Pool mit der geringsten Anzahl an Tasks. Dazu wird vor jedem Aufruf von `queue.process(...)` die Größe des Pools abgefragt. Zusätzlich wird die Methode `queue.process(...)` in der vorgestellten Variante `n`-Mal mit dem Parameter `1` aufgerufen anstatt einmalig

 doi:10.17170/kobra-2018122577

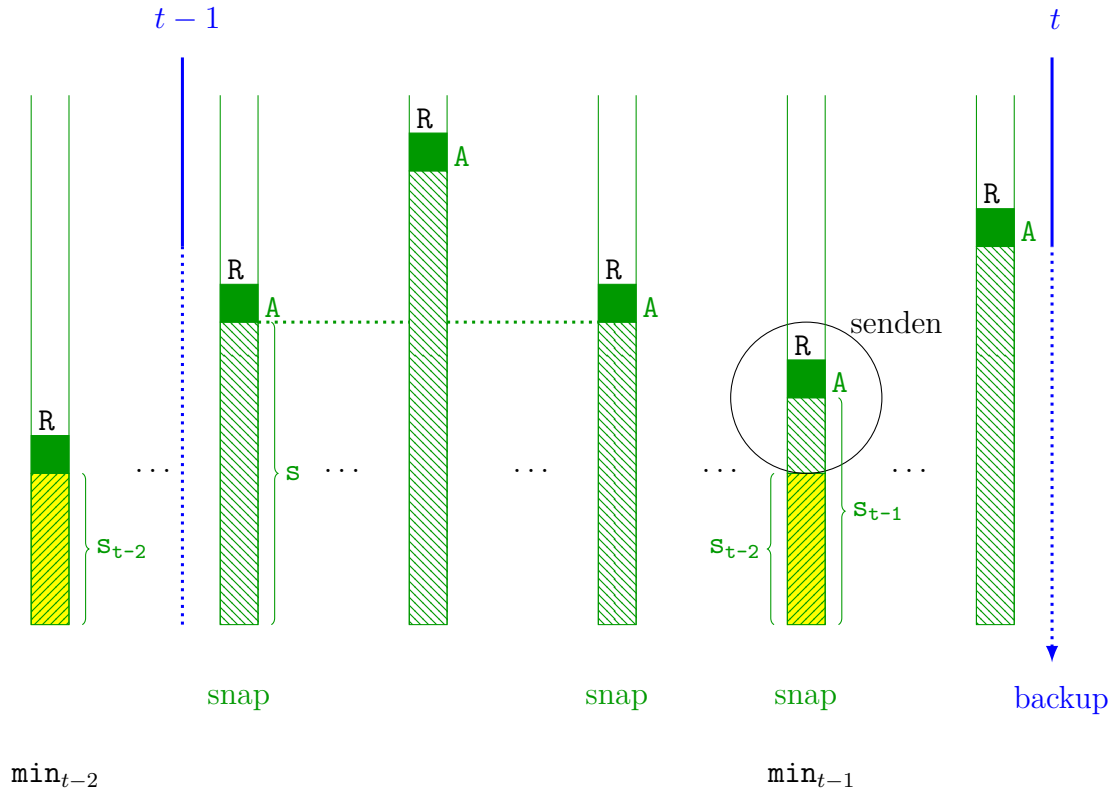


Abbildung 6.1.: Inkrementelles Backupschema (siehe [40])

mit dem Parameter `n`. Nur so ist eine lückenlose Beobachtung der Poolgröße möglich.

Wird ein neuer Minimalzustand des lokalen Pools erkannt, so wird ein *Snapshot* (kurz „snap“, deutsch: Schnappschuss) des aktuellen Poolzustandes angelegt. Dabei wird der oberste Task `A` des Pools, die aktuelle Poolgröße $\texttt{min}_{t-1}$ sowie eine Kopie des lokalen Ergebnisses abgespeichert. Ein Snapshot wird auch dann erstellt, wenn der momentane Pool die Größe $\texttt{min}_{t-1}$ hat. Nach dem Senden eines Backups, also vor dem ersten Aufruf von `queue.process(1)` wird ebenfalls immer ein Snapshot gemacht. Dieser dient als Basis für die weitere Minimalzustandsbetrachtung. Quellcodeabbildung 6.1 zeigt den Pseudocode für die Snapshotroutine. Wir weisen an dieser Stelle darauf hin, dass `A` vom Typ `Queue` ist und der Aufruf `queue.getTopmostTask()` eine Kopie des Workerergebnisses anlegt und mit `A` zurückgibt.

```
1 if (afterBackup || queue.size() <= min) {
2   A = queue.getTopmostTask();
3   min = queue.size();
4   afterBackup = false;
5 }
```

Quellcodeabbildung 6.1: Snapshotroutine

Ist der nächste Backup-Zeitpunkt erreicht, wird aus dem Snapshot der zu sendende Pool rekonstruiert. Dies geschieht abhängig von der Größe $\texttt{min}_{t-1}$ des Snapshots sowie der Größe $\texttt{min}_{t-2}$ des letzten Backups. Ziel ist es, nur jene Tasks zu übertragen, die noch nicht im Backup gespeichert sind. Hier lassen sich zwei generelle Fälle unterscheiden: Der rekonstruierte Pool ist mindestens so groß wie das gespeicherte Backup ($\texttt{min}_{t-1} \geq \texttt{min}_{t-2}$) oder der zu sendende Pool ist kleiner als das Backup ($\texttt{min}_{t-1} < \texttt{min}_{t-2}$). Da das Schema die Größe des Pools nach der Bearbeitung jedes Tasks überprüft, können wir sicher sein, dass die untersten Tasks in beiden Pools identisch sind (dargestellt durch die gelb schraffierten Flächen in Abbildung 6.1). Der erste Task, in dem sich die Pools unterscheiden, ist der oberste Task des kleineren Pools.

Ist der zu sendende Pool mindestens so groß wie das Backup, müssen die untersten $\texttt{min}_{t-2} - 1$ nicht mitgesendet werden und werden aus dem zu sendenden Pool über den Aufruf `removeFromBottom(...)` entfernt.

Ist der zu sendende Pool kleiner als das Backup, so reicht es, die Anzahl $\texttt{removeFromTop} = \texttt{min}_{t-2} - \texttt{min}_{t-1} + 1$ der von oben zu entfernenden Tasks sowie der oberste Task `A` des Snapshots zu übertragen.

6.3.2. Stehl- und wiederherstellungsinduzierte Backups

Wir beginnen die Betrachtung mit den stehlinduzierten Backups auf der Seite des Opfers. Da wir fordern, dass `queue.split()` Tasks vom Ende des Pools entnimmt und die Klasse `TaskBag` ebenfalls eine `size()`-Methode besitzt, kann ein Worker bei der Formung einer Stehltransaktion mitzählen, wie viele Tasks `stolen` von unten aus dem Taskpool entnommen wurden. Das zu sendende Stehl-Backup hängt in dem Fall von den Werten `stolen`, $\texttt{min}_{t-1}$ und $\texttt{min}_{t-2}$ ab:

1. $\texttt{stolen} \geq \texttt{min}_{t-1} \wedge \texttt{stolen} \geq \texttt{min}_{t-2}$: Es werden alle im Backup gespeicherten sowie alle im Snapshot gespeicherten Tasks gestohlen. Es kann kein inkrementelles Backup gesendet werden. Der aktuellen Pool

doi:10.17170/kobra-201812257

nach Herauslösen der Beute wird als nicht-inkrementelles Stehl-Backup gesendet.

2. $\texttt{stolen} < \texttt{min}_{t-1} \wedge \texttt{stolen} \geq \texttt{min}_{t-2}$: Es werden zwar alle Tasks aus dem Backup, jedoch nicht aus dem Snapshot gestohlen. Verringere $\texttt{min}_{t-1}$ um `stolen` und rekonstruiere den Taskpool aus $\texttt{min}_{t-1}$, entferne dabei keine Tasks vom Ende der Queue. Dieser Taskpool wird als nicht-inkrementelles Stehl-Backup gesendet.

3. $\texttt{stolen} \geq \texttt{min}_{t-1} \wedge \texttt{stolen} < \texttt{min}_{t-2}$: Es werden alle Tasks aus dem Snapshot, jedoch nicht alle Tasks aus dem Backup gestohlen. Protokolliere den Zustand nach dem Herauslösen der Beute als neuen minimalen Zustand. Der aktuelle Taskpool wird als nicht-inkrementelles Stehl-Backup gesendet.

4. $\texttt{stolen} < \texttt{min}_{t-1} \wedge \texttt{stolen} < \texttt{min}_{t-2}$: Es werden weder alle Tasks aus dem Snapshot noch alle Tasks aus dem Backup gestohlen. Die Anzahl der gestohlenen Tasks `stolen` wird als inkrementelles Stehl-Backup gesendet, $\texttt{min}_{t-1}$ wird um `stolen` verringert.

Die Stehl-Backups im 3. und 4. Fall bestehen aus einer einzigen Zahl. Erhält ein Worker in seiner Rolle als Backup-Place ein solches Stehl-Backup, so entfernt er `stolen`-viele Tasks von unten aus dem gespeicherten Backup durch den Aufruf `backup.removeFromBottom(stolen)`.

Wir betrachten nun, wie sich der Empfang von Beute auf das Schreiben inkrementeller Backups auswirkt. Aus der Betrachtung des Stehlprotokolls wissen wir, dass ein Dieb `T`, der Beute über eine Nachricht `give` erhält, eine `victimLink`-Nachricht an seinen Backup-Place `Back(T)` sendet. Mit dem Erhalt des nächsten Backup von `T` löscht `Back(T)` den Link und sendet eine Nachricht `delOpen` an das Opfer um zu signalisieren, dass die Beute im Backup von `T` enthalten ist. Um diese Bedingung aufrechtzuerhalten, wird das nächste Backup von `T` nach dem Erhalt von Beute immer ein nicht-inkrementelles Backup sein. Dies gilt insbesondere, wenn das nächste Backup ein inkrementelles Stehl-Backup ist.

Übernahme-Backups werden immer als nicht-inkrementelle Backups gesendet. Der Prozess des Einspielens eines Backups in den Taskpool gleicht im Wesentlichen dem Einspielen von Beute.

Für Einweihungs-Backups kann kein inkrementelles Backup gesendet werden, da der Backup-Place noch keine Backup-Daten vorhält. Somit wird dieses Backup auch immer als nicht-inkrementelles Backup gesendet.

 doi:10.17170/kobra-2018122577

7. Erweiterung der MPI-Netzwerkschnittstelle von X10 um Elastizität

In X10 wird die Netzwerkkommunikation durch die Implementierung einer Schnittstelle realisiert. Native X10 bietet eine Implementierung dieser Schnittstelle mit Hilfe des in Abschnitt 1.1 erwähnten MPI-Standards und ULFM in der Programmiersprache C++ an. Allerdings unterstützt diese Implementierung keine Elastizität. Wir haben sie deshalb entsprechend erweitert. Die Erweiterung wurde bereits in Form eines Posters [44] präsentiert.

In Abschnitt 7.1 besprechen wir die grundlegende Funktionsweise von MPI. Insbesondere gehen wir auf die MPI-Konstrukte ein, die für die Erweiterung benötigt werden. Auch Abschnitt 7.2 führt Hintergrundwissen ein. Dieser Abschnitt behandelt die X10-Netzwerkschnittstelle und ihre bisherige MPI-Implementierung. Unsere Änderungen an der MPI-Schnittstelle sind der Inhalt von Abschnitt 7.3. Wir zeigen die zu lösenden Probleme auf und erklären anhand dieser unsere Erweiterung.

7.1. MPI

Wie in Abschnitt 1.1 erwähnt, ist MPI der im HPC-Bereich verbreitete Standard zur Realisierung von nachrichtenbasierter Kommunikation auf Systemen mit verteiltem Speicher. In der aktuellen Version 3.1 unterstützt MPI keine Fehlertoleranz gegenüber permanenten Prozessausfällen. Stürzt ein Prozess ab, wird die gesamte Programmausführung abgebrochen. ULFM erweitert den MPI-Standard entsprechend, um Prozessausfälle zu tolerieren. Wir gehen am Ende dieses Abschnitts näher auf ULFM ein.

Der MPI-Standard beschreibt eine Menge von Bibliotheksfunktionen. Er wurde in verschiedenen Bibliotheken, beispielsweise Open MPI [12] und

MPICH [13], implementiert.

MPI operiert auf einer niedrigeren Abstraktionsschicht als X10. Datenaustausch wird über das Senden und Empfangen von Nachrichten umgesetzt, ein globaler Speicher existiert nicht.

Beim Start eines MPI-Programms werden gleichzeitig alle Prozesse gestartet. Nach Programmstart führt jeder Prozess die `main`-Methode des Programms aus. Dies bildet einen wesentlichen Unterschied zu X10, wo die Ausführung mit der Wurzelaktivität auf Place `0` beginnt.

Bevor ein Prozess mittels MPI kommunizieren kann, muss er die Funktion `MPI_Init(...)` oder `MPI_Init_thread(...)` aufrufen. Vor Programmende muss jeder Prozess die Funktion `MPI_Finalize()` ausführen. Nach Aufruf von `MPI_Finalize()` kann dieser Prozess keine Kommunikationsmethoden von MPI mehr verwenden. Bei Verwendung von `MPI_Init_thread(...)` kann der Nutzer eine Zusicherung über den verwendeten Threading-Modus, also die parallele Ausführung von Kommunikationsroutinen innerhalb eines MPI-Prozesses, machen. Die folgenden Modi stehen zur Verfügung:

- `MPI_THREAD_SINGLE`: Es wird nur ein einziger Thread pro Prozess genutzt.
- `MPI_THREAD_FUNNELED`: Nur der Thread, der `MPI_Init_thread(...)` aufgerufen hat, ruft Kommunikationsroutinen auf.
- `MPI_THREAD_SERIALIZED`: Zu einem Zeitpunkt führt höchstens ein Thread pro Prozess Kommunikationsroutinen aus.
- `MPI_THREAD_MULTIPLE`: Zu einem Zeitpunkt können mehrere Threads pro Prozess Kommunikationsroutinen ausführen.

Für die Einhaltung der getroffenen Zusicherung ist der Nutzer verantwortlich.

Sende- und Empfangsoperationen stehen in mehreren Varianten zur Verfügung. **Blockierende Kommunikation** wird meist über die Funktionen `MPI_Send(...)` und `MPI_Recv(...)` realisiert. Hier blockiert ein Aufruf von `MPI_Send(...)`, bis die entsprechende Nachricht entweder im Speicher der Netzwerkkarte gepuffert oder durch ein entsprechendes `MPI_Recv(...)` empfangen wurde. Das genaue Verhalten ist implementierungsspezifisch. Ein Aufruf von `MPI_Recv(...)` blockiert, bis eine entsprechende Nachricht empfangen wurde.

 doi:10.17170/kobra-2018122577

Nichtblockierende Kommunikation nutzt die Funktionen `MPI_Isend(...)` und `MPI_Irecv(...)`, welche nach dem Aufruf sofort zurückkehren. Als Rückgabewert liefern diese Funktionen einen `MPI_Request`. Dieser wird genutzt, um später den Status zu überprüfen. Hierzu stellt MPI unter anderem die Funktion `MPI_Test(...)` zur Verfügung. Diese Funktion liefert `true` zurück, falls die entsprechende Sende- bzw. Empfangsoperation abgeschlossen wurde, und `false` sonst. Eine über `MPI_Isend(...)` gestartete Sendeoperation gilt als abgeschlossen, wenn die Nachricht entweder gepuffert oder vom Sender empfangen wurde. Eine über `MPI_Irecv(...)` gestartete Empfangsoperation gilt als abgeschlossen, wenn eine entsprechende Nachricht empfangen wurde.

Die Funktion `MPI_Iprobe(...)` überprüft, ob eine Nachricht zum Empfang vorliegt.

Neben `MPI_Isend(...)` bietet MPI die Funktionen `MPI_Ibsend(...)` und `MPI_Issend(...)`, um nichtblockierende Nachrichten zu versenden. Das zusätzliche `b` steht für `buffered`, das zusätzliche `s` steht für `synchronous`. Dabei bedeutet `buffered`, dass der Sendevorgang als abgeschlossen gilt, wenn die Nachricht erfolgreich gepuffert oder empfangen wurde. `Synchronous` bedeutet, dass der Sendevorgang nur als abgeschlossen gilt, wenn auf der Empfängerseite ein entsprechendes `MPI_Irecv(...)` aufgerufen wurde und somit die Zustellung der Nachricht garantiert erfolgt.

Neben dem paarweisen Nachrichtenaustausch über `MPI_Send(...)` und `MPI_Recv(...)` unterstützt MPI **kollektive Operationen**, bei denen mehr als zwei Prozesse an einem Nachrichtenaustausch teilnehmen können.

Die Funktionen `MPI_Recv(...)` und `MPI_Irecv(...)` erwarten als Parameter den `rank` des Senders bzw. Empfängers. Es kann der Wert `MPI_ANY_SOURCE` übergeben werden. So gestartete Empfangsoperationen nehmen eine Nachricht von einem beliebigen Sender entgegen.

Kollektive Operationen werden durch spezielle Funktionen, beispielsweise `MPI_Bcast(...)`, realisiert. Auch hier gibt es blockierende und nichtblockierende Varianten, zum Beispiel `MPI_Bcast(...)` und `MPI_Ibcast(...)`.

Alle Kommunikationsmethoden erwarten als Parameter einen *Kommunikator*. Darunter versteht man eine verteilte Datenstruktur zur Beschreibung der gegenseitigen Erreichbarkeit einer Gruppe von Prozessen.

MPI ordnet jedem Prozess innerhalb eines Kommunikators eine eindeutige Nummer, den `rank`, zu. Die `rank`s eines Kommunikators sind kompakt, d. h. in einem Kommunikator mit `N` Prozessen haben diese die `rank`s von `0` bis `N-1`.

Ein Prozess kann in verschiedenen Kommunikatoren verschiedene `rank`s haben. Die Funktion `MPI_Comm_rank(...)` erlaubt es, den `rank` eines Prozesses in einem Kommunikator abzufragen.

Zu Programmstart eines MPI-Programms existiert ein einziger Kommunikator welcher in der Konstante `MPI_COMM_WORLD` gespeichert ist.

Zur Erstellung eines neuen Kommunikators ist immer mindestens ein bereits existierenden Kommunikator erforderlich. Die Operationen zum Erstellen neuer Kommunikatoren sind blockierend und kollektiv über alle Prozesse des/der zur Erstellung genutzten Kommunikators/Kommunikatoren.

MPI bietet drei Möglichkeiten, neue Kommunikatoren zu erstellen. Mit `MPI_Comm_split(...)` und ähnlichen Funktionen können aus einem Kommunikator mehrere neue Kommunikatoren erzeugt werden. Die Anzahl der erzeugten Kommunikatoren sowie die Zuordnung der Prozesse zu den jeweiligen Kommunikatoren wird mit Übergabeparametern gesteuert.

Die zweite Möglichkeit basiert auf der Nutzung von *Gruppen*. Eine Gruppe repräsentiert die Prozesse eines Kommunikators. Über die Funktion `MPI_Comm_group(...)` kann die Gruppe eines Kommunikators ermittelt werden. Mit Hilfe der Funktion `MPI_Comm_create(...)` wird aus einem alten Kommunikator und einer Gruppe ein neuer Kommunikator erzeugt. Die übergebene Gruppe muss dabei eine echte Teilmenge der Gruppe des alten Kommunikators sein. Die Funktion `MPI_Comm_create(...)` ist blockierend und kollektiv über alle Prozesse des alten Kommunikators. MPI definiert eine Sammlung von Funktionen, wie beispielsweise `MPI_Group_union(...)` und `MPI_Group_intersection(...)`, um aus bestehenden Gruppen neue Gruppen zu erstellen. Diese Operationen arbeiten lokal und sind somit weder kollektiv noch blockierend.

Die dritte Möglichkeit zur Erstellung von Kommunikatoren basiert auf den Funktionen `MPI_Open_port(...)`, `MPI_Comm_accept(...)` und `MPI_Comm_connect(...)`. Diese erlauben es, eine Verbindung zwischen zwei separat gestarteten MPI-Programmen herzustellen. Ein Prozess aus einem der Programme öffnet einen Netzwerkport durch `MPI_Open_port(...)`. Wir nennen das entsprechende Programm nachfolgend *Server*. Der Server ruft im Anschluss `MPI_Comm_accept(...)` auf. Das andere Programm, der *Client*, verbindet sich zum geöffneten Port des Servers mit `MPI_Comm_connect(...)`. Die Aufrufe `MPI_Comm_accept(...)` und `MPI_Comm_connect(...)` sind kollektiv über alle Prozesse des Server- bzw. Client-Programms. Mit diesen Aufrufen wird ein neuer *Interkommunikator* erstellt. Interkommunikatoren dienen der

 doi:10.17170/kobra-2018122577

Kommunikation zwischen zwei disjunkten Prozessgruppen. Die bisher vorgestellten Kommunikatoren werden dagegen als *Intrakommunikatoren* bezeichnet, da sie die Kommunikation zwischen Prozessen innerhalb einer Gruppe realisieren. Reden wir im weiteren Verlauf dieses Kapitels von Kommunikatoren, so meinen wir **Intra**kommunikatoren.

In einem Interkommunikator existieren zwei Gruppen: Lokale Prozesse und entfernte Prozesse. In beiden Gruppen starten die `rank`s bei jeweils bei `0` und sind kompakt. Wird für eine Kommunikationsfunktion ein Interkommunikator verwendet und beispielsweise eine Nachricht an `rank 0` gesendet, so wird diese Nachricht an `rank 0` der entfernten Gruppe gesendet. An lokale Prozesse kann über einen Interkommunikator keine Nachricht gesendet werden.

Die Funktionen `MPI_Comm_accept(...)` und `MPI_Comm_connect(...)` sind blockierend über alle Server- und Client-Prozesse, kehren also erst dann zurück, wenn auf allen Server-Prozessen `MPI_Comm_accept(...)` und auf allen Client-Prozessen `MPI_Comm_connect(...)` aufgerufen wurde. Zudem ist für `MPI_Comm_accept(...)` und `MPI_Comm_connect(...)` kein Timeout definierbar. Werden die Funktionen aufgerufen, ohne dass ein entsprechender Gegenaufruf stattfindet, so blockieren sie also für immer.

Mit der Funktion `MPI_Intercomm_merge(...)` wird aus einem Interkommunikator ein Intrakommunikator erzeugt. Die Parameter der Funktion steuern, welche Prozesse im neuen Kommunikator die niedrigeren `rank`s erhalten. Auch dieser Aufruf ist blockierend und kollektiv über alle Prozesse des Interkommunikators.

Wir haben eingangs erwähnt, dass der MPI-Standard den Ausfall von Prozessen nicht unterstützt. Um Fehlertoleranz zu realisieren, nutzt X10 die **ULFM-Erweiterung** des MPI-Standards. Dabei steht ULFM für User Level Failure Mitigation, das heißt, ULFM stellt Funktionalitäten zur Erkennung und Behebung von Prozessausfällen bereit. Wir stellen die Grundkonzepte nachfolgend kurz vor.

Stellt der Aufruf einer Kommunikationsmethode fest, dass mindestens ein an der Kommunikation beteiligter Prozess ausgefallen ist, gibt die Funktion den Fehlercode `MPI_ERR_PROC_FAILED` zurück.

Über einen Kommunikator, in dem mindestens ein Prozess ausgefallen ist, können keine kollektiven Operationen mehr ausgeführt werden. ULFM bietet eine Funktion `MPIX_Comm_revoke(...)` an, um einen Kommunikator zurückzuziehen. Alle darauf folgenden, nicht-lokalen Operationen auf diesem Kommunikator auf allen Prozessen des übergebenen Kommunikators

geben nach diesem Aufruf den Fehlercode `MPI_ERR_REVOKED` zurück. Insbesondere gibt die Funktion `MPI_Iprobe(...)` für `Requests`, die den entsprechenden Kommunikator verwenden, auch den entsprechenden Fehlercode zurück. Die Ausnahme bilden die Funktionen `MPIX_Comm_shrink(...)`, `MPIX_Comm_agree(...)` und `MPIX_Comm_iagree(...)`, deren Aufruf nicht zur Rückgabe von `MPI_ERR_REVOKED` führt. Der Aufruf von `MPIX_Comm_revoke` ist nicht lokal und nicht kollektiv.

Die Funktion `MPIX_Comm_shrink(...)` erstellt aus einem Kommunikator mit ausgefallenen Prozessen einen neuen intakten Kommunikator. Dieser enthält nur die lebendigen Prozesse aus dem alten Kmmunikator. Auch im neuen Kommunikator sind die `ranks` der Prozesse aufeinander folgend, das heißt, Lücken werden geschlossen. Damit erhalten einige Prozesse im geschrumpften Kommunikator ggf. einen anderen `rank`. Hierzu ein Beispiel: Fällt in einem Kommunikator mit fünf Prozessen der Prozess mit `rank 1` aus, so wird im geschrumpften Kommunikator der Prozess mit `rank 2` den `rank 1` erhalten.

ULFM Version 1.1 unterstützt den Threading-Modus `MPI_THREAD_MULTIPLE` nicht. Ab Version 2 wird auch dieser unterstützt. Die MPI-Netzwerkschnittstelle wurde allerdings auf Basis von ULFM Version 1.1 entwickelt und nutzt daher bei Verwendung von ULFM den Threading-Modus `MPI_THREAD_SERIALIZED`.

7.2. Aufbau der MPI-Netzwerkschnittstelle von X10

Dieser Abschnitt beschreibt zum einen die von der X10-Implementierung definierte abstrakte Netzwerkschnittstelle und zum anderen ihre mit X10 ausgelieferte Umsetzung mittels MPI. Es gibt keine dedizierte ULFM-Umsetzung dieser Schnittstelle, sondern zur Übersetzungszeit wird die genutzte MPI-Implementierung ermittelt (Open MPI, MPICH oder ULFM) und das Verhalten der Umsetzung entsprechend angepasst. Hamouda *et al.* [50] haben die MPI-Schnittstelle unter Verwendung von ULFM bereits um Fehlertoleranz erweitert. Wir betrachten die MPI-Umsetzung der Schnittstelle unter der Voraussetzung, dass ULFM als MPI-Implementierung genutzt wird. Wie in Abschnitt 7.1 erwähnt, verwendet die MPI-Umsetzung der Netzwerkschnittstelle den Threading-Modus `MPI_THREAD_SERIALIZED`, wenn ULFM genutzt wird.

Die für die weitere Erklärung relevanten Methoden der Netzwerkschnittstelle sind:

 doi:10.17170/kobra-2018122577

- `x10rt_net_probe()`: Überprüft ob neue Nachrichten für den aufrufenden Place vorliegen und beantwortet diese gegebenenfalls. Überprüft zusätzlich, ob gesendete Nachrichten übermittelt wurden. Diese Methode wird regelmäßig durch die X10-Laufzeitumgebung aufgerufen.

- `x10rt_net_send(p, msg)`: Sendet eine Nachricht `msg` an Place `p`.

- `x10rt_net_send_get(p, data)`: Fordert die Daten `data` von Place `p` an. Diese Methode wird genutzt, um entfernte Speicherzugriffe zu realisieren.

Die MPI-Realisierung dieser Schnittstelle bildet einen Place auf einen MPI-Prozess ab. Die `rank`s der MPI-Prozesse werden hierbei als Place-IDs genutzt. Kommunikation zwischen zwei Places erfolgt ausschließlich mittels `MPI_Isend(...)` und `MPI_Irecv(...)`. Innerhalb von `x10rt_net_probe()` werden die Funktionen `MPI_Iprobe(...)` und `MPI_Test(...)` verwendet, um den Status von eingegangenen und gesendeten Nachrichten zu überprüfen.

Ein `Team` in X10 wird über einen separaten Kommunikator realisiert. Für kollektive Operationen über `Team`s (siehe Abschnitt 3.1) sind weitere Methoden in der Schnittstelle definiert.

Jeder MPI-Prozess pflegt eine Datenstruktur `global_state`, die folgende Daten enthält:

- Eine Liste `free_req` ungenutzter `MPI_Request`s. Die Liste wird zu Programmstart mit `MPI_Request`s gefüllt. Anstatt für jede Sende- und Empfangsoperation einen neuen `MPI_Request` zu verwenden, werden `MPI_Request`s aus dieser Liste entnommen und wiederverwendet.

- Zwei Listen `pending_sends` und `pending_recvs` ausstehender `MPI_Request`s. Wird `MPI_Isend(...)` oder `MPI_Irecv(...)` aufgerufen, wird der entsprechende `MPI_Request` in die entsprechende Liste eingefügt. Abgeschlossene `MPI_Request`s werden durch eine Ausführung von `MPI_Test(...)` innerhalb der Methode `x10_net_probe()` aus der entsprechenden Liste entfernt und wieder zu `free_req` hinzugefügt.

- Einen Kommunikator `comm`, der alle Prozesse (Places) umfasst und für alle Aufrufe von `MPI_Isend(...)` und `MPI_Irecv(...)` genutzt wird.

- Eine Liste `failed_ranks` aller ausgefallenen `rank`s in `comm`.

Im weiteren Verlauf dieses Kapitels werden wir den Präfix `global_state` als implizit annehmen, wenn wir über Daten aus dieser Datenstruktur sprechen.

Die Implementierung der Methode `x10rt_net_probe()` prüft mittels `MPI_Iprobe(...)`, ob neue Nachrichten zum Empfang bereitstehen. Für jede bereitstehende Nachricht wird ein `MPI_Request` aus `free_req` entnommen und mit diesem `MPI_Irecv(...)` aufgerufen. Die zum Empfangen verwendeten `MPI_Request`s werden, wie bereits erwähnt, in `pending_recvs` eingefügt.

Darauf folgend wird für jeden `MPI_Request` in `pending_recvs` wiederholt die Methode `MPI_Test(...)` aufgerufen. Falls eine Nachricht eingegangen ist, wird der `MPI_Request` aus `pending_recvs` entfernt. Bedarf die Nachricht einer Antwort, wird sofort eine entsprechende Nachricht mittels `MPI_Isend(...)` versendet. Der `MPI_Request`, der zum Empfangen der Nachricht genutzt wurde, wird hierbei wiederverwendet und in die Liste `pending_sends` aufgenommen. Bedarf es keiner Antwort, so wird der zum Empfang genutzte `MPI_Request` wieder in die Liste `free_reqs` eingefügt. Die Liste `pending_sends` der ausstehenden Sendeoperationen wird in gleicher Weise bearbeitet.

Die Methode `x10rt_net_send(p, msg)` ist mit Hilfe von `MPI_Isend(...)` realisiert.

Für die Realisierung von `x10_net_send_get(p, data)` werden zwei Nachrichten ausgetauscht:

1. Der anfragende Place sendet eine Datenanfrage mittels `MPI_Isend(...)` an den Ziel-Place.

2. Sobald der Ziel-Place diese Anfrage durch einen Aufruf von `x10rt_net_probe()` erhält, sendet er die angeforderten Daten über einen Aufruf von `MPI_Isend(...)` an den anfragenden Place.

3. Stellt ein Aufruf von `x10rt_net_probe()` auf dem anfragenden Place fest, dass die Nachricht aus 1. gesendet wurde, wird ein entsprechendes `MPI_Irecv(...)` zum Empfang der erwarteten Daten ausgeführt. Hierbei wird der `MPI_Request` aus 1. wiederverwendet.

4. Der Datenaustausch endet mit dem Empfang der angefragten Daten auf dem anfragenden Place. Mit erfolgreichem Erhalt der Daten wird der `MPI_Request`, der zuerst in 1. und dann in 3. genutzt wurde, wieder in `free_req` eingefügt.

 doi:10.17170/kobra-2018122577

Wir haben in Abschnitt 7.1 besprochen, wie der Aufruf `MPIX_Comm_shrink(...)` arbeitet. Muss der Kommunikator `comm` geschrumpft und ausgetauscht werden, können sich die `rank`s der Prozesse ändern und stimmen nicht mehr mit den Place-IDs überein. Die Liste `failed_ranks` wird benötigt um die `rank`s auf Place-IDs umzurechnen.

7.3. Erweiterung um Elastizität

Neue MPI-Prozesse werden gestartet, indem das Programm erneut mittels `mpirun` gestartet wird. Über die Umgebungsvariable `X10_JOIN_EXISTING` wird dem neu gestarteten Programm mitgeteilt, zu welchem bereits laufenden Programm es eine Verbindung aufbauen soll. Dieses Vorgehen ist der bereits existierenden, elastischen Socket-Implementierung für Native X10 nachempfunden.

Die Grundidee unserer Erweiterung besteht darin, dass wir aus den Kommunikator `comm` aus `global_state` einen um die neuen Prozesse erweiterten Kommunikator erstellen. Unsere Erweiterung nutzt die in Abschnitt 7.1 eingeführten Methoden `MPI_Open_port(...)`, `MPI_Comm_accept(...)`, `MPI_Comm_connect(...)` sowie `MPI_Intercomm_merge(...)`, um einen neuen Kommunikator `newComm` zwischen bereits laufenden und neu gestarteten Programmen zu erstellen. Dieser Kommunikator enthält alle alten und neuen Prozesse. Genauer gesagt ruft das bereits laufende Programm die Methode `MPI_Comm_accept(...)` auf, während das neu gestartete Programm `MPI_Comm_connect(...)` aufruft. Wir bezeichnen somit das bereits laufende Programm als *Server* und das neu gestartete Programm als *Client*.

Zur Realisierung der Elastizität müssen die folgenden Probleme gelöst werden:

1. Wir wissen aus Abschnitt 7.1, dass ein Aufruf von `MPI_Comm_accept(...)` blockiert, bis ein entsprechender Aufruf von `MPI_Comm_connect(...)` erfolgt. Aus Abschnitt 7.2 wissen wir, dass der Threading-Modus `MPI_THREAD_SERIALIZED` verwendet wird. Die initiale Verbindung zwischen Client und Server kann daher nicht mittels `MPI_Comm_connect(...)` und `MPI_Comm_accept(...)` realisiert werden, da alle Server-Prozesse bei Programmstart `MPI_Comm_accept(...)` aufrufen müssten, um auf eingehende Clients reagieren zu können. Diese Aufrufe blockieren, bis sich ein Client verbindet, und da

`MPI_THREAD_SERIALIZED` als Threading-Modus genutzt wird, dürfen keine anderen Threads in diesem Zeitraum MPI-Routinen ausführen. Daher wird eine Alternativlösung für die initiale Verbindung zwischen Client und Server benötigt.

2. Im Zuge der elastischen Erweiterung muss mittels `MPI_Comm_accept(...)`, `MPI_Comm_connect(...)` und `MPI_Intercomm_merge(...)` aus dem Kommunikator `comm` aus `global_state` ein Kommunikator `newComm` erstellt werden, der auch die neu hinzugefügten MPI-Prozesse enthält. Zudem muss `comm` durch `newComm` ersetzt werden. Dies muss zu einem Zeitpunkt geschehen, zu dem über `comm` keine Nachrichten mehr gesendet und empfangen werden, da über `comm` gesendete Nachrichten nicht über `newComm` empfangen werden können. Wir bezeichnen den Zustand, in dem keine Nachrichten mehr über `comm` zu empfangen sind, als *stillen Zustand*.

Da X10 kollektive Operationen nur auf `Team`s zulässt und `Team`s separate Kommunikatoren nutzen, müssen wir kollektive Operationen nicht betrachten. Die `Team`s werden durch das Hinzufügen neuer Prozesse nicht beeinflusst.

Unsere Erweiterung löst das erste Problem, indem sie auf Prozess `0` der Server-Seite einen dedizierten *Horch-Thread* startet, welcher während der gesamten Programmlaufzeit läuft und erst endet, wenn alle X10-Aktivitäten enden. Der Horch-Thread öffnet einen MPI-unabhängigen Netzwerkport und hört diesen regelmäßig mit einem Timeout ab. Da der Netzwerkport unabhängig von MPI ist, kann das Abhören des Ports parallel zu MPI-Kommunikationsroutinen erfolgen. Durch den Timeout kann der Horch-Thread in regelmäßigen Abständen überprüfen, ob das Programm beendet wurde.

Ein Client verbindet sich zu einem Server, indem er an den geöffneten Port des Servers eine Nachricht mit seiner Netzwerkadresse sendet. Wir bezeichnen dies nachfolgend als `Hello`-Nachricht.

Geht eine `Hello`-Nachricht auf dem Netzwerkport ein, so wird die empfangene Netzwerkadresse in eine neue globale Variable `global_state.joining` geschrieben. Auf die Auswertung dieser Variable gehen wir später ein.

Prozess `0` des Server-Programms liest die Portnummer des zu öffnenden Ports aus der Umgebungsvariablen `X10_FORCEPORTS`. Um einem Programm mitzuteilen, dass es sich als Client mit einem Server verbinden soll, muss vor dem Start des Programms die Umgebungsvariable `X10_JOIN_EXISTING` auf einen Wert der Form `(IP-Adresse|Hostname):port` gesetzt werden. Dieser

 doi:10.17170/kobra-2018122577

Wert stellt die Verbindungsinformationen zu Prozess `0` des Servers dar und wird von Server-Prozess `0` zu Programmstart ausgegeben. Ist dieser Wert nicht gesetzt, verbindet sich der Client mit keinem Server und startet die eigentliche Berechnung. Ist der Wert falsch gesetzt (entweder weil der Inhalt nicht der erwarteten Form entspricht oder weil auf dem angegebenen Port keine Verbindung möglich ist), terminiert das Client-Programm.

Zur Lösung des zweiten Problems haben wir ein Protokoll entwickelt, welches in Abbildung 7.1 dargestellt ist. Das Protokoll dient dem Erreichen des stillen Zustands sowie der Konstruktion von `newComm` und dem Austausch des Kommunikators `comm`. Wir haben alle `MPI_Isend(...)`-Aufrufe in der gesamten MPI-Schnittstelle durch `MPI_Issend(...)`-Aufrufe ersetzt um zu garantieren, dass keine weiteren Nachrichten über den Kommunikator `comm` zu empfangen sind, sobald der stille Zustand erreicht ist.

Nachfolgend beschreiben wir die Funktionsweise des Protokolls. Wie bereits erklärt, wird die Protokollausführung mit einer Nachricht `hello` von Prozess `0` des Clients an Prozess `0` des Servers initiiert. Die `hello`-Nachricht wird durch den Horch-Thread empfangen und die Verbindungsinformationen zum Client-Prozess `0` in eine Variable `global_state.joining` von Server-Prozess `0` geschrieben.

Die Methode `x10rt_net_probe()` von Server-Prozess `0` überprüft zusätzlich zu ihren in Abschnitt 7.2 beschriebenen Aufgaben, ob der Wert von `global_state.joining` ungleich `0` ist, also ob eine `hello`-Nachricht empfangen wurde. Ist dies der Fall, so sendet Server-Prozess `0` eine Nachricht `stop_send` über `MPI_Issend(...)` an alle Server-Prozesse.

Bei Erhalt dieser Nachricht sperrt ein Server-Prozess die Liste `global_state.free_reqs`, sodass keine neuen `MPI_Request`s für Sendeoperationen entnommen werden dürfen. Der Abschluss von `x10rt_net_send_get(...)` wird hierdurch nicht beeinträchtigt, da die verwendeten `MPI_Request`s, wie in Abschnitt 7.2 beschrieben, wiederverwendet werden. Dieser Server-Prozess ruft dann in einer Schleife die Methode `x10rt_net_probe()` auf, bis die Liste `global_state.pending_sends` leer ist. Dann sendet er eine Nachricht `rdy_join` an Server-Prozess `0`. Die Liste `pending_recvs` kann zu diesem Zeitpunkt nicht leer sein. Wenn ein MPI-Prozess `p` auf eine Nachricht eines anderen Prozesses `q` wartet, so muss auf Seiten von `q` eine noch nicht abgeschlossene Sendeoperation zu der Empfangsoperation von `p` vorliegen. Sind alle Sendeoperationen auf allen Prozessen abgeschlossen, so ist durch die Verwendung von `MPI_Issend(...)` sichergestellt, dass keine weiteren Nachrichten über `comm` gesendet werden.

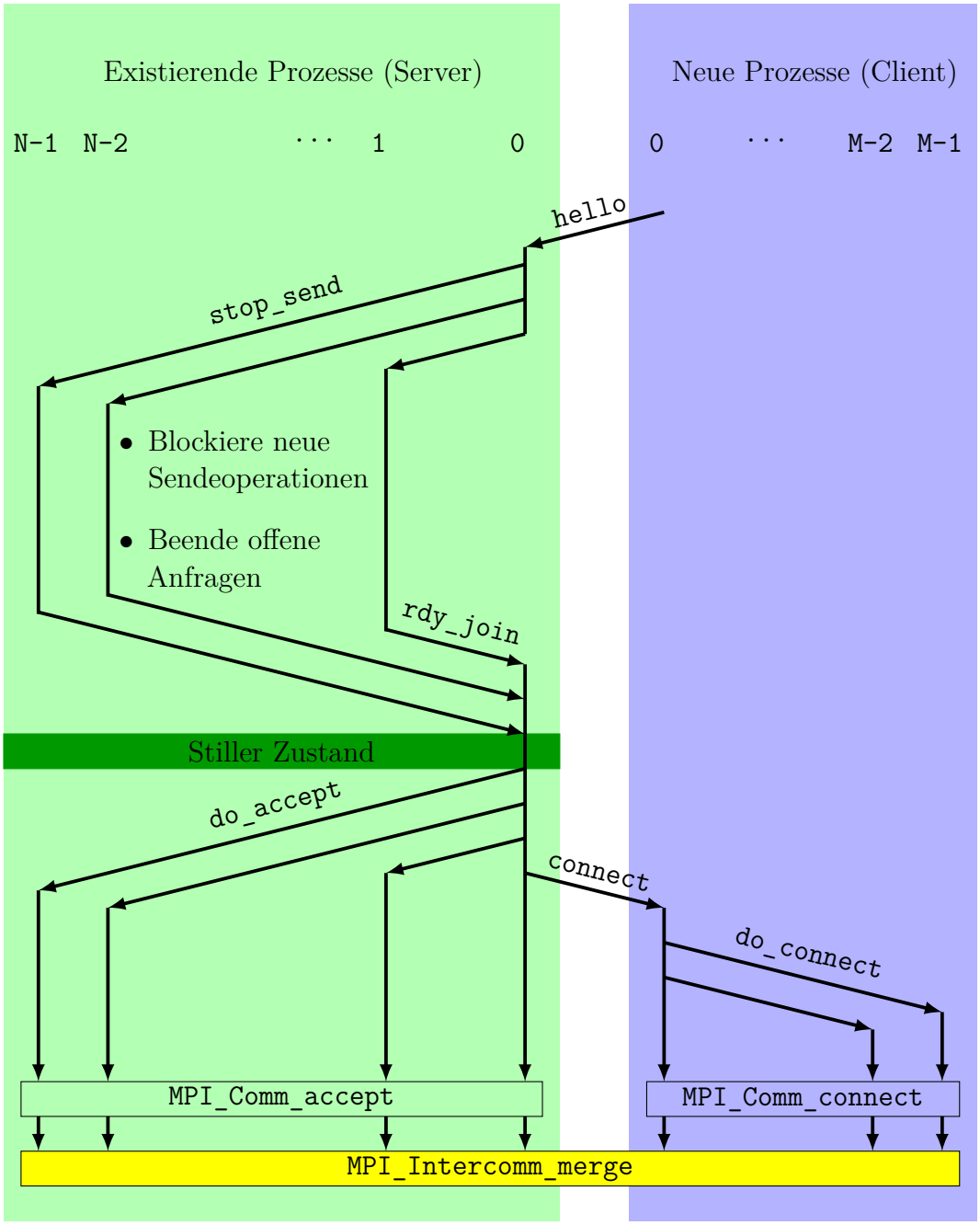


Abbildung 7.1.: Protokoll zum Hinzufügen von Prozessen (siehe [44])

doi:10.17170/kobra-2018122577

Hat Server-Prozess `0` von allen Server-Prozessen eine Nachricht `rdy_join` erhalten, so ist der stille Zustand erreicht und der Kommunikator kann ausgetauscht werden. Prozess `0` öffnet einen Port mittels `MPI_Open_port(...)` und sendet an alle Server-Prozesse ein Nachricht `do_accept`. Bei Erhalt dieser Nachricht startet jeder Prozess einen separaten Thread, in dem er `MPI_Comm_accept(...)` aufruft. Wir bezeichnen diesen Thread als *Join-Thread* und den Thread, aus dem der Join-Thread erstellt wurde als *Parent-Thread*. Wir gehen am Ende dieses Kapitels darauf ein, warum wir pro Prozess einen Join-Thread starten. An dieser Stelle ist das Blockieren von `MPI_Comm_accept(...)` unkritisch, da zuvor der stille Zustand erreicht wurde.

Nach Versenden der `do_accept`-Nachrichten sendet Server-Prozess `0` die Portnummer des MPI-Ports mittels einer Nachricht `connect` an Prozess `0` der Client-Seite. Hierfür wird der Netzwerkport genutzt, der bereits für die `hello`-Nachricht verwendet wurde. Diese Nachricht enthält zusäztlich die `rank`s aller bereits ausgefallenen Prozesse und wird auf der Client-Seite dafür verwendet, aus dem `rank` eines Client-Prozesses die korrekte Place-ID zu berechnen.

Bei Ausführung der Nachricht `connect` sendet Client-Prozess `0` an alle Client-Prozesse eine Nachricht `do_connect` welche eine Liste `dead_places` aller IDs aosgefallener Place enthält. Die Portnummer zu Server-Prozess `0` wird nur auf Client-Prozess `0` benötigt und wird daher nicht mitgesendet. Diese Nachricht wird analog zur Nachricht `do_accept` auf der Server-Seite behandelt, nur dass die Client-Prozesse `dead_places` abspeichern und einen Thread starten, der `MPI_Comm_connect(...)` anstatt `MPI_Comm_accept(...)` ausführt.

Nach erfolgreicher Ausführung von `MPI_Comm_connect(...)` und `MPI_Comm_accept(...)` führen alle Prozesse auf Client- und Server-Seite sofort `MPI_Intercomm_merge(...)` aus, um einen neuen Kommunikator `newComm` zu erstellen. Dies geschieht noch in den Join-Threads der jeweiligen Prozesse. Nach erfolgreicher Ausführung dieser Methode setzt jeder Thread `comm` auf `newComm` und endet. Das Hinzufügen der neuen Prozesse ist hiermit abgeschlossen und es kann wieder kommuniziert werden. Dazu wird die Liste `free_req` auf jedem Server-Prozess entsperrt.

Es ist möglich, dass bei der Ausführung von `MPI_Comm_accept(...)`, `MPI_Comm_connect(...)` oder `MPI_Intercomm_merge(...)` ein Fehler auftritt, der dazu führt, dass der entsprechende Aufruf nicht beendet wird. Beispielsweise könnte auf Client-Seite Prozess `0` abstürzen, bevor er die Nachricht `connect` erhält, jedoch nachdem alle Server-Prozesse bereits

`MPI_Comm_accept(...)` aufgerufen haben. Unsere Erweiterung vermeidet dies, indem die Join-Threads nach einem Timeout durch die Parent-Threads abgebrochen werden. Die Überwachung ist so realisiert, dass der Parent-Thread, der den Join-Thread erzeugt, die Prozess-ID des erzeugten Join-Threads abspeichert und mittels busy waiting wartet, bis die Liste `free_req` entsperrt wird. Geschieht dies nicht innerhalb des definierten Timeouts, so wird der Join-Thread über seine Prozess-ID vom Parent-Thread beendet. Auf der Server-Seite wird daraufhin die Ausführung des Protokolls abgebrochen und das Server-Programm wird nur mit den Server-Prozessen fortgesetzt. Auf der Client-Seite wird das Client-Programm sofort abgebrochen.

doi:10.17170/kobra-2018122577

8. Experimentelle Auswertung

Nachfolgend bezeichnen wir das GLB-Framework aus der X10-Distribution als *GLB*. Den fehlertoleranten und elastischen Algorithmus bezeichnen wir als *FEGLB*. Die in Abschnitt 6.1 vorgestellte Variante des vorausschauenden Stehlens bezeichnen wir als $FEGLB_{FS}$ (**F**ore**S**ight, deutsch Voraussicht). Die Variante der Stehl-Weiterleitung aus Abschnitt 6.2 bezeichnen wir als $FEGLB_{FW}$ (**F**or**W**ard, deutsch Weiterleitung). Die in Abschnitt 6.3 vorgestellte Variante zum Senden inkrementeller Backups bezeichnen wir als $FEGLB_{Inc}$ (**Inc**remental, deutsch inkrementell). Sie verwendet weder vorausschauendes Stehlen noch Stehlweiterleitung.

Wir haben zu unterschiedlichen Zeitpunkten Places ausfallen lassen, um die Korrektheit unseres Frameworks zu überprüfen. Diese Tests werden nachfolgend nicht besprochen. Stattdessen konzentrieren wir uns auf Laufzeitmessungen. Der Fokus liegt auf dem zeitlichen Mehraufwand von FEGLB bzw. seiner Varianten, gegenüber GLB. Eine Auswertung in dieser Form erfolgt erstmalig in dieser Arbeit. Insbesondere wurden alle Messungen mit der gleichen Hard- und Softwareumgebung durchgeführt, welche sich von der in früheren Publikationen verwendeten unterscheidet.

Abschnitt 8.1 beschreibt diese Hard- und Softwareumgebung während die verwendeten Benchmarks in Abschnitt 8.2 besprochen werden. Inhalt von Abschnitt 8.3 ist die Konfiguration der Benchmarks sowie der Parameter von GLB, FEGLB und seinen Varianten. Die Präsentation und Diskussion der Experimente erfolgt in Abschnitt 8.4.

8.1. Hard- und Softwareumgebung

Die Experimente wurden auf der FB16-Partition des Clusters der Universität Kassel durchgeführt [110]. Die Partition besteht aus 12 homogenen, über Infiniband verbundenen Rechenknoten. In jedem Rechenknoten sind zwei Intel® Xeon® E5-2643 v4 CPUs mit jeweils 6 Rechenkernen sowie 256

GB Arbeitsspeicher verbaut. Die CPUs sind mit einer Frequenz von 3.40 GHz getaktet. Jeder Rechenkern besitzt einen 192 KB großen L1-Cache und einen $1,5$ MB großen L2-Cache [111]. Alle Kerne einer CPU teilen sich einen 20 MB großen L3-Cache [112]. Hyperthreading war während der Messungen deaktiviert.

Auf den Rechenknoten ist das Betriebssystem CentOS in Version 7.2 installiert. Das Cluster verwendet als Batchsystem Slurm in Version 17.11.7, sowie GPFS in Version 4.2.3-10 als paralleles Dateisystem.

Zur Übersetzung der Programme wurden X10 in Version 2.6.1 und der GNU C Compiler in Version 7.1.0 verwendet. Für GLB wurde die MPI-Schnittstelle von X10 mit Open MPI in Version 2.1.1 kompiliert und ausgeführt. Der Fehlertoleranzmodus von X10 wurde für GLB deaktiviert. Für FEGLB und seine Varianten wurde die MPI-Schnittstelle, inklusive der in Abschnitt 7.3 vorgestellten Elastizität, mit ULFM Version 2.0 kompiliert und ausgeführt. Außerdem wurde die Fehlertoleranz von X10 aktiviert.

8.2. Verwendete Benchmarks

Für die Laufzeitmessungen haben wir Experimente mit drei verschiedenen Benchmarks durchgeführt, welche wir nachfolgend vorstellen.

Unbalanced Tree Search [37] (kurz *UTS*) generiert einen unbalancierten Baum. Die Expansion der Baumknoten findet erst zur Laufzeit statt. Die Tiefe `d` des Baumes, die maximale Anzahl `b` an Kindern pro Knoten, die Wahrscheinlichkeitsverteilung zur Generierung der Kinder eines Knotens sowie ein initialer Seed `r` sind Parameter des Benchmarks. Durch den Seed ist gewährleistet, dass der Benchmark bei gleicher Konfiguration immer den gleichen Baum erzeugt. Die Form des Baumes ist ebenfalls durch einen Parameter beeinflussbar, es kann zur Generierung des Baumes zwischen einer Binomial- und einer geometrische Verteilung gewählt werden. Jeder Knoten wird durch einen Hash-Wert dargestellt. Der Hash-Wert des Wurzelknotens wird über den Seed bestimmt. Der Hash-Wert aller anderen (Kind-)Knoten berechnet sich aus dem Hash-Wert des Elternknotens und der Nummer des jeweiligen Kindes, wobei die Kinder von 0 bis $k-1$ durchnummeriert sind. Hierbei hängt k vom Hash-Wert des Elternknotens und der gewählten Wahrscheinlichkeitsverteilung ab. Die Knoten des Baumes werden zur Laufzeit expandiert, indem aus dem Hash-Wert des Elternknotens der Hash-Wert eines Kindknotens berechnet wird. Das Ergebnis der Berechnung ist die Anzahl der Knoten des Baumes,

 doi:10.17170/kobra-2018122577

also ein einziger ganzzahliger Wert.

Eine Implementierung dieses Benchmarks für GLB ist in der X10-Distribution enthalten. Die Kinder werden in dieser Implementierung mit Hilfe einer geometrischen Verteilung erzeugt. Jeder Knoten des Baumes stellt einen abzuarbeitenden Task dar.

Zu Beginn ist nur ein einzelner Task, der den Wurzelknoten repräsentiert, vorhanden. Dieser wird von Place `0` ausgeführt. Es findet also ein dynamischer Start mit anschließender Arbeitsverteilung über den Lifeline-Graphen statt. Die Ausführung von Tasks erzeugt ggf. neue Tasks.

Die GLB-Implementierung von UTS verwendet eine kompakte Darstellung für die `Queue`. Statt jeden Knoten explizit als in sich abgeschlossenen Task abzuspeichern, werden drei `Rail`s (`hash`, `lower` und `upper`) zum Abspeichern der Knoten verwendet. Jeder Eintrag `i` beschreibt den Hash-Wert eines Elternknotens (`hash(i)`) sowie die Nummern von Kindern ([`lower(i)`, `upper(i)`)), die zu berechnen sind. Somit repräsentieren drei zusammengehörige Werte aus `hash`, `lower` und `upper` exakt (`upper` − `lower`) Tasks. Diese Variante von UTS bezeichnen wir mit UTS_C (**C**ompact, deutsch kompakt).

Leider lässt die kompakte Implementierung der `Queue` keine Dequeue-Implementierung zu und ist somit nicht mit dem in Abschnitt 6.3 vorgestellten inkrementellen Backupalgorithmus kompatibel. Für die experimentelle Bewertung des inkrementellen Backupalgorithmus mit UTS haben wir die Implementierung so umgeschrieben, dass die `Queue` durch ein Dequeue von `TreeNode`s implementiert ist. Eine Instanz der Klasse `TreeNode` beinhaltet die Nummer des Kindes, welches sie repräsentiert, sowie den Hash-Wert des zugehörigen Elternknotens ab. Jeder Knoten des Baumes, und somit jeder Task, wird also als eigenständiger Eintrag in dieser Dequeue dargestellt. Die abgeänderte Variante bezeichnen wir als UTS_{Ex} (**Ex**plizit).

Betweenness Centrality [38] (kurz BC) berechnet für jeden Knoten v in einem gerichteten, gewichteten und azyklischen Graphen den Wert

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$$

Dabei bezeichnet

- $\sigma_{s,t}$ die Anzahl der kürzesten Wege von Knoten s zu Knoten t und

- $\sigma_{s,t}(v)$ die Anzahl der kürzesten Wegen von Knoten s zu Knoten t, die durch Knoten v laufen.

Der Graph liegt, anders als der Baum bei UTS, zu Beginn der Berechnung vollständig vor. Die Form des Graphen kann über Übergabeparameter gesteuert werden. Die Größe des Graphen $\texttt{N} = 2^{\texttt{n}}$ wird über den Übergabeparameter `n` vorgegeben. Die Knoten sind von `0` bis `N-1` durchnummeriert. Wie bei UTS wird der Graph auf Grundlage eines Seeds `s` generiert, sodass bei gleicher Konfiguration des Benchmarks der gleiche Graph erzeugt wird. Das Ergebnis der Berechnung ist ein `Rail`, welches zu jedem Knoten v den Betweenness Centrality Wert $g(v)$ angibt. Dies stellt einen grundlegenden Unterschied zum UTS-Benchmark dar, bei dem das Ergebnis ein einziger Wert ist.

Ein Task der GLB-Implementierung des BC-Benchmarks führt hintereinander die folgenden zwei Schritte aus:

1. Für einen gegebenen Knoten s werden alle kürzesten Wege von s zu allen anderen Knoten berechnet.

2. Für alle Knoten v, die auf einem kürzesten Weg von s zu einem anderen Knoten t liegen, wird der Wert $g(v)$ aktualisiert.

Die Ausführungszeit eines Tasks hängt vom Graphen ab und kann groß sein. Daher unterbricht die GLB-Implementierung des Benchmarks ggf. die Abarbeitung eines Tasks, um Nachrichten zu empfangen und Stehlanfragen zu beantworten.

Während der Programmausführung werden keine neuen Tasks erzeugt. Die Tasks werden zu Beginn der Berechnung auf die Places verteilt, es findet also ein statischer Start statt. Eine Lastenbalancierung ist notwendig, da die Tasks unterschiedliche Laufzeit haben können. Ähnlich zur UTS-Implementierung verwendet auch die BC-Implementierung eine kompakte Darstellung der `Queue`. Hier definieren korrespondierende Einträge in den `Rails` `lower` und `upper` Intervalle von Knoten, die als Startknoten s abzuarbeiten sind. Durch Stehlanfragen kann es dazu kommen, dass jedes `Rail` mehr als einen Eintrag speichert. Wir nennen diese Variante nachfolgend BC_C.

Auch für diesen Benchmark haben wir eine abgeänderte Variante implementiert, die die `Queue` durch eine Dequeue realisiert. Sie unterbricht die Ausführung eines Tasks ebenso wie BC_C, um Nachrichten zu empfangen und

 doi:10.17170/kobra-2018122577

aufgezeichnete Nachrichten abzuarbeiten. Wir nennen diese Variante nachfolgend BC_{Ex}.

Das **`N`-Damen Problem** [113] (kurz *NQueens*) berechnet die Anzahl der Möglichkeiten, `N` Damen auf einem `N` × `N`-großen Schachbrett zu positionieren, ohne dass sich die Damen gegenseitig schlagen können. Ähnlich zu UTS ist das Ergebnis der Berechnung ein einzelner, ganzzahliger Wert. Der Wert `N` kann per Übergabeparameter gesetzt werden.

Wir haben den in [114] zur Verfügung gestellten Algorithmus mittels X10 für GLB implementiert. Ein Task stellt hierbei ein Schachbrett mit bereits $k < N$ gültig positionierten Damen dar. Die Berechnung wird mit einem einzigen Task gestartet, der ein leeres Schachbrett repräsentiert. Es findet, wie bei UTS, ein dynamischer Start mit Arbeitsverteilung über den Lifeline-Graphen statt.

Ein Task wird abgearbeitet, indem für jedes freie Feld geprüft wird, ob auf diesem eine neue Dame platziert werden kann. Kann auf einem Feld eine Dame platziert werden, so wird ein neuer Task erstellt, der das um eine Dame erweiterte Schachbrett repräsentiert. Sind nur noch `t` (**t**hreshold, deutsch Schwellwert) Damen zu positionieren, so wird der Task sequentiell abgearbeitet, es findet keine Erzeugung neuer Tasks statt. Hierbei ist `t` ein Parameter des Benchmarks.

Die Implementierung der `Queue` erfolgt über eine Dequeue, sodass keine Anpassung für den inkrementellen Backupalgorithmus notwendig ist.

8.3. Konfiguration

Wir haben zur experimentellen Auswertung starke Skalierung verwendet. Das heißt, dass wir jeden Benchmark in gleicher Konfiguration mit verschiedener Anzahl an Places gestartet haben. Die Konfigurationen der einzelnen Benchmarks sind in Tabelle 8.1 aufgeführt. Die Parameter für FEGLB_{FS} und FEGLB_{FW} sind identisch zu den Parametern von FEGLB, bei FEGLB_{FS} kommt lediglich der Parameter `f` für das vorausschauende Stehlen hinzu.

Für die UTS- und BC-Benchmarks liefert $\texttt{n} \in [255, 8191]$ gleich gute Laufzeiten, daher haben wir den Parameter auf dem voreingestellten Standardwert von 511 belassen. Größere Werte führen zu einer schlechteren Taskverteilung und in Konsequenz zu einer längeren Laufzeit. Wird der Parameter `n` kleiner gewählt, so wird die Taskausführung häufiger unterbrochen, um Nachrichten

Benchmark	Benchmark Parameter(s)	Framework	Framework Parameter(s)
UTS	$\texttt{d} = 17$, $\texttt{b} = 4$, $\texttt{r} = 19$	GLB	$\texttt{n} = 511$
		FEGLB	$\texttt{n} = 511$, $\texttt{k} = 8\,192$, $\texttt{g} = 512$
		FEGLB_{FS}	siehe FEGLB, $\texttt{f} = 15$
BC	$\texttt{N} = 2^{17}$, $\texttt{a} = 0.55$, $\texttt{b} = 0.1$, $\texttt{c} = 0.1$, $\texttt{d} = 0.25$, $\texttt{s} = 2$	GLB	$\texttt{n} = 511$
		FEGLB	$\texttt{n} = 511$, $\texttt{k} = 8\,192$, $\texttt{g} = 512$
		FEGLB_{FS}	siehe FEGLB, $\texttt{f} = 50$
NQueens	$\texttt{s} = 17$, $\texttt{t} = 6$	GLB	$\texttt{n} = 1$
		FEGLB	$\texttt{n} = 1$, $\texttt{k} = 128$, $g = 512$
		FEGLB_{FS}	siehe FEGLB, $\texttt{f} = 5$

Tabelle 8.1.: Parameter für die Benchmarks

zu empfangen. Die Konsequenz daraus ist ebenfalls eine höhere Laufzeit. Der NQueens-Benchmark erzielt bei $\texttt{n}= 1$ die kürzeste Laufzeit.

Den Parameter `k` für das Intervall zwischen zwei regulären Backups haben wir so gewählt, dass ungefähr ein Backup pro Place und Sekunde geschrieben wird. Dieser Wert wurde experimentell bestimmt und hat sich als guter Kompromiss erwiesen: wird `k` höher gesetzt, so dominieren Stehl-Backups; bei geringeremm `k` steigt die Laufzeit aufgrund zu häufiger regulärer Backups.

Der GLB-Parameter `w`, der die Anzahl der zufälligen Stehlversuche bestimmt, wurde bei Testläufen auf 6 oder weniger Places auf die Anzahl der Places $\texttt{N} - 1$ gesetzt. Bei mehr als 6 Places wurde er auf $\lfloor \texttt{N}/10 \rfloor$, aber mindestens auf 6 gesetzt. Den Parameter `l` zur Berechnung des Lifeline-Graphen haben wir auf $\lceil \sqrt{\text{Anzahl Places}} \rceil$ gesetzt. In experimentellen Messungen haben sich diese Parameterwerte als gute Standardwerte erwiesen.

8.4. Experimente und Diskussion

Alle nachfolgenden Experimente wurden 5-fach wiederholt. Die Tabellen mit den gemessenen Laufzeiten befinden sich in Anhang A. Angegeben sind die Mittelwerte der 5 gemessenen Laufzeiten.

Wir verwenden nachfolgend die Begriffe *Overhead*, *Beschleunigung* und *Speedup*. Den Overhead der Laufzeit $t(\texttt{P})$ eines Programms `P` im Vergleich zur Laufzeit $t(\texttt{Q})$ eines Programms `Q` bei gleicher Anzahl an Places definieren wir als $(t(\texttt{P})/t(\texttt{Q})) - 1$ (in Prozent). Ist der Overhead kleiner 0, reden wir von einer

 doi:10.17170/kobra-2018122577

Beschleunigung. Den Speedup eines Programms P mit N Places berechnen wir durch $t_1(\mathtt{P})/t_{\mathtt{N}}(\mathtt{P})$, wobei $t_1(\mathtt{P})$ die Laufzeit von P auf einem Place und $t_{\mathtt{N}}(\mathtt{P})$ die Laufzeit bei Ausführung von P auf N Places meint.

Nachfolgend stellen wir den Overhead unseres Algorithmus und seiner Varianten dar. Die Darstellung erfolgt in Form eines Graphen über die Anzahl der Places. Gegenüber welcher Vergleichsbasis der Overhead dargestellt wird, geben wir in den jeweiligen Abschnitten an.

Für die genauere Analyse nutzen wir eine bereits in GLB implementierte Logging-Komponente, die

- die Anzahl der gestohlenen Tasks
- die Anzahl der erfolgreich beantworteten zufälligen Stehlanfragen,
- die Anzahl der erfolgreiche beantworteten Lifeline-Anfragen und
- pro Place die Anzahl der abgearbeiteten Tasks

mitzählt. Für die fehlertoleranten Frameworks haben wir diese Logging-Komponente so erweitert, dass zusätzlich die Anzahl der Stehl-Backups und der regulären Backups mitgezählt wird.

UTS_C und BC_C wurden mit GLB, FEGLB, FEGLB_{FS} und FEGLB_{FW} gestartet. UTS_{Ex} und BC_{Ex} wurden mit GLB, FEGLB und FEGLB_{Inc} gestartet.

In Abschnitt 8.4.1 werden die gemessenen Intra- und Inter-Knoten Laufzeiten mit bis zu 12 Places vorgestellt und besprochen. Mit diesen Messungen wurden die Performanceeinbußen beim Starten mehrerer Places pro Knoten ermittelt. Es zeigte sich, dass es Abweichungen bag, diese aber bei allen Frameworks ähnlich waren.

In den Abschnitten 8.4.2 bis 8.4.4 zeigen und diskutieren wir die Ergebnisse für UTS, BC und NQueens. Bei einer Placeanzahl ≤ 12 wurden alle Places auf einem Knoten gestartet. Mit mehr als 12 Places wurden 12 Places pro Knoten gestartet. Die Places wurden zyklisch auf die Knoten verteilt; in einem Programmlauf mit K Knoten wurde Place P auf Knoten P mod K gestartet. Wir haben eine zyklische Verteilung gewählt, damit die Wahrscheinlichkeit von endgültigem Datenverlust bei Ausfall eines Rechenknotens vermindert wird. Die in Abschnitt 8.4.1 erhobenen Laufzeiten mit 1 bis 12 Places auf einem Knoten haben wir für die Analyse wiederverwendet und zusätzlich die Laufzeiten mit 24 bis 144 Places gemessen. Bei diesen Experimenten haben wir keine Places ausfallen lassen und keine Places hinzugefügt.

Messungen zu Fehlertoleranz und Elastizität sind Inhalt von Abschnitt 8.4.5. Wir messen, wie sich Ausfälle von Places sowie das Hinzufügen von Places auf die Laufzeiten auswirkt. Wir beschreiben die Testaufbauten, die uns erlauben, den Verlust bzw. Gewinn an Rechenleistung, der durch Ausfälle bzw. Hinzufügen von Places gegeben ist, zu bestimmen. Bei der Analyse vergleichen wir hierzu die Laufzeiten dieser Experimente mit geeignet gewählten Experimenten ohne Ausfälle bzw. ohne Hinzufügen von Places.

Eine abschließende Zusammenfassung der Messergebnisse nehmen wir in Abschnitt 8.4.6 vor.

8.4.1. Intra- und Inter-Knoten Laufzeiten

Für eine volle Auslastung eines Rechenknotens müssen 12 Places pro Knoten gestartet werden, da jeder Rechenknoten 12 Rechenkerne besitzt (siehe Abschnitt 8.1). Um zu gewährleisten, dass keinem Framework ein unfairer Vor- oder Nachteil durch die Ausführung von bis zu 12 Places pro Knoten entsteht, haben wir die Intra- und Inter-Knoten Laufzeiten gemessen. Hierzu haben wir zwei Gruppen von Experimenten durchgeführt. In beiden Gruppen wurde jeder Benchmark mit 1 bis 12 Places gestartet. Für die erste Gruppe wurden alle Places eines Programmlaufs auf einem Rechenknoten gestartet. Wir kürzen diese Gruppe von Experimenten mit *SN* (**S**ingle **N**ode, deutsch einzelner Knoten) ab. In der zweiten Gruppe wurde jeder Place auf einem separaten Knoten gestartet, es wurden also so viele Knoten wie Places genutzt. Wir kürzen diese Gruppe mit *MN* (**M**ultiple **N**odes, deutsch mehrere Knoten) ab.

Die Laufzeiten der MN-Experimente waren in der Regel niedriger als die Laufzeiten der SN-Experimente. Wir begründen dies damit, dass X10 die knoteninterne Kommunikation nicht optimiert. Somit kann bei den MN-Experimenten der einzelne Place pro Knoten die ihm zur Verfügung stehende Hardware alleine nutzen. In den SN-Experimenten konkurrieren die Places um Hardwareressourcen, wodurch ein Overhead zustande kommt.

In den Abbildungen 8.1 bis 8.5 ist der Overhead der jeweiligen SN- über die entsprechenden MN-Laufzeiten in Prozent aufgetragen. Die Laufzeiten der SN- und MN-Experimente sind in den Tabellen A.1 bis A.10 aufgeführt (der Übersichtlichkeit halber sind SN- und MN-Läufe in separaten Tabellen aufgelistet).

Die Graphen für UTS_C, UTS_{Ex} (Abbildungen 8.1 und 8.2), und NQueens

 doi:10.17170/kobra-2018122577

(Abbildung 8.5) zeigen einen Overhead von -2% bis 10%.

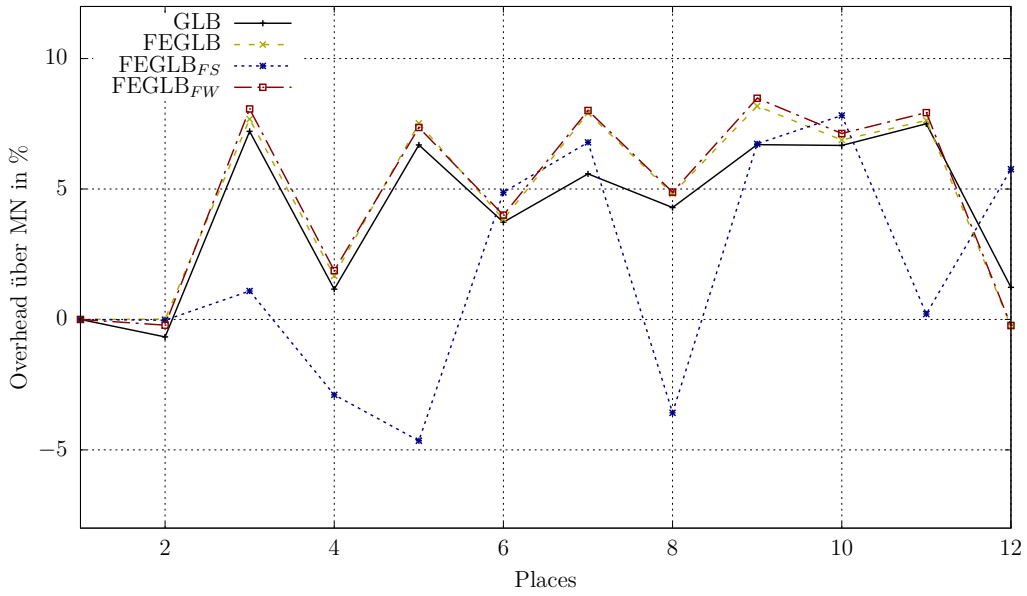


Abbildung 8.1.: UTS$_C$: Overhead der SN- über die entsprechenden MN-Laufzeiten

Bei BC$_C$ und BC$_{Ex}$ (Abbildungen 8.3 und 8.4) haben alle Frameworks einen stetig wachsenden Overhead von bis zu 160%. Dieser Overhead setzt jeweils ab 3 Places ein. Wir haben während der Läufe sichergestellt, dass weder zu wenig Arbeitsspeicher noch eine Überlastung der CPU durch zu viele Prozesse für dieses Verhalten verantwortlich ist. Es sei daran erinnert, dass das Ergebnis einer BC-Berechnung ein `Rail` der Länge $N = 2^{17}$ ist (siehe Abschnitt 8.2 und Abschnitt 8.3). Jeder Place hält ein `Rail` dieser Größe als lokales Ergebnis vor und modifiziert die Einträge während des Programmlaufs. Daraus ergibt sich eine hohe Anzahl von Speicherzugriffen, die für den hohen Overhead verantwortlich sein könnte.

Trotz der genannten Unterschiede lässt sich beobachten, dass alle untersuchten Frameworks mit gleichem Benchmark ein ähnliches Verhalten zeigen. Somit können für die nachfolgenden Messungen pro Rechenknoten 12 Places gestartet werden, ohne dass einem der Frameworks daraus Vor- oder Nachteile entstehen.

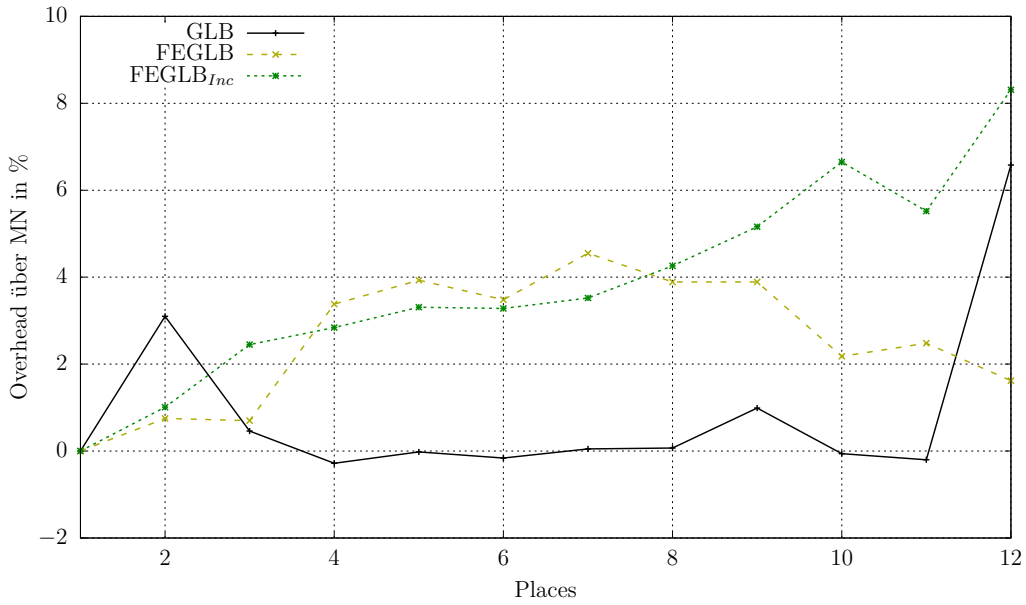


Abbildung 8.2.: UTS$_{Ex}$: Overhead der SN- über die entsprechenden MN-Laufzeiten

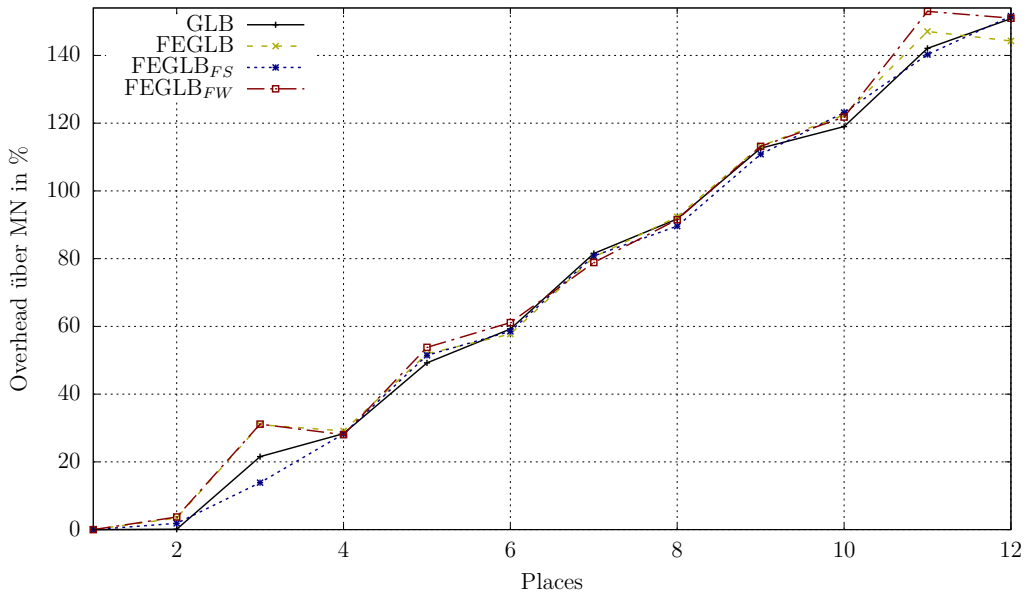


Abbildung 8.3.: BC$_C$: Overhead der SN- über die entsprechenden MN-Laufzeiten

doi:10.17170/kobra-2018122577

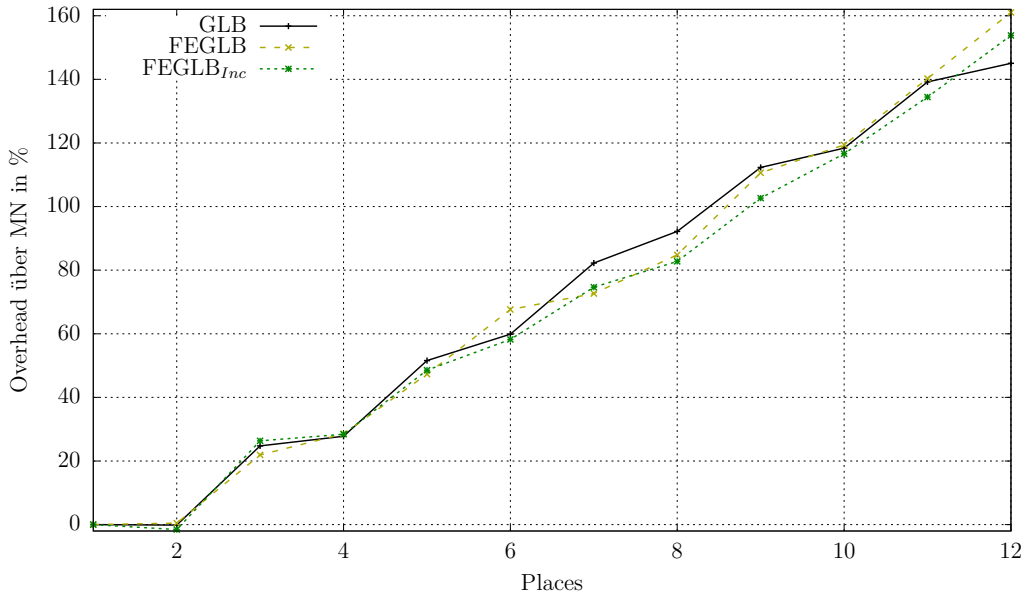


Abbildung 8.4.: BC$_{Ex}$: Overhead der SN- über die entsprechenden MN-Laufzeiten

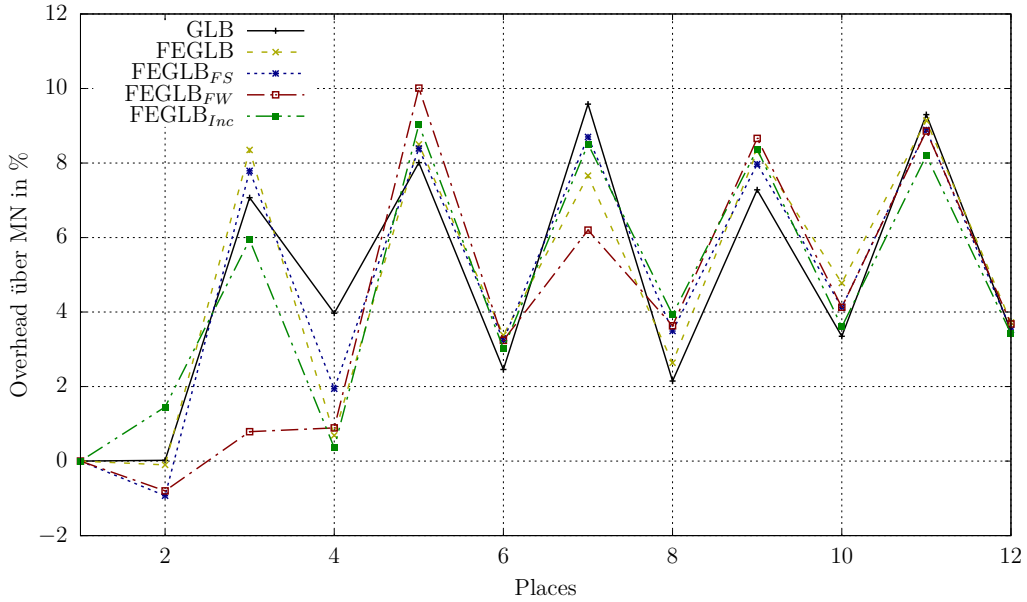


Abbildung 8.5.: NQueens: Overhead der SN- über die entsprechenden MN-Laufzeiten

doi:10.17170/kobra-2018122577

8.4.2. UTS$_C$- und UTS$_{Ex}$-Benchmark

UTS$_C$ Abbildung 8.6 zeigt den Overhead der FEGLB-Varianten über das GLB-Framework in Abhängigkeit der Placeanzahl. In den Tabellen A.1 und A.11 finden sich die entsprechenden Laufzeiten mit 1 bis 12 respektive 24 bis 144 Places. Weiterhin sind in Tabelle 8.2 die Speedups der Frameworks mit 144 Places aufgeführt.

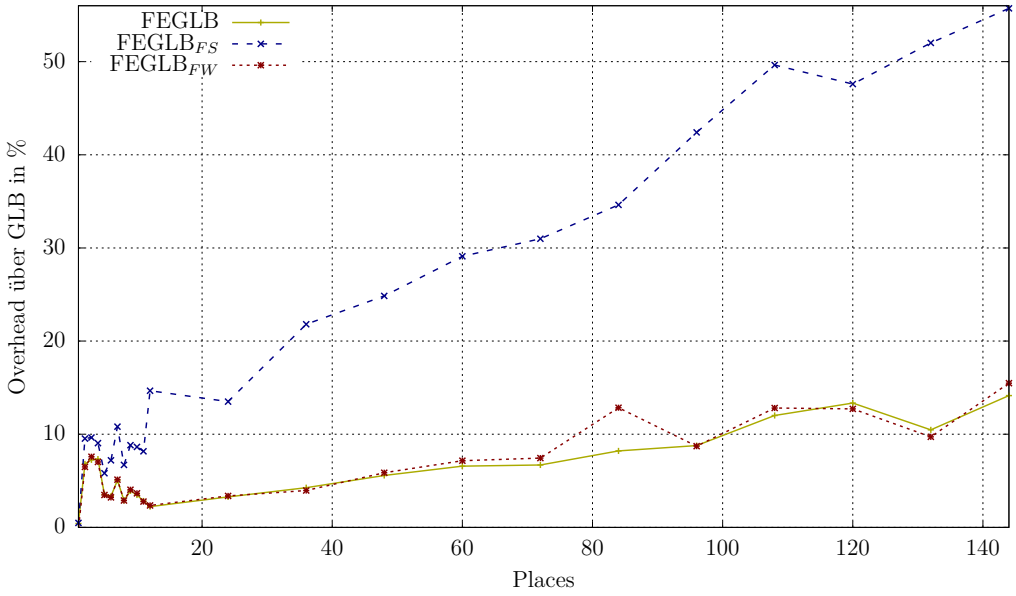


Abbildung 8.6.: UTS$_C$: Overhead der FEGLB-Varianten gegenüber GLB

Framework	GLB	FEGLB	FEGLB$_{FS}$	FEGLB$_{FW}$
Speedup	133,79	117,69	86,33	115,65

Tabelle 8.2.: UTS$_C$: Speedup mit 144 Places.

FEGLB hat gegenüber GLB einen Overhead zwischen $0,42\%$ (auf einem Place) und $14,15\%$ (auf 144 Places). Der Overhead wächst proportional zur Anzahl gesendeter Backups: FEGLB sendet bei Läufen mit 12 Places im Schnitt circa 450 Stehl-Backups und 6 300 reguläre Backups, mit 144 Places sind es circa 9 600 Stehl-Backups und 4 100 reguläre Backups. Diese Entwicklung der Stehl-Backups war zu erwarten, da die Unbalanciertheit des Baumes zu

doi:10.17170/kobra-2018122577

häufigen Stehlanfragen – und somit Stehlbackups – führt. Die Anzahl der Anfragen wächst mit der Anzahl der Places, die mit Arbeit zu versorgen sind. FEGLB erreicht auf 144 Places einen Speedup von $117,69$ und weicht damit um $12,03\%$ von GLB-Speedup ab.

FEGLB$_{FS}$ hat einen deutlich höheren Overhead als FEGLB und FEGLB$_{FW}$. Ursache ist ein rapider Ansteig der gesendeten Stehl-Backups. Ab 12 Places ist die Anzahl der gesendeten Stehl-Backups rapide angestiegen. Mit 12 Places sendet FEGLB$_{FS}$ im Schnitt circa 2 300 Stehl-Backups und 5 900 reguläre Backups. Mit 144 Places sendet FEGLB$_{FS}$ circa 24 500 Stehl-Backups und 2 100 reguläre Backups.

Das frühzeitige Stehlen führt im Fall von UTS dazu, dass Worker ständig stehlen: Steigt ihr Taskpool durch eine von einem anderen Place erfolgreich beantwortete Stehlanfrage nicht über den Schwellwert `f`, so beginnen sie sofort, erneut zu stehlen. Wird der Parameter `f` kleiner gesetzt, hat er geringere Auswirkungen und bei `f`$= 2$ ist die Laufzeit fast identisch zu den Laufzeiten von FEGLB und FEGLB$_{FW}$. Für UTS ist vorausschauende Stehlen somit als ungeeignet einzustufen.

Die Verläufe der Overheads von FEGLB und FEGLB$_{FW}$ sind nahezu identisch. Eine nähere Auswertung hat ergeben, dass

- die Anzahl an gestohlenen Tasks,
- die Anzahl an Stehltransaktionen, sowie
- die Anzahl an regulären und Stehl-Backups

sich ebenfalls gleichen. Einzig die Anzahl gestohlener Tasks ist bei FEGLB$_{FW}$ geringfügig größer, es werden mehr Tasks pro Stehltransaktion an mehr Worker weitergereicht. Das Weiterleiten von Stehlanfragen hat somit keinen negativen Effekt auf die Laufzeit, bietet im Fall von UTS jedoch auch keinen Vorteil.

Den höchsten Overhead gegenüber GLB haben FEGLB$_{FS}$ und FEGLB$_{FW}$ mit 144 Places: FEGLB$_{FS}$ mit $55,72\%$ und FEGLB$_{FW}$ mit $15,48\%$.

UTS$_{Ex}$ Der Overhead von FEGLB und FEGLB$_{Inc}$ gegenüber GLB ist in Abbildung 8.7 aufgetragen. Die zugehörigen Laufzeiten von 1 bis 12 bzw. 24 bis 144 Places sind in den Tabellen A.4 und A.12 angegeben. Weiterhin sind die Speedups in Tabelle 8.4 gezeigt.

Der Overhead von FEGLB gegenüber GLB ist bei UTS_{Ex} ähnlich zu dem Overhead mit UTS_C. Er liegt zwischen $1,72\%$ (auf einem Place) und $11,89\%$ (auf 120 Places). Der Speedup auf 144 Places ist mit $103,30$ geringer als der Speedup bei UTS_C, jedoch liegt der Speedup von GLB ebenfalls bei lediglich $110,40$. Somit weicht der Speedup von FELGB um $6,43\%$ von dem Speedup von GLB ab. Die Anzahl gesendeter Backups sind ähnlich zu denen aus den UTS_C-Läufen.

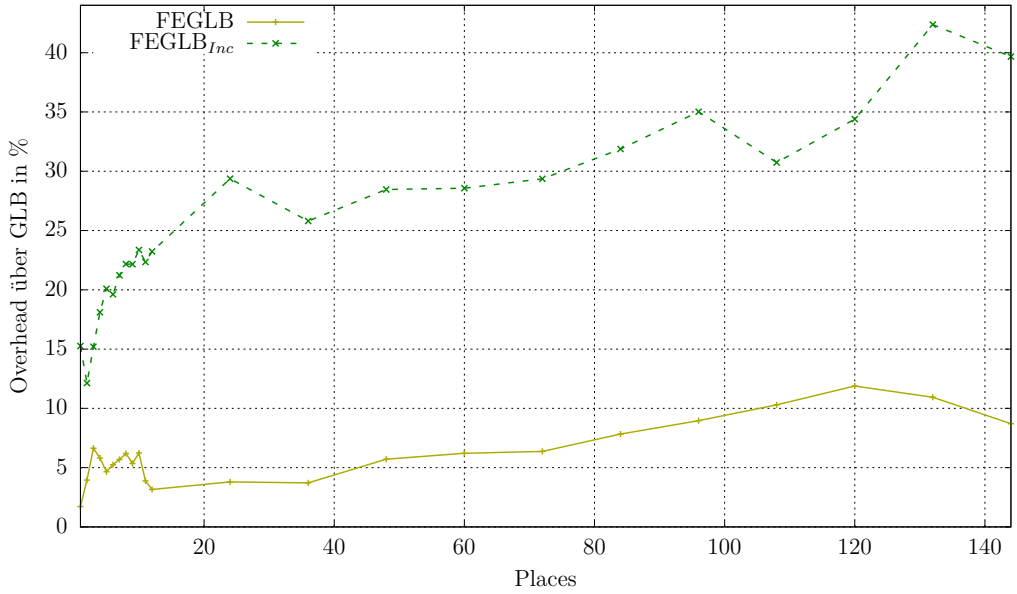


Abbildung 8.7.: UTS_{Ex}: Overhead der FEGLB-Varianten gegenüber GLB

Framework	GLB	FEGLB	FEGLB$_{Inc}$
Speedup	110,40	103,30	91,10

Tabelle 8.3.: UTS_{Ex}: Speedup mit 144 Places.

FEGLB$_{Inc}$ hat einen Overhead zwischen $12,13\%$ (mit 2 Places) und $44,37\%$ (mit 132 Places). Sowohl die Stehlraten als auch die Anzahl der gesendeten Backups ist ähnlich zu GLB und FEGLB. Wir erklären den Overhead damit, dass das inkrementelle Framework nach der Abarbeitung jedes Tasks überprüfen muss, ob ein neuer Minimalzustand erreicht wurde.

doi:10.17170/kobra-2018122577

8.4.3. BC_C- und BC_{Ex}-Benchmark

BC_C Abbildung 8.8 zeigt den Overhead der FEGLB-Varianten gegenüber GLB, während die Laufzeiten für BC in den Tabellen A.5 und A.13 angegeben sind. Tabelle 8.4 zeigt die Speedups der Frameworks. Da alle Frameworks einen schlechten Intra-Knoten-Speedup von 3 bis 12 Places haben (vergleiche Abbildung 8.8), sind alle Speedups entsprechend klein.

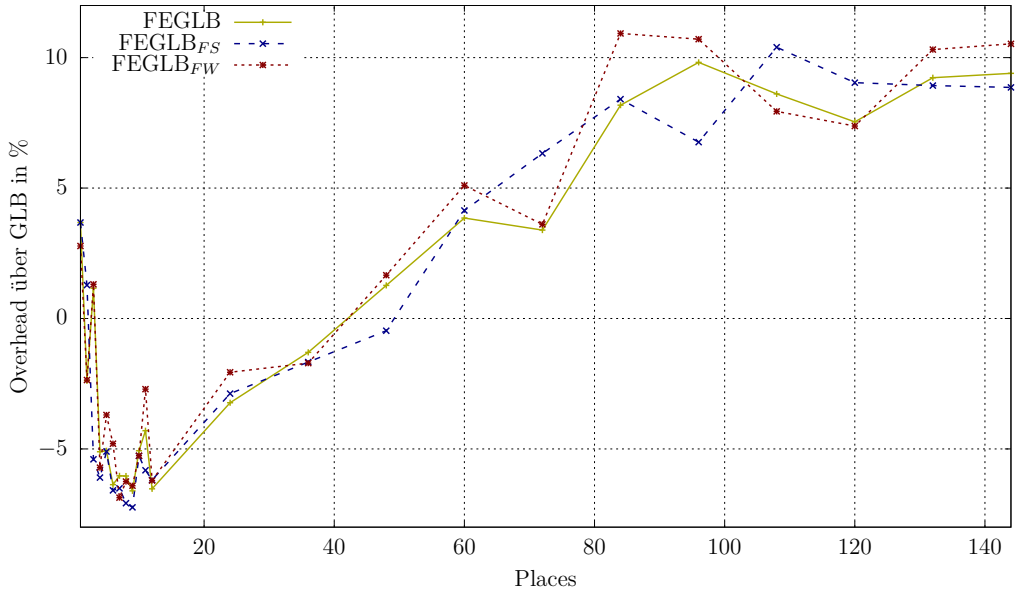


Abbildung 8.8.: BC_C: Overhead der FEGLB-Varianten gegenüber GLB

Framework	GLB	FEGLB	FEGLB$_{FS}$	FEGLB$_{FW}$
Speedup	52,06	50,12	50,37	49,17

Tabelle 8.4.: BC_C: Speedup mit 144 Places.

FEGLB hat bis 36 Places eine Beschleunigung gegenüber GLB von bis zu $6,61\%$ (bei 9 Places). Der maximale Overhead wird bei 144 Places mit $9,40\%$ erreicht. Betrachten wir die Anzahl der Stehl-Backups und der regulären Backups mit 144 Places (circa 550 Stehl-Backups und 1 200 reguläre Backups), sehen wir, dass die regulären Backups dominieren. Zusammen mit der Umstellung auf das aktorenartige Schema erklärt dies die gute Performance von FEGLB. Der Speedup von FEGLB weicht um $3,73\%$ von GLB ab.

FEGLB$_{FS}$ erreicht bis einschließlich 48 Places einen Beschleunigung gegenüber GLB von bis zu $7,24\%$. Bei 144 Places hat FEGLB$_{FS}$ von allen Frameworks den geringsten Overhead. Im Gegensatz zu UTS werden bei BC zur Laufzeit keine neuen Tasks erzeugt. Zudem stellt die Startverteilung eine gute Anfangsverteilung dar. Die Lastenbalancierung kommt somit erst gegen Ende des Programmlaufs zum Tragen. In diesem Fall hat das vorausschauende Stehlen Vorteile gegenüber der beiden anderen Varianten. Bei 108 Places erreicht der Overhead von FEGLB$_{FS}$ sein Maximum von $10,40\%$.

FEGLB$_{FW}$ erreicht bis 36 Places eine Beschleunigung gegenüber GLB von bis zu $6,87\%$. Bei 84 Places erreicht der Overhead von FEGLB$_{FW}$ mit $10,93\%$ sein Maximum.

BC$_{Ex}$ Abbildung 8.9 vergleicht den Overhead von FEGLB und seinen Varianten gegenüber GLB. Die Laufzeiten finden sich in den Tabellen A.5 (1 bis 12 Places) und A.13 (24 bis 144 Places). Wie auch bei BC$_C$ sind die Speedups in Tabelle 8.5 klein.

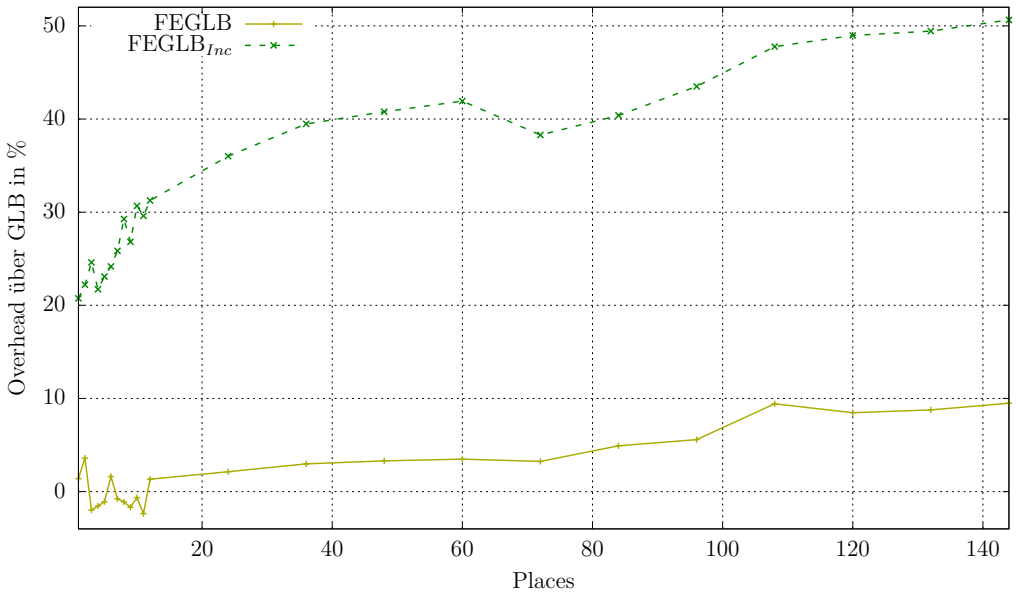


Abbildung 8.9.: BC$_{Ex}$: Overhead der FEGLB-Varianten gegenüber GLB

FEGLB zeigt gelegentliche Beschleunigungen von bis zu $2,35\%$ mit bis zu 12 Places. Der Overhead ist ähnlich zu dem der BC$_C$ Läufe und erreicht sein

doi:10.17170/kobra-2018122577

Framework	GLB	FEGLB	FEGLB_{Inc}
Speedup	52,52	48,64	42,10

Tabelle 8.5.: BC_{Ex}: Speedup mit 144 Places.

Maximum von $9,49\%$ mit 144 Places. Wie auch bei BC_C ist die Stehlrate gering, wodurch sich der allgemein gute Overhead für BC_{Ex} erklärt.

FEGLB_{Inc} hat einen auffällig hohen Overhead. Für BC_{Ex} ist der Overhead höher als für UTS_{Ex}. Er liegt zwischen $20,74\%$ (mit einem Place) und $50,64\%$ (mit 144 Places). Wie auch schon bei UTS_{Ex} vermuten wir, dass die Protokollierung des Minimalzustandes den Overhead verursacht. Bei BC_{Ex} kommt hinzu, dass das lokale Ergebnis eines Workers ein `Rail` der Länge $\texttt{N} = 2^{17}$ ist, welches kopiert werden muss, wenn ein neuer minimaler Taskpool protokolliert wird.

8.4.4. NQueens-Benchmark

Der Overhead der Frameworks gegenüber GLB ist in Abbildung 8.10 gezeigt, die Laufzeiten für 1 bis 12 bzw. 24 bis 144 Places sind in den Tabellen A.5 und A.13 zu finden, und die Speedups sind in Tabelle 8.6 zu finden.

Framework	GLB	FEGLB	FEGLB_{FS}	FEGLB_{FW}	FEGLB_{Inc}
Speedup	137,73	131,11	126,72	130,62	128,73

Tabelle 8.6.: NQueens: Speedup mit 144 Places.

Mit 3 und 11 Places hat FEGLB eine Beschleunigung von $0,22\%$ bzw. $0,19\%$ gegenüber GLB. Für die restlichen Läufe liegt der Overhead zwischen $0,17\%$ (mit 2 Places) und $5,55\%$ (mit 144 Places). Das Verhältnis von Stehl-Backups zu regulären Backups ähnelt dem der BC_C Läufe. Der Speedup von FEGLB auf 144 Places weicht um $4,81\%$ vom Speedup des GLB-Frameworks ab. Bei der Betrachtung der absoluten Laufzeiten aller Benchmarks fällt auf, dass die Ausführungszeit von NQueens wesentlich höher ist als die der restlichen Benchmarks. Konkret betragen die Laufzeiten mit 144 Places bei UTS_C circa 60 Sekunden, bei UTS_{Ex} circa 75 Sekunden, bei BC_C circa 50 Sekunden, bei BC_{Ex} circa 55 und bei NQueens circa 150 Sekunden. Die langen Laufzeiten könnte ein Grund für den geringen Overhead sein, da es eine lange Rechenphase mit vergleichsweise wenig Kommunikation gibt.

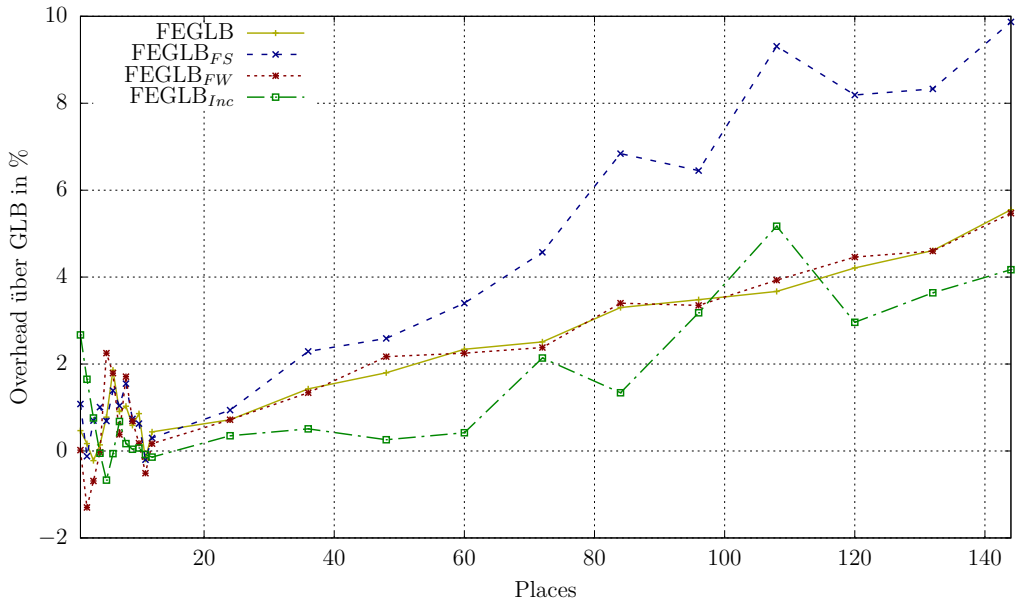


Abbildung 8.10.: NQueens: Overhead der FEGLB-Varianten gegenüber GLB

Der Overhead aller Frameworks lag bei allen Messungen unter 10%. FEGLB$_{FS}$ hat den höchsten Overhead. Wir vermuten, dass, wie bei UTS, wiederholtes Stehlen in der Terminierungsphase der Grund für den Overhead ist, jedoch schlägt sich dieses nicht in der Anzahl der Stehlanfragen oder der gesendeten Stehl-Backups nieder.

FEGLB und FEGLB$_{FW}$ zeigen erneut ein ähnliches Laufzeitverhalten. FEGLB$_{Inc}$ zeigt in diesem Benchmark das beste Laufzeitverhalten. Im Gegensatz zu UTS und BC wurde NQueens mit GLB-Parameter `n` = 1 gestartet (siehe Tabelle 8.1). FEGLB$_{Inc}$ entsteht somit ein vergleichsweise geringer Mehraufwand durch das Protokollieren der Minimalzustände. Zeitgleich ist das lokale Ergebnis jedes Workers bei NQueens ein einziger `Long`-Wert.

8.4.5. Aufwand für Wiederherstellung und Elastizität

Zielsetzung der folgenden Laufzeitmessungen war die Ermittlung des Protokollaufwandes für

- die Wiederherstellung ausgefallener, bzw.
- das Hinzufügen neuer Places.

doi:10.17170/kobra-2018122577

Hierzu haben wir drei Szenarien entwickelt, die uns erlauben die durch ausgefallene bzw. hinzugefügte Places verlorene bzw. gewonnene Rechenleistung herauszurechnen. Die Szenarien wurden mit FEGLB und UTS_C durchgeführt. Wir erklären den Aufbau der Szenarien nachfolgend. Eine Analyse der erhobenen Daten ist am Ende dieses Abschnitts zu finden.

FT Szenario: Messung des Protokollaufwandes für die Wiederherstellung In früheren Experimenten [40, 41, 45, 42] haben wir beobachtet, dass der Aufwand zur Wiederherstellung einer geringen Zahl von Places vernachlässigbar ist. Wir haben daher 24 Places (von insgesamt 120 Places) auf 2 Rechenknoten ausfallen lassen. Die 24 Places wurden so gewählt, dass kein endgültiger Datenverlust auftrat. Um eine möglichst typische Ausfallsituation zu simulieren, haben wir die Places nach ungefähr der Hälfte der erwarteten Laufzeit (also nach circa $78,25/2 \approx 39$ Sekunden, vgl. Tabelle A.11) abstürzen lassen. Dies geschah durch eine Modifikation der Hauptarbeitsschleife: Der Programmlauf wurde auf den entsprechenden Places nach 39 Sekunden durch einen Aufruf von `Runtime.killHere()` abgebrochen.

Der GLB-Parameter `l` wurde auf 12 gesetzt um eine gezielte Degeneration des Lifeline-Graphen durch die Placeausfälle zu erzwingen (siehe Abschnitt 4.4.3), sodass die Programmausführung auch eine Rekonstruktion des Lifeline-Graphen umfasst.

Die Programmlaufzeit bei initial 120 Places und Absturz von 24 Places nach der Hälfte der Laufzeit sollte im Bereich der Programmlaufzeit von 108 Places ohne Placeausfälle liegen.

EL Szenario: Messung des Protokollaufwandes für Elastizität Einen ähnlichen Aufbau wurde für die Messung von elastischen Programmläufen genutzt: UTS_C wurde mit 120 Places gestartet und nach der Hälfte der Laufzeit (also circa 39 Sekunden) wurden 24 Places auf 2 Rechenknoten dem Programmlauf hinzugefügt. Das Hinzufügen der 24 Places auf 2 Knoten geschah manuell über einen Start des Programms mit entsprechend gesetzten Umgebungsvariablen. Die Laufzeit sollte ähnlich zu UTS_C auf 132 Places ohne Erweiterung sein.

FT+EL Szenario: Messung des Protokollaufwandes für Fehlertoleranz und Elastizität Zur Auswertung der gemeinsamen Nutzung von Fehlertoleranz und Elastizität wurden die beiden obigen Testaufbauten kombiniert:

UTS_C wurde mit 120 Places gestartet. Wir haben nach 39 Sekunden 24 Places abstürzen lassen und fügten nach Ausfall der 24 Places 24 neue Places auf 2 neuen Rechenknoten dem Programmlauf hinzu. Die Laufzeit sollte ähnlich zu der Laufzeit von UTS_C mit 120 Places sein.

Auswertung Tabelle 8.7 zeigt die Laufzeiten der oben beschriebenen Experimente. In den Spalten sind von links nach rechts die folgenden Werte gezeigt:

- Die Laufzeit für UTS_C mit 108 Places und ohne Ausfall,
- die Laufzeit für UTS_C mit 120 Places und ohne Ausfall,
- die Laufzeit für UTS_C mit 132 Places und ohne Ausfall,
- die Laufzeit für UTS_C mit 120 Places und 24 Placeausfällen nach ca. 39 Sekunden (FT Szenario),
- die Laufzeit für UTS_C mit 12 Places und 24 elastisch Erweiterung um 24 Places auf 2 neuen Knoten nach 39 Sekunden (EL Szenario), und
- die Laufzeit für UTS_C mit 120 Places, 24 Placeausfällen nach 39 Sekunden und anschließender elastischer Erweiterung um 24 neue Places auf 2 neuen Knoten (FT+EL Szenario).

Places	108	120	132	120 - 24	120 + 24	120 - 24 + 24
Laufzeit	85,22	78,25	68,46	87,36	70,62	81,72

Tabelle 8.7.: Laufzeiten in Sekunden des UTS_C-Benchmarks bei Placeausfällen und elastischer Erweiterung. Gezeigt sind die Laufzeiten für UTS_C mit 108, 120 und 132 Places ohne Ausfälle und elastischer Erweiterungen, sowie des FT- $(120 - 24)$, EL- $(120 + 24)$ und FT+EL Szenarios $(120 - 24 + 24)$.

Das FT Szenario hat im Vergleich zu UTS_C mit 108 Places ohne Placeausfälle einen Overhead von $2,51\%$. Die absolute Differenz der Ausführungszeiten zwischen den UTS_C-Läufen mit 108 Places und dem FT Szenario liegt bei circa $2,14$ Sekunden. Da der Parameter k so eingestellt wurde, dass circa ein Backup pro Sekunde geschrieben wird, müssen im schlimmsten Fall pro

 doi:10.17170/kobra-2018122577

ausgefallenen Place Tasks mit einer Gesamtrechenzeit von höchstens einer Sekunde neu berechnet werden. Die Verteilung der Tasks zeigt, dass Places, die das Backup eines anderen Places eingespielt haben, ungefähr 56% mehr Tasks berechnet haben, als Places, die nicht an der Wiederherstellung beteiligt waren. Hierbei zählen Tasks, die aus dem Backup eines anderen Places übernommen wurden, ebenfalls als „berechnet “. Sonst ist die Taskverteilung über die Places hinweg gleichmäßig. Die Reparatur des Lifeline-Graphen war somit erfolgreich.

Das EL Szenario weist einen Overhad von $3,16\%$ im Vergleich zu UTS_C mit 132 Places auf. Die absolute Differenz der Laufzeiten beträgt $2,16$ Sekunden. Diese Differenz beinhaltet

- das Warten weiterer Places vor der Ausführung des Elastizitätsprotokolls (siehe Abschnitt 5.1),
- das eigentlich Ausführen des Protokolls, sowie
- eine Häufung von Stehlanfragen nach Hinzufügen der neuen Places.

Die Zeit zwischen dem elastischen Hinzufügen der Places zum Programmlauf und der Ausführung des Elastizitätsprotokolls beträgt 1 Sekunde. Werden die Places in Sekunde 39 zum Programmlauf hinzugefügt, so werden sie erst in Sekunde 40 in die Berechnung eingebunden. Diese Zeitspanne kann vernachlässigt werden, da hierdurch 24 Rechensekunden (eine Sekunde pro hinzugefügtem Place) verloren gehen. Der Programmlauf hat jedoch in der Summe rund $(70,62 \cdot 120) + ((70,62 - 39) \cdot 24) \approx 9\,233$ Rechensekunden benötigt. Der erste Summand gibt die Rechensekunden für die 120 initial gestarteten Places an. Der zweite Summand gibt die verbrauchten Rechensekunde der 24 Places an, welche nach circa 39 Sekunden der Berechnung hinzugefügt wurden. Die absolute Differenz zwischen UTS_C und dem EL Szenario beinhaltet

- die Ausführung des Elastizitätsprotokolls und
- den durch die elastische Hinzufügung induzierten Stehlanfragen.

Die Taskverteilung zeigt, dass ein neu hinzugefügter Worker circa 55% weniger Tasks berechnet hat als ein Place, der von Beginn an in die Berechnung eingebunden war. Die Verteilung der Tasks ist sonst gleichmäßig.

Das FT+EL Szenario hat einen Overhead von $4,43\%$. Die absolute Differenz der Laufzeiten von UTS_C mit 120 Places und den F+E Experimenten

beträgt $3,47$ Sekunden. Vergleichen wir diese Differenz mit den Differenzen der vorangegangenen Experimente, so ist dies ein plausibler Wert.

8.4.6. Abschließende Zusammenfassung

Wir schließen die experimentelle Auswertung mit den folgenden Beobachtungen:

- FEGLB hatte mit 144 Places meist einen Overhead von unter 10%. Nur mit UTS_C und 144 Places stieg der Overhead über 10%.
- Über alle Benchmarks hinweg boten FEGLB und FEGLB_{FW} im Schnitt die beste Performance.
- FEGLB_{FS} hatte Vorteile bei BC_C. Mit 144 Places hatte FEGLB_{FS} bei diesem Benchmark einen Overhead von $8,86\%$. Bei den anderen Benchmarks lag der Overhead über dem von FEGLB.
- FEGLB_{Inc} hatte mit NQueens und 144 Places einen Overhead von $4,17\%$. Der Overhead mit UTS_{Ex} und BC_{Ex} lag deutlich über dem Overhead von FEGLB. Der Overhead von FEGLB_{Inc} lag bei diesen beiden Benchmarks meist über 20%.

  Fohry, Posner und Reitz [86] haben mit ähnlichen inkrementellen und nicht-inkrementellen Backup-Algorithmen ähnliche Experimente mit UTS und BC durchgeführt. Sie zeigen ebenfalls, dass ihr inkrementeller Algorithmus bei diesen Benchmarks einen größeren Overhead als ihr nicht-inkrementeller Algorithmus hat. Die Differenz zwischen den Varianten ist jedoch deutlich geringer. Der Algorithmus verwendet die APGAS-Bibliothek für Java sowie Hazelcast zur Implementierung des Algorithmus. Insbesondere wird die `IMap` von Hazelcast zur redundanten Speicherung der Backup-Daten verwendet. Durch den Einsatz von Hazelcast kommt der Algorithmus mit weniger Nachrichten als unser Algorithmus aus. Diese Unterschiede könnten ein Grund für die Diskrepanz zwischen den zitierten und unseren Experimenten sein.
- Ausführungen des Wiederherstellungsprotokolls führen zu einem vernachlässigbaren Overhead.
- Ausführungen des Elastizitätsprotokolls führen ebenso zu einem vernachlässigbaren Overhead.

 doi:10.17170/kobra-2018122577

9. Zusammenfassung und Ausblick

In dieser Arbeit haben wir einen Algorithmus entwickelt, der Fehlertoleranz und Elastizität für Taskpools realisiert. Wir haben dabei als Grundlage den Lifeline-based Global Load Balancing Algorithmus [35] genutzt. Die Implementierung des Algorithmus erfolgte in der parallelen Programmiersprache X10 in Form eines Frameworks.

Die Fehlertoleranz haben wir über das Sichern der berechnungskritischen Daten, also der lokale Taskpools sowie das lokale Ergebnis jedes Workers, in den Hauptspeicher eines anderen Workers realisiert. Die Worker haben wir hierzu in einem Backup-Ring angeordnet, sodass jeder Worker an genau einen anderen Worker Backup-Daten sendet und Backup-Daten von genau einem anderen Worker speichert.

Der vorgestellte Algorithmus hält die Anzahl der durch einen Ausfall neu zu berechnenden Tasks gering, indem jeder Worker in periodischen Abständen ein neues Backup sendet, welches das alte Backup ersetzt.

Zur konsistenten Datenhaltung der Backups, insbesondere während Tasks im Zuge von Stehlanfragen von einem Worker zu einem anderen Worker gesendet werden, haben wir ein auf den Stehlvorgang zugeschnittenes Protokoll entwickelt.

Wir haben verschiedene Techniken vorgestellt, mit denen unser Algorithmus die Lebendigkeit der Worker überwacht, um zeitnah auf Ausfälle reagieren zu können. Treten Ausfälle auf, so ist der Algorithmus in der Lage zu erkennen, ob eine Überführung in einen konsistenten Programmzustand möglich ist. Falls dies möglich ist, so werden die durch den Ausfall verloren gegangenen Tasks und Teilergebnisse in den Taskpool eines Workers übernommen. Für diese Aufgabe haben wir ein Wiederherstellungsprotokoll entwickelt. Das Wiederherstellungsprotokoll ist in der Lage, weitere Ausfällen vor Erreichen eines konsitenten Programmzustandes durch verschachtelte Ausführungen zu

behandeln.

Elastizität haben wir in unserem Algorithmus über eine Erweiterung des Backup-Rings umgesetzt. Während der Erweiterung halten wir die Kommunikation zwischen bereits eingebundenen und neuen Workern minimal, um Effizienz zu gewährleisten. Während der Erweiterung des Backup-Rings muss das Backup eines alten Places auf einen neuen Place übertragen werden. Zudem müssen die neuen Places in die Lastenbalancierung des Taskpools integriert werden. Dies übernimmt das entworfene Elastizitätsprotokoll. Durch eine Verzahnung der Wiederherstellungs- und Elastizitätsprotokolle kann unser Algorithmus auch auf Ausfälle von alten und neuen Workern, während seiner Ausführung reagieren und behandelt diese korrekt.

Wir haben die Varianten vorausschauendes Stehlen, Stehlweiterleitung und ein inkrementelles Backupschema vorgestellt.

Die MPI-Umsetzung der Netzwerkschnittstelle von X10 unterstützte bisher keine Elastizität. Wir haben sie so modifiziert, dass sie Elastizität bereitstellt. Die initiale Verbindung zwischen Client- und Server-Seite haben wir über einen dedizierten Netzwerkport realisiert. Mit Hilfe der MPI Client-Server-Routinen wurde ein neuer Kommunikator zwischen alten und neuen Places erstellt. Der Austausch des Kommunikators erfolgt zu einem passenden Zeitpunkt, wenn über den bis dahin verwendeten Kommunikator keine Nachrichten mehr gesendet werden.

Wir haben eine experimentelle Auswertung unseres Frameworks mit drei Benchmarks vorgenommen. Im Zentrum der Auswertung stand ein Performancevergleich mit GLB, wenn keine Ausfälle auftreten. Daneben haben wir analysiert, welchen Overhad Ausfälle und das elastische Hinzufügen von Places verursachen.

Bei dem Vergleich mit GLB hatten unser Framework und seine Varianten über alle Benchmarks hinweg einen Overhead von $4,17\%$ bis $55,72\%$. Wir haben Stärken und Schwächen der einzelnen Varianten aufgezeigt. Für Wiederherstellungen ausgefallener Worker und das elastische Hinzufügen von Workern haben wir keinen nennenswerten Overhead feststellen können.

In zukünftigen Arbeiten wäre eine weitere experimentelle Analyse des Frameworks sinnvoll. Vor allem Ausführungen auf mehr Rechenknoten und mit mehr Places könnten genaueren Aufschluss über die Skalierbarkeit unseres

 doi:10.17170/kobra-2018122577

Algorithmus geben. Eine breitere Testbasis mit mehr Benchmarks ist ebenfalls wünschenswert. Von Interesse ist der Vergleich unseres Frameworks mit einem rein in MPI bzw. ULFM implementierten System oder einem anderen, fehlertoleranten Programmiersystem, beispielsweise GASPI.

Auf algorithmischer Ebene wäre eine Modifikation, die placeinterne Parallelität zulässt, denkbar. Somit könnte pro Rechenknoten ein Place, jedoch mit mehreren Workerthreads gestartet werden. Denkbar wäre sowohl die Haltung eines lokalen Pools für alle Worker auf einem Knoten als auch die Nutzung eines lokalen Pools pro Workerthread. An die zweite Möglichkeit gekoppelt wäre ein hierarchisches Work-Stealing wünschenswert: Bevor Worker von entfernten Workern stehlen, versuchen sie zuerst, bei lokalen Workern zu stehlen. Unabhängig hiervon wäre eine Umstellung der Backup-Intervalle möglich: Anstatt die Backup-Intervalle über eine Anzahl abgearbeiteter Tasks zu regeln, können diese zeitgesteuert geschrieben werden.

Die Verifikation des Algorithmus wäre ein weiterer möglicher Aspekt zukünftiger Arbeiten. Eine Unterstützung durch Tools, beispielsweise Spin [115] wäre dabei zu empfehlen.

Da ULFM seit Version 2.0 auch den Threading-Modus `MPI_THREAD_MULTIPLE` zulässt, könnte dadurch die elastische Erweiterung der MPI-Schnittstelle vereinfacht werden. Hierzu wäre jedoch auch zu prüfen, ob alle Teile der MPI-Schnittstelle (besonders bezüglich der Fehlertoleranz) eine parallele Ausführung der Kommunikationsroutinen unterstützen, und wenn nicht, wie diese anzupassen wären.

A. Laufzeittabellen

Places	**GLB**	**FEGLB**	**FEGLB$_{FS}$**	**FEGLB$_{FW}$**
1	7456,21	7487,19	7491,64	7443,50
2	3589,62	3834,19	3931,69	3822,56
3	2577,08	2763,91	2825,81	2772,52
4	1820,31	1953,74	1984,89	1948,09
5	1600,24	1656,59	1693,26	1655,87
6	1285,21	1328,05	1377,69	1326,60
7	1126,63	1183,67	1248,45	1184,43
8	978,49	1006,89	1044,12	1006,79
9	887,04	921,80	965,28	922,87
10	798,02	825,75	866,97	827,24
11	735,33	756,11	795,44	755,71
12	626,67	640,65	718,61	641,41

Tabelle A.1.: UTS$_C$-Benchmark: Laufzeiten in Sekunden auf einem Rechenknoten mit 1 bis 12 Places.

Places	**GLB**	**FEGLB**	**FEGLB$_{FS}$**	**FEGLB$_{FW}$**
1	7456,21	7487,19	7491,64	7443,50
2	3613,69	3833,84	3932,71	3831,05
3	2403,78	2566,70	2795,28	2565,55
4	1799,49	1921,73	2044,06	1912,24
5	1499,90	1540,67	1775,90	1542,30
6	1238,95	1279,00	1313,70	1275,57
7	1067,06	1096,95	1169,02	1096,61
8	938,25	960,55	1082,85	959,94
9	831,32	852,07	904,38	850,73
10	748,13	772,58	804,06	772,17
11	684,02	702,53	793,76	700,19
12	619,07	642,34	679,50	642,91

Tabelle A.2.: UTS$_C$-Benchmark: Laufzeiten in Sekunden mit einem Place pro Rechenknoten und 1 bis 12 Places.

Places	**GLB**	**FEGLB**	**FEGLB$_{Inc}$**
1	7743,78	7876,97	8925,48
2	3997,05	4154,93	4481,89
3	2788,38	2973,53	3212,22
4	1986,46	2101,87	2346,21
5	1679,10	1757,35	2016,44
6	1355,49	1426,65	1621,44
7	1198,92	1267,14	1453,45
8	1030,45	1094,33	1259,00
9	938,58	988,79	1146,57
10	839,48	891,86	1035,58
11	759,92	789,41	929,76
12	701,57	723,74	864,55

Tabelle A.3.: UTS$_{Ex}$-Benchmark: Laufzeiten in Sekunden auf einem Rechenknoten mit 1 bis 12 Places.

doi:10.17170/kobra-2018122577

Places	**GLB**	**FEGLB**	**FEGLB$_{Inc}$**
1	7743,78	7876,97	8925,48
2	3869,30	4186,09	4527,15
3	2599,70	2994,35	3290,92
4	1954,33	2172,92	2412,84
5	1568,02	1826,41	2083,18
6	1307,88	1476,30	1674,62
7	1123,19	1324,79	1504,61
8	982,81	1136,90	1312,63
9	877,54	1027,26	1205,73
10	790,22	911,30	1104,44
11	719,27	808,98	981,09
12	661,31	735,47	936,39

Tabelle A.4.: UTS$_{Ex}$-Benchmark: Laufzeiten in Sekunden mit einem Place pro Rechenknoten und 1 bis 12 Places.

Places	**GLB**	**FEGLB**	**FEGLB$_{FS}$**	**FEGLB$_{FW}$**
1	2590,06	2685,28	2685,28	2661,96
2	1304,79	1273,58	1321,46	1273,95
3	1048,16	1060,29	991,53	1061,91
4	827,84	785,50	777,36	780,52
5	773,26	734,07	733,81	744,66
6	685,06	641,39	639,94	652,15
7	667,58	627,30	624,15	621,72
8	618,21	580,85	574,45	579,56
9	613,11	572,59	568,69	573,80
10	569,03	540,16	539,02	539,12
11	569,89	545,39	536,73	554,45
12	543,17	507,70	509,57	509,44

Tabelle A.5.: BC$_C$-Benchmark: Laufzeiten in Sekunden auf einem Rechenknoten mit 1 bis 12 Places.

Places	**GLB**	**FEGLB**	**FEGLB$_{FS}$**	**FEGLB$_{FW}$**
1	2590,06	2685,28	2685,28	2661,96
2	1303,29	1231,15	1297,25	1228,21
3	862,41	809,60	870,84	809,68
4	645,04	608,19	606,12	609,62
5	518,21	483,20	484,45	484,11
6	430,24	406,68	403,64	404,79
7	367,81	347,56	345,21	347,59
8	322,46	301,98	303,01	302,61
9	288,40	268,49	269,76	269,20
10	259,84	242,87	241,60	243,06
11	235,40	220,76	223,41	219,15
12	216,54	207,85	202,55	202,97

Tabelle A.6.: BC$_C$-Benchmark: Laufzeiten in Sekunden mit einem Place pro Rechenknoten und 1 bis 12 Places.

Places	**GLB**	**FEGLB**	**FEGLB$_{Inc}$**
1	2654,71	2691,91	3205,23
2	1331,07	1378,89	1626,69
3	1107,73	1085,80	1380,35
4	852,68	839,52	1037,91
5	807,20	798,43	993,57
6	703,71	715,11	873,80
7	688,40	683,12	866,33
8	635,30	628,31	821,26
9	626,39	615,79	794,39
10	580,29	576,53	758,28
11	580,29	566,65	752,06
12	542,87	550,07	712,55

Tabelle A.7.: BC$_{Ex}$-Benchmark: Laufzeiten in Sekunden auf einem Rechenknoten mit 1 bis 12 Places.

doi:10.17170/kobra-2018122577

Places	**GLB**	**FEGLB**	**FEGLB$_{Inc}$**
1	2654,71	2691,91	3205,23
2	1332,74	1372,36	1652,15
3	887,85	890,55	1092,42
4	667,25	653,08	808,25
5	532,51	542,00	668,77
6	440,12	426,62	552,22
7	377,73	395,67	496,06
8	330,56	340,00	449,38
9	295,03	292,37	392,02
10	265,74	262,81	350,11
11	242,60	235,86	320,80
12	221,54	210,73	280,77

Tabelle A.8.: BC$_{Ex}$-Benchmark: Laufzeiten in Sekunden mit einem Place pro Rechenknoten und 1 bis 12 Places.

Places	**GLB**	**FEGLB**	**FEGLB$_{FS}$**	**FEGLB$_{FW}$**	**FEGLB$_{Inc}$**
1	19456,38	19548,00	19666,99	19460,07	19979,07
2	9888,53	9905,38	9876,71	9759,78	9920,81
3	7055,67	7040,22	7105,14	7006,77	7060,16
4	4978,84	4985,81	5029,15	4977,49	4975,19
5	4226,76	4259,67	4256,10	4322,05	4293,10
6	3344,24	3406,54	3390,73	3404,05	3401,97
7	3032,58	3060,08	3064,15	3044,16	3064,95
8	2518,39	2544,38	2557,35	2561,46	2565,78
9	2345,38	2359,25	2362,63	2361,57	2362,62
10	2028,60	2046,15	2041,37	2032,24	2033,65
11	1943,73	1940,05	1939,89	1933,79	1932,30
12	1690,32	1697,72	1695,33	1693,18	1690,76

Tabelle A.9.: NQueens-Benchmark: Laufzeiten in Sekunden auf einem Rechenknoten mit 1 bis 12 Places.

Places	GLB	FEGLB	FEGLB$_{FS}$	FEGLB$_{FW}$	FEGLB$_{Inc}$
1	19456,38	19548,00	19666,99	19460,07	19979,07
2	9886,97	9914,99	9969,05	9838,08	9778,56
3	6589,54	6497,54	6591,38	6952,17	6664,00
4	4788,78	4951,88	4933,53	4933,60	4957,33
5	3913,23	3926,13	3926,71	3928,75	3937,54
6	3264,11	3295,11	3284,45	3297,23	3302,20
7	2767,53	2842,48	2818,78	2866,54	2824,67
8	2465,39	2478,86	2471,03	2471,82	2468,77
9	2186,13	2177,97	2188,13	2173,41	2180,19
10	1962,86	1952,83	1960,33	1951,63	1962,98
11	1778,31	1777,59	1781,48	1776,45	1785,87
12	1633,34	1636,78	1637,66	1633,02	1634,81

Tabelle A.10.: NQueens-Benchmark: Laufzeiten in Sekunden mit einem Place pro Rechenknoten und 1 bis 12 Places.

Places	GLB	FEGLB	FEGLB$_{FS}$	FEGLB$_{FW}$
24	333,38	344,29	378,37	344,65
36	223,13	232,65	271,81	231,97
48	168,61	178,00	210,51	178,51
60	135,19	144,07	174,54	144,87
72	113,23	120,82	148,32	121,66
84	97,10	105,06	130,71	109,57
96	85,23	92,72	121,37	92,66
108	76,08	85,22	113,85	85,84
120	69,03	78,25	101,88	77,81
132	61,97	68,46	94,21	67,99
144	55,73	63,62	86,78	64,36

Tabelle A.11.: UTS$_C$-Benchmark: Laufzeiten in Sekunden mit 24 bis 144 Places.

doi:10.17170/kobra-2018122577

Places	GLB	FEGLB	FEGLB$_{Inc}$
24	359,92	373,60	465,63
36	246,39	255,55	309,96
48	190,24	201,10	244,38
60	153,99	163,55	197,99
72	134,44	143,01	173,91
84	117,28	126,46	154,67
96	100,40	109,40	135,57
108	158,20	174,50	206,83
120	84,18	94,18	113,13
132	76,05	84,37	108,27
144	70,14	76,25	97,97

Tabelle A.12.: UTS$_{Ex}$-Benchmark: Laufzeiten in Sekunden mit 24 bis 144 Places.

Places	GLB	FEGLB	FEGLB$_{FS}$	FEGLB$_{FW}$
24	269,89	261,17	262,13	264,32
36	181,51	179,16	178,48	178,41
48	138,13	139,88	137,48	140,42
60	111,31	115,59	115,91	116,98
72	94,37	97,57	100,34	97,77
84	81,19	87,84	88,03	90,07
96	71,83	78,88	76,69	79,52
108	64,62	70,18	71,34	69,75
120	58,72	63,14	64,02	63,05
132	53,42	58,36	58,19	58,93
144	48,97	53,58	53,31	54,13

Tabelle A.13.: BC$_C$-Benchmark: Laufzeiten in Sekunden mit 24 bis 144 Places.

Places	GLB	FEGLB	FEGLB$_{Inc}$
24	273,85	279,72	372,46
36	185,62	191,16	258,92
48	140,75	145,40	198,16
60	114,21	118,20	162,10
72	96,55	99,68	133,52
84	83,84	87,97	117,70
96	73,59	77,70	105,61
108	66,31	72,56	97,99
120	60,04	65,12	89,44
132	54,73	59,53	81,80
144	50,54	55,34	76,14

Tabelle A.14.: BC$_{Ex}$-Benchmark: Laufzeiten in Sekunden mit 24 bis 144 Places.

Places	GLB	FEGLB	FEGLB$_{FS}$	FEGLB$_{FW}$	FEGLB$_{Inc}$
24	842,18	848,25	850,13	848,25	851,19
36	561,46	569,49	574,30	568,98	571,90
48	421,26	428,87	432,17	430,39	431,49
60	337,08	344,98	348,53	344,66	346,11
72	281,49	288,55	294,36	288,20	294,36
84	240,98	248,94	257,47	249,17	252,52
96	210,91	218,25	224,52	217,98	224,92
108	187,97	194,87	205,46	195,36	205,46
120	169,09	176,21	182,95	176,64	181,87
132	154,03	161,13	166,86	161,11	166,96
144	141,26	149,10	155,20	148,99	155,20

Tabelle A.15.: NQueens-Benchmark: Laufzeiten in Sekunden mit 24 bis 144 Places.

doi:10.17170/kobra-2018122577

Literatur

[1] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk und Jesper Larsson Traff. *MPI at Exascale*. In: *Proceedings of the Scientific Discovery through Advanced Computing*. Bd. 2. 2010. URL: http://aegjcef.unixer.de/publications/img/mpi_exascale.pdf (besucht am 10. 09. 2018).

[2] Zizhong Chen und Jack Dongarra. *Algorithm-Based Fault Tolerance for Fail-Stop Failures*. In: *IEEE Transactions on Parallel and Distributed Systems* 19.12 (2008). DOI: 10.1109/tpds.2008.58.

[3] Jośe Carlos Sancho, Fabrizio Petrini, Greg Johnson und Juan Fernández. *On the Feasibility of Incremental Checkpointing for Scientific Computing*. In: *Proceedings of the 18th Symposium on Parallel and Distributed Computing*. 2004. DOI: 10.1109/IPDPS.2004.1302982.

[4] Adam Moody, Greg Bronevetsky, Kathryn Mohror und Bronis R. Supinski de. *Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System*. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010. DOI: 10.1109/SC.2010.18.

[5] Bianca Schroeder und Garth A. Gibson. *Understanding Failures in Petascale Computers*. In: *Journal of Physics: Conference Series* 78 (2007). DOI: 10.1088/1742-6596/78/1/012022.

[6] Kuang-Hua Huang und Jacob A. Abraham. *Algorithm-Based Fault Tolerance for Matrix Operations*. In: *IEEE Transactions on Computers* C-33.6 (1984). DOI: 10.1109/tc.1984.1676475.

[7] Jing-Yang Jou und Jacob A. Abraham. *Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures*. In: *Proceedings of the IEEE* 74.5 (1986). DOI: 10.1109/proc.1986.13535.

[8] Abhishek Gupta, Bilge Acun, Osman Sarood und Laxmikant V. Kale. *Towards Realizing the Potential of Malleable Jobs*. In: *Proceedings of the 21st International Conference on High Performance Computing*. 2014. DOI: 10.1109/hipc.2014.7116905.

[9] Dror G. Feitelson und Larry Rudolph. *Toward Convergence in Job Schedulers for Parallel Supercomputers*. In: *Job Scheduling Strategies for Parallel Processing*. 1996. DOI: 10.1007/bfb0022284.

[10] MPI Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. 2015. URL: https://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (besucht am 10. 09. 2018).

[11] *MPI Forum*. URL: https://mpi-forum.org/ (besucht am 10. 09. 2018).

[12] *Open MPI Homepage*. URL: https://www.open-mpi.org/ (besucht am 10. 09. 2018).

[13] *MPICH Homepage*. URL: https://www.mpich.org/ (besucht am 10. 09. 2018).

[14] *ULFM Standard (Entwurf)*. URL: https://fault-tolerance.org/wp-content/uploads/2012/10/20170221-ft.pdf (besucht am 10. 09. 2018).

[15] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca und Jack Dongarra. *Post-failure Recovery of MPI Communication Capability: Design and Rationale*. In: *International Journal High Performance Computing Applications* 27.3 (2013). DOI: 10.1177/1094342013488238.

[16] *ULFM git Repository*. URL: https://bitbucket.org/icldistcomp/ulfm (besucht am 10. 09. 2018).

[17] Bardford L. Chamberlain, David Callahan und Hans P. Zima. *Parallel Programmability and the Chapel Language*. In: *The International Journal of High Performance Computing Applications* 21.3 (2007). DOI: 10.1177/1094342007078442.

[18] *Chapel Homepage*. URL: https://chapel-lang.org/ (besucht am 10. 09. 2018).

 doi:10.17170/kobra-2018122577

[19] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun und Vivek Sarkar. *X10: An Object-oriented Approach to Non-uniform Cluster Computing*. In: *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 2005. DOI: 10.1145/1094811.1094852.

[20] *X10 Homepage*. URL: http://x10-lang.org/ (besucht am 10. 09. 2018).

[21] *X10 git Repository*. URL: https://github.com/x10-lang (besucht am 10. 09. 2018).

[22] Tarek El-Ghazawi und Lauren Smith. *UPC: Unified Parallel C*. In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. 2006. DOI: 10.1145/1188455.1188483.

[23] *UPC Homepage*. URL: https://upc-lang.org/ (besucht am 10. 09. 2018).

[24] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan und Katherine Yelick. *UPC++: A PGAS Extension for C++*. In: *Proceedings of the 28th International Parallel and Distributed Processing Symposium*. 2014. DOI: 10.1109/ipdps.2014.115.

[25] *UPC++ git Repository*. URL: https://bitbucket.org/berkeleylab/upcxx/wiki/Home (besucht am 10. 09. 2018).

[26] Olivier Tardieu. *The APGAS library: Resilient Parallel and Distributed Programming in Java 8*. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. 2015. DOI: 10.1145/2771774.2771780.

[27] Marek Nowicki, Lukasz Gorski, Patryk Grabrczyk und Piotr Bala. *PCJ - Java Library for High Performance Computing in PGAS Model*. In: *Proceedings of the International Conference on High Performance Computing and Simulation*. 2014. DOI: 10.1109/hpcsim.2014.6903687.

[28] *PCJ Homepage*. URL: https://pcj.icm.edu.pl/ (besucht am 10. 09. 2018).

[29] *PCJ git Repository*. URL: https://github.com/hpdcj/pcj (besucht am 10. 09. 2018).

[30] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi und Olivier Tardieu. *Resilient X10: Efficient Failure-Aware Programming*. In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2014. DOI: 10.1145/2555243.2555248.

[31] Michał Szynkiewicz und Marek Nowicki. *Fault-Tolerance Mechanisms for the Java Parallel Codes Implemented with the PCJ Library*. In: *Parallel Processing and Applied Mathematics*. 2018. DOI: 10.1007/978-3-319-78054-2_28.

[32] Reid Smith. *Panel on Design Methodology*. In: *ACM SIGPLAN Notices* 23.5 (1988). DOI: 10.1145/62139.62151.

[33] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. 1. Aufl. Pearson, 1995. ISBN: 9780201575941.

[34] Robert D. Blumofe und Charles E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing*. In: *Journal of the ACM* 46.5 (1999). DOI: 10.1145/324133.324234.

[35] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove und Sriram Krishnamoorthy. *Lifeline-based Global Load Balancing*. In: *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*. 2011. DOI: 10.1145/1941553.1941582.

[36] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat und Mikio Takeuchi. *GLB: Lifeline-based Global Load Balancing Library in X10*. In: *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*. 2014. DOI: 10.1145/2567634.2567639.

[37] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan und Chau-Wen Tseng. *UTS: An Unbalanced Tree Search Benchmark*. In: *Languages and Compilers for Parallel Computing*. 2. DOI: 10.1007/978-3-540-72521-3_18.

[38] *HPCS Scalable Synthetic Compact Applications #2: Graph Analysis*. URL: http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.0.pdf (besucht am 10. 09. 2018).

 doi:10.17170/kobra-2018122577

[39] Marco Bungart, Claudia Fohry und Jonas Posner. *Fault-Tolerant Global Load Balancing in X10*. In: *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2014. DOI: 10.1109/synasc.2014.69.

[40] Claudia Fohry, Marco Bungart und Jonas Posner. *Towards an Efficient Fault-Tolerance Scheme for GLB*. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. 2015. DOI: 10.1145/2771774.2771779.

[41] Claudia Fohry, Marco Bungart und Jonas Posner. *Fault Tolerance Schemes for Global Load Balancing in X10*. In: *Scalable Computing: Practice and Experience* 16.2 (2015). DOI: 10.12694/scpe.v16i2.1088.

[42] Claudia Fohry und Marco Bungart. *A Robust Fault Tolerance Scheme for Lifeline-Based Taskpools*. In: *Proceedings of the 45th International Conference on Parallel Processing Workshops*. 2016. DOI: 10.1109/icppw.2016.40.

[43] Marco Bungart und Claudia Fohry. *A Malleable and Fault-Tolerant Task Pool Framework for X10*. In: *Proceedings of the IEEE International Conference on Cluster Computing*. 2017. DOI: 10.1109/cluster.2017.27.

[44] Marco Bungart und Claudia Fohry. *Extending the MPI Backend of X10 by Elasticity*. EuroMPI Poster. 2017. URL: https://www.mcs.anl.gov/eurompi2017/pics/posters/Fohry-EuroMPI2017-posterX10.pdf (besucht am 10. 09. 2018).

[45] Claudia Fohry, Marco Bungart und Paul Plock. *Fault Tolerance for Lifeline-Based Global Load Balancing*. In: *Journal of Software Engineering and Applications* 10.13 (2017). DOI: 10.4236/jsea.2017.1013053.

[46] Jack Dongarra, Thomas Herault und Yves Robert. *Fault Tolerance Techniques for High-Performance Computing*. In: *Computer Communications and Networks*. 2015. DOI: 10.1007/978-3-319-20943-2_1.

[47] Saurabh Hukerika und Christian Engelmann. *Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale*. In: *Supercomputing Frontiers and Innovations* 4.3 (2017). DOI: 10.14529/jsfi170301.

[48] Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, Olaf Krzikalla, Edmund Kügeler, Carsten Lojewski, Guy Lonsdale, Ralph Müller-Pfefferkorn, Wolfgang Nagel, Lena Oden, Franz-Josef Pfreundt, Mirko Rahn, Michael Sattler, Mareike Schmidtobreick, Annika Schiller, Christian Simmendinger, Thomas Soddemann, Godehard Sutmann, Henning Weber und Jan-Philipp Weiss. *GASPI – A Partitioned Global Address Space Programming Interface*. In: *Facing the Multicore-Challenge III*. 2013. DOI: 10.1007/978-3-642-35893-7_18.

[49] Faisal Shahzad, Moritz Kreutzer, Thomas Zeiser, Rui Machado, Andreas Pieper, Georg Hager und Gerhard Wellein. *Building a Fault Tolerant Application using the GASPI Communication Layer*. In: *Proceedings of the IEEE International Conference on Cluster Computing*. 18. Mai 2015. arXiv: 1505.04628v1 [cs.DC].

[50] Sara S. Hamouda, Benjamin Herta, Josh Milthorpe, David Grove und Olivier Tardieu. *Resilient X10 over MPI User Level Failure Mitigation*. In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. 2016. DOI: 10.1145/2931028.2931030.

[51] David Grove, Sara S. Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthrope, Vijay Saraswat, Avraham Shinnar, Miko Takeuchi und Olivier Tardieu. *Failure Recovery in Resilient X10*. Research Report. IBM, 2017. URL: http://domino.watson.ibm.com/library/CyberDig.nsf/papers/F6FD6AF7798BA365852581780055018E/$File/rc25660.pdf (besucht am 10. 09. 2018).

[52] *Hazelcast Homepage*. URL: https://hazelcast.com/ (besucht am 10. 09. 2018).

[53] Rob V. Van Nieuwpoort, Gosia Wrzesińska, Ceriel J. H. Jacobs und Henri E. Bal. *Satin: a High-Level and Efficient Grid Programming Model*. In: *ACM Transactions on Programming Languages and Systems* 32.3 (2010). DOI: 10.1145/1709093.1709096.

[54] Robert D. Blumofe und Philip A. Lisiecki. *Adaptive and reliable parallel computing on networks of workstations*. In: *Proceedings of the Annual*

 doi:10.17170/kobra-2018122577

Conference on USENIX. 1997. URL: http://supertech.csail.mit.edu/papers/USENIX97.pdf (besucht am 10. 09. 2018).

[55] Matteo Frigo, Charles E. Leiserson und Keith H. Randall. *The Implementation of the Cilk-5 Multithreaded Language*. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1998. DOI: 10.1145/277650.277725.

[56] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang und David B. Johnson. *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*. In: *ACM Computing Surveys* 34.3 (2002). DOI: 10.1145/568522.568525.

[57] Luís Moura Silva und João Gabriel Silva. *System-Level versus User-Defined Checkpointing*. In: *Proceedings of the 17th Symposiumon Reliable Distributed Systems*. 1998. DOI: 10.1109/RELDIS.1998.740476.

[58] Joel F. Bartlett. *A NonStop kernel*. In: *Proceedings of the Eighth Symposium on Operating systems principles*. 1981. DOI: 10.1145/800216.806587.

[59] James S. Plank, Micah Beck, Gerry Kingsley und Kai Li. *Libckpt: Transparent Checkpointing under Unix*. In: *Proceedings of the Usenix Winter Technical Conference*. 1995. URL: https://courses.cs.vt.edu/~cs5204/fall07-kafura/Papers/FaultTolerance/LibCkpt.pdf (besucht am 10. 09. 2018).

[60] Wiliam R. Dieter und James E. Lumpp. *A User-Level Checkpointing Library for POSIX Threads Programs*. In: *Proceedings of the Annual International Symposium on Fault-Tolerant Computing*. 1999. DOI: 10.1109/ftcs.1999.781054.

[61] Jason Ansel, Kapil Arya und Gene Cooperman. *DMTCP: Transparent checkpointing for cluster computations and the desktop*. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 2009. DOI: 10.1109/ipdps.2009.5161063.

[62] Greg Bronevetsky, Daniel Marques, Keshav Pingali und Paul Stodghill. *Automated application-level checkpointing of MPI programs*. In: *ACM SIGPLAN Notices* 38.10 (2003). DOI: 10.1145/966049.781513.

[63] Kanianthra Mani Chandy und Leslie Lamport. *Distributed snapshots: determining global states of distributed systems*. In: *ACM Transactions on Computer Systems* 3.1 (1985). DOI: 10.1145/214451.214456.

[64] Yi-Min Wang, Yennun Huang und W. Kent Fuchs. *Progressive Retry for Software Error Recovery in Distributed Systems*. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1993. DOI: 10.1109/ftcs.1993.627317.

[65] Shen Gao, Bingsheng He und Jianliang Xu. *Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems*. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015. DOI: 10.1145/2751205.2751212.

[66] Esteban Meneses. *Scalable Message-Logging Techniques for Effective Fault Tolerance in HPC Applications*. Diss. University of Illinois at Urbana-Champaign, 2013. URL: https://charm.cs.illinois.edu/newPapers/13-17/paper.pdf (besucht am 10. 09. 2018).

[67] Esteban Meneses, Xiang Ni, Gengbin Zheng, Celso L. Mendes und Laxmikant V. Kale. *Using Migratable Objects to Enhance Fault Tolerance Schemes in Supercomputers*. In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015). DOI: 10.1109/tpds.2014.2342228.

[68] Brian Randell. *System Structure for Software Fault Tolerance*. In: *IEEE Transactions on Software Engineering* SE-1.2 (1975). DOI: 10.1109/tse.1975.6312842.

[69] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir und Franck Cappello. *Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications*. In: *Processing IEEE International Parallel and Distributed Symposium*. 2011. DOI: 10.1109/ipdps.2011.95.

[70] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen und Philip C. Roth. *Modeling the Impact of Checkpoints on Next-Generation Systems*. In: *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*. 2007. DOI: 10.1109/MSST.2007.4367962.

 doi:10.17170/kobra-2018122577

[71] Luís Moura Silva und João Gabriel Silva. *An Experimental Study about Diskless Checkpointing*. In: *Proceedings of the 24th EUROMICRO Conference*. 1998. DOI: 10.1109/eurmic.1998.711832.

[72] Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez und Franck Cappello. *Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications*. In: *Proceedings of the International Parallel and Distributed Processing Symposium*. 2014. DOI: 10.1109/ipdps.2014.122.

[73] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama und Satoshi Matsuoka. *FTI: High Performance Fault Tolerance Interface for Hybrid Systems*. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011. DOI: 10.1145/2063384.2063427.

[74] Anne Benoit, Aurelien Cavelan, Valentin Le Fevre, Yves Robert und Hongyang Sun. *Towards Optimal Multi-Level Checkpointing*. In: *IEEE Transactions on Computers* 66.7 (2017). DOI: 10.1109/tc.2016.2643660.

[75] John W. Young. *A First Order Approximation to the Optimum Checkpoint Interval*. In: *Communications of the ACM* 17.9 (1974). DOI: 10.1145/361147.361115.

[76] J.T. Daly. *A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps*. In: *Future Generation Computer Systems* 22.3 (2006). DOI: 10.1016/j.future.2004.11.016.

[77] Elmootazbellah Nabil Elnozahy, David B. Johnson und Willy Zwaenepoel. *The Performance of Consistent Checkpointing*. In: *Proceedings of the 11th Symposium on Reliable Distributed Systems*. 1992. DOI: 10.1109/reldis.1992.235144.

[78] Zizhong Chen und Jack Dongarra. *Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing*. In: *IEEE Transactions on Computers* 58.11 (2009). DOI: 10.1109/tc.2009.42.

[79] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz und Bronis R. de Supinski. *Evaluating User-Level Fault Tolerance for MPI Applications*. In: *Proceedings of the 21st European MPI Users' Group Meeting*. 2014. DOI: 10.1145/2642769.2642775.

[80] Zizhong Chen. *Algorithm-based Recovery for Iterative Methods without Checkpointing*. In: *Proceedings of the 20th international Symposium on High Performance Distributed Computing*. 2011. DOI: 10.1145/1996130.1996142.

[81] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding und Zizhong Chen. *High Performance Linpack Benchmark*. In: *Proceedings of the International Conference on Supercomputing*. 2011. DOI: 10.1145/1995896.1995923.

[82] James S. Plank, Youngbae Kim und Jack J. Dongarra. *Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations*. In: *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*. 1995. DOI: 10.1109/ftcs.1995.466964.

[83] Gosia Wrzesinska, Ana-Maria Oprescu, Thilo Kielmann und Henri Bal. *Persistent Fault-Tolerance for Divide-and-Conquer Applications on the Grid*. In: *Proceedings of the European Conference on Parallel Processing*. 2007. DOI: 10.1007/978-3-540-74466-5_46.

[84] Gokcen Kestor, Sriram Krishnamoorthy und Wenjing Ma. *Localized Fault Recovery for Nested Fork-Join Programs*. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 2017. DOI: 10.1109/ipdps.2017.75.

[85] Jonas Posner und Claudia Fohry. *A Java Task Pool Framework providing Fault-Tolerant Global Load Balancing*. In: *International Journal of Networking and Computing* 8.1 (2018). DOI: 10.15803/ijnc.8.1_2.

[86] Claudia Fohry, Jonas Posner und Lukas Reitz. *A Selective and Incremental Backup Scheme for Task Pools*. In: *Proceedings of the International Conference on High Performance Computing & Simulation*. im Erscheinen.

[87] *AllScale Homepage*. URL: http://www.allscale.eu/home (besucht am 10. 09. 2018).

[88] *AllScale git Repository*. URL: https://github.com/allscale (besucht am 10. 09. 2018).

 doi:10.17170/kobra-2018122577

[89] Laxmikant V. Kale und Sanjeev Krishnan. *CHARM++: A Portable Concurrent Object Oriented System based on C++*. In: *Proceedings of the of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. 1993. DOI: 10.1145/165854.165874.

[90] *CHARM++ Homepage*. URL: https://charm.cs.illinois.edu/ (besucht am 10. 09. 2018).

[91] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. Diss. Royal Institute of Technology, 2003. URL: http://erlang.org/download/armstrong_thesis_2003.pdf (besucht am 10. 09. 2018).

[92] *Erlang Homepage*. URL: https://www.erlang.org/ (besucht am 10. 09. 2018).

[93] Tatiana Martsinkevich, Omer Subasi, Osman Unsal, Franck Cappello und Jesus Labarta. *Fault-Tolerant Protocol for Hybrid Task-Parallel Message-Passing Applications*. In: *IEEE International Conference on Cluster Computing*. 2015. DOI: 10.1109/cluster.2015.104.

[94] *OmpSs Homepage*. URL: https://pm.bsc.es/ompss (besucht am 10. 09. 2018).

[95] Jeffrey D. Ullman. *NP-Complete Scheduling Problems*. In: *Journal of Computer and System Sciences* 10.3 (1975). DOI: 10.1016/s0022-0000(75)80008-0.

[96] *Torque Homepage*. URL: https://www.adaptivecomputing.com/products/torque/ (besucht am 10. 09. 2018).

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

[98] Jeremy Buisson, Ozan Sonmez, Hashim Mohamed, Wouter Lammers und Dick Epema. *Scheduling Malleable Applications in Multicluster Systems*. In: *Proceedings of the IEEE International Conference on Cluster Computing*. 2007. DOI: 10.1109/CLUSTR.2007.4629252.

[99] *Koala Homepage*. URL: https://ds.st.ewi.tudelft.nl/koala/ (besucht am 10. 09. 2018).

[100] Rajesh Sudarsan und Calvin J. Ribbens. *Combining performance and priority for scheduling resizable parallel applications*. In: *Journal of Parallel and Distributed Computing* 87 (2016). DOI: 10.1016/j.jpdc.2015.09.007.

[101] Claudia Leopold, Michael Süß und Jens Breitbart. *Programming for Malleability with Hybrid MPI-2 and OpenMP - Experiences with a Simulation Program for Global Water Prognosis*. In: *Proceedings of the European Conference on Modelling and Simulation*. 2006. URL: https://michaelsuess.net/michaelsuess/publications/leopold_suess_breitbart_malleability_06.pdf (besucht am 10. 09. 2018).

[102] Claudia Leopold und Michael Süß. *Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments*. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. 2006. DOI: 10.1007/11846802_41.

[103] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt und Hans-Joachim Bungartz. *Infrastructure and API Extensions for Elastic Execution of MPI Applications*. In: *Proceedings of the of the 23rd European MPI Users' Group Meeting*. 2016. DOI: 10.1145/2966884.2966917.

[104] Iván Cores Gonzáles. *Fault-Tolerance and Malleability in Parallel Message-Passing Applications*. Diss. Universidade da Coruña, 2015. URL: https://ruc.udc.es/dspace/bitstream/handle/2183/16073/CoresGonzalez_Ivan_TD_2015.pdf (besucht am 10. 09. 2018).

[105] Iván Cores Gonzáles, Patricia González, Emmanuel Jeannot, María J. Martín und Gabriel Rodríguez. *An Application-Level Solution for the Dynamic Reconfiguration of MPI Applications*. In: *Proceedings of the High Performance Computing for Computational Science*. 2017. DOI: 10.1007/978-3-319-61982-8_18.

[106] Gonzalo Martin Cruz. *Optimization Techniques for Adaptability in MPI Application*. Diss. Universidad Carlos III de Madrid, 2015. URL: https://orff.uc3m.es/handle/10016/22631#preview (besucht am 10. 09. 2018).

 doi:10.17170/kobra-2018122577

[107] Sebastian Buchwald, Manuel Mohr und Andreas Zwinkau. *Malleable Invasive Applications*. In: *Proceedings der 8. Arbeitstagung Programmiersprachen*. 2015. URL: https://pdfs.semanticscholar.org/7057/f8a6549bc8a8200af192fbcee34d9fb250d0.pdf (besucht am 10. 09. 2018).

[108] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky und Olivier Tardieu. *The Asynchronous Partitioned Global Address Space Model*. In: *Proceedings of the ACM SIGPLAN Workshop on Advances in Message Passing*. 2010. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.464.557&rep=rep1&type=pdf (besucht am 10. 09. 2018).

[109] Andreas Prell. *Embracing Explicit Communication in Work-Stealing Runtime Systems*. Diss. Universität Bayreuth, 2016. URL: https://epub.uni-bayreuth.de/2990/1/main_final.pdf (besucht am 10. 09. 2018).

[110] *Universität Kassel – Wissenschaftliche Datenverarbeitung*. URL: https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html (besucht am 10. 09. 2018).

[111] *Wikichip – Intel® Xeon® Processor E5-2643 v4*. URL: https://en.wikichip.org/wiki/intel/xeon_e5/e5-2643_v4 (besucht am 10. 09. 2018).

[112] *Intel® Xeon® Processor E5-2643 v4 Produktspezifikation*. URL: https://ark.intel.com/products/92989/Intel-Xeon-Processor-E5-2643-v4-20M-Cache-3_40-GHz (besucht am 10. 09. 2018).

[113] Evgeni J. Gik. *Schach und Mathematik*. 1. Aufl. Thun, 1987. ISBN: 3871449873.

[114] *HabaneroUPC++ git Repository*. URL: https://github.com/habanero-rice/habanero-upc (besucht am 10. 09. 2018).

[115] *Spin Homepage*. URL: http://spinroot.com (besucht am 10. 09. 2018).