Das C++ Actor Framework im Leistungsvergleich

Marian Triebe, Dominik Charousset, Raphael Hiesgen, Thomas C. Schmidt
Internet Technologies Group, Dept. Informatik, HAW Hamburg, Germany
{marian.triebe, dominik.charousset, raphael.hiesgen, t.schmidt}@haw-hamburg.de

***Zusammenfassung*—Der langjährige Trend in der CPU-Entwicklung vermehrt Kerne und nicht die Leistung pro Kern. Die dabei entstehenden Multicore-Architekturen sind nur mittels nebenläufiger Programmierung gleichmäßig auslastbar. Gleichzeitig lassen die Omnipräsenz des Internets zwischen heterogenen, weltweit verteilten Komponenten, insbesondere aber spezialisierte Internet-Lösungen wie Clouds, das Internet der Dinge (IoT) etc. verteilte Anwendungen immer stärker in den Vordergrund treten. Ein Lösungskonzept für die transparente, robuste Programmierung paralleler und verteilter Systeme ist das Aktorenmodell. Das C++ Actor Framework (CAF) ist unser Beitrag zur Realisierung dieses Konzepts. CAF kann sowohl hochparallele Maschinen, als auch heterogene Hardware-Komponenten wie GPUs, aber auch lose gekoppelte Systeme im Internet und stark beschränkte IoT-Knoten in einem Programmsystem verknüpfen.**

In diesem Beitrag stellen wir die Leistungsmerkmale von CAF vor. Wir vergleichen CAF zunächst mit dem elementaren Messaging Ansatz von MPI, wobei vor allem die Skalierbarkeit von Interesse ist. Wir stellen weiterhin CAF anderen Implementierungen des Aktorenmodells gegenüber und vergleichen Speicherverbrauch und Terminierungszeit. Unsere Messungen zeigen, dass CAF in den gemessenen Kategorien durch hochskalierbares Design und effiziente Implementierung in C++ überlegene Leistungswerte aufweist. Außerdem präsentieren wir Messungen für die Bereiche der eingebetteten und mobilen Systeme, wo effiziente RAM Nutzung eine wichtige Rolle spielt.

***Index Terms*—Actor Model, C++, Message-oriented Middleware, Performance Measurement**

I. Einleitung

Die Software-Industrie sieht sich derzeit gleich mehreren Trends gegenüber. Der größte Umbruch war das Ende des stetigen Anstiegs der Taktfrequenzen um die Jahrtausendwende und der Beginn der "Multicore-Ära". Die Notwendigkeit der nebenläufigen Ausführung, um von neuen Hardware-Generationen profitieren zu können, erforderte ein Umdenken bei der Software-Entwicklung. Gleichzeitig verbindet die Cloud mehr und mehr Bereiche des täglichen Lebens, was auch bedeutet, dass immer mehr Computer-Programme kommunizieren und in einem globalen Netz organisiert sind. Der dritte Trend ist die Automatisierung der Maschinen-Kommunikation und die Schaffung des IoT. Hierbei lösen viele Kleinstrechner im Verbund Probleme, was ein hohes Maß an Organisation und Verlässlichkeit erfordert. Heterogene Ablaufumgebungen bilden den vierten Trend. Technologien wie OpenCL [1] erlauben das Ausführen beliebiger Programme auf der Grafikkarte, wodurch Lösungen für stark parallelisierbare Probleme, wie beispielsweise Kryptographie oder Video(de)kodierung, enorme Geschwindigkeitssteigerung erreichen können.

Das Aktorenmodell [2] beschreibt Computer-Programme als Summe einzelner Software-Komponenten, die durch den Austausch von Nachrichten kommunizieren. Da Aktoren voneinander unabhängig agieren, kann eine intelligente Laufzeitumgebung nebenläufige Aktoren beliebig auf einem Rechner und auch über mehrere Rechner hinweg parallel ausführen. Dabei ist der logische Programmaufbau unabhängig von der realen Verteilung. Darüber hinaus bietet das Aktorenmodell ein starkes Fehlermodell auf Basis von uni- und bidirektionaler Überwachung mit nachrichtenbasierter Fehlersignalisierung. Beim Ausfall einer oder mehrerer Aktoren lassen sich Teile einer Anwendung im laufenden Betrieb neu verteilen, wodurch eine Anwendung tolerant gegenüber ausfallenden Systemkomponenten wird.

Konzeptionell eignet sich das Aktorenmodell zur Entwicklung nebenläufiger und verteilter Systeme gleichermaßen. Auf Mehrkernsystemen skalieren Aktorensysteme, indem ein Problem in viele Teilprobleme zerlegt wird und entsprechend viele Aktoren gestartet werden. Dadurch entstehen im Vergleich zur Anzahl physisch vorhandener CPU-Kerne viele Aktoren, die jedoch einzeln eine geringe Komplexität haben. Gemäß dem Amdahlschen Gesetz [3] minimiert dieser Ansatz die sequentielle Komponente eines Programms und ermöglicht effizientes Skalieren mit der Anzahl vorhandener Kerne.

Mit dem C++ Actor Framework (CAF) [4] stellen wir unser Konzept zur transparenten und elastischen Programmierung von Aktoren in C++ vor und vermessen seine Leistungsmerkmale. Hierbei analysieren wir einerseits die Zusatzkosten des CAF-Laufzeitsystems, andererseits vergleichen wir die Performance in markanten Benchmarks mit den bekanntesten Aktor-Laufzeitsystemen sowie mit MPI. Wir konzentrieren uns insbesondere auf Aspekte der Skalierbarkeit und können für CAF ein hervorragendes Leistungsverhalten zeigen.

Die Architektur von CAF sowie die Hauptkomponenten für eine effiziente Laufzeit werden in Abschnitt II vorgestellt. In Abschnitt III vergleichen wir die Leistungsmerkmale von CAF mit anderen Aktor-Laufzeitsystemen. Eine Leistungsmessung für verteilte und heterogene Systemen findet sich in Abschnitt IV. Abschließend möchten wir unsere Ergebnisse in Abschnitt V diskutieren.

II. Das C++ Actor Framework und verwandte Arbeiten

Mit stagnierenden Taktraten und dem absehbaren Ende des Mooreschen Gesetzes [5] stieg auch das Interesse innerhalb der Entwicklergemeinde an nativen Programmiersprachen, um auf ein größeres Optimierungspotential zurückgreifen zu

können. Insbesondere C++ erlebte durch diese Entwicklung eine Renaissance, was sich unter anderem im Erscheinen von zwei neuen Standard-Versionen (2011 und 2014) widerspiegelt. Jedoch bleiben die verfügbaren Programmierwerkzeuge zur Entwicklung nebenläufiger Anwendungen vergleichsweise niedrigstehend und beschränken sich auf Synchronisationsprimitive wie Mutexe und Condition-Variablen, welche zur korrekten Benutzung viel Expertenwissen erfordern [6]. Außerdem kann es bei naivem Speicherlayout zu False Sharing kommen, was zu langsamerer Laufzeit trotz einem Ansatz mit geringerer Abstraktion führt [7]. Ähnlich niedrigstehende Primitive stellt C++ zur verteilten Programmierung bereit (Programmierung auf Socket-Ebene). Dies ist fehleranfällig, komplex und aufwendig.

Das Aktorenmodell adressiert sowohl Nebenläufigkeit als auch Verteilung auf einer einheitlichen Abstraktionsschicht. Mit CAF möchten wir auf Basis des Aktorenmodells eine hochstehende Programmierschnittstelle für Entwickler anbieten, die Nebenläufigkeit sowie Verteilung in einem robusten Programmierparadigma zusammenfasst. Ziel ist dabei, das Aktorenmodell für den Einsatz in Performance-kritischer Infrastruktur-Software, im Embedded-Bereich und für das IoT anzupassen und eine Laufzeitumgebung anzubieten, die Ressourcen effizient verwaltet und ein effizientes Scheduling vornimmt. Mit Ausrichtung auf diese Anwendungsfelder ergeben sich Anforderungen an Scheduling sowie Ressourcen-Management, die beim Entwurf bisheriger Implementierungen nur eine untergeordnete Rolle hatten. Traditionelle Implementierungen des Aktorenmodells basierten meist auf virtualisierten Umgebungen wie der JVM [8] und verfolgten entsprechend andere Ziele. Eine Implementierung für C++, die sich zumindest am Aktorenmodell orientiert, ist Charm++ [9], das konzeptionell jedoch auf MPI aufsetzt und seinen Fokus auf die Entwicklung von Anwendungen für Cluster und Supercomputer legt.

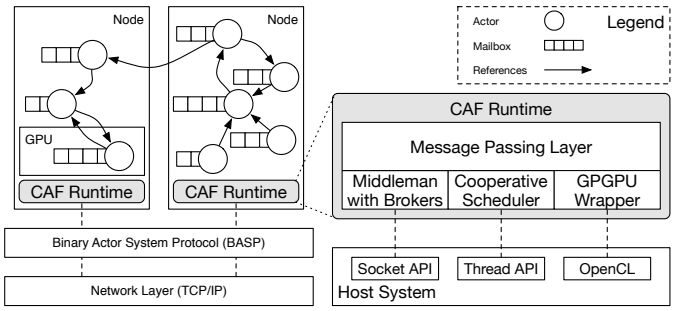


Abbildung 1. CAF-Architektur

Aktoren in CAF laufen in einer Laufzeitumgebung, die Nachrichtenversand, lokales Scheduling sowie Queue-Management zur Verfügung stellt.

Die Kommunikation mit entfernten Aktoren wird durch diese Laufzeitumgebung transparent bereitgestellt. Logische Verbindungen zwischen Aktoren werden durch einen ”Middleman” hergestellt, der als Multiplexer dient. Auf dem Hostsystem wird daher nur ein offener Port benötigt. Abbildung 1 zeigt die Architektur einer verteilten CAF-Anwendung. Aktoren sind sich der physischen Verteilung nicht bewusst, sondern formen ein logisches Kommunikationsnetzwerk, welches von der Laufzeitumgebung prinzipiell beliebig auf physische Knoten verteilt werden kann. Diese flexible Topologie wird ermöglicht durch das ”Binary Actor System Protocol” (BASP). Verteilte Laufzeitumgebungen realisieren ihre Kommunikation über Middleman-Instanzen. Die Hauptaufgabe eines Middlemans ist es, von der Socket-API des Hostsystems zu abstrahieren und eine nachrichtenbasierte Schnittstelle für Netzwerkkommunikation bereitzustellen. Paket- und Byte-Streams auf der Netzwerkschicht werden als Nachrichten sogenannten Brokern zugestellt. Broker sind Aktoren, die asynchrones I/O ausführen und in der Event-Schleife des Middlemans laufen. Entfernte Aktoren werden durch den ”BASP Broker” kontaktiert, welcher interne Aktor-Nachrichten an das Netzwerk weiterleitet. Ein kooperativer Scheduler organisiert paralleles sowie faires Ausführen von Aktoren auf einem lokalen Knoten und verwendet die Threading-API aus der C++ Standard-Bibliothek. Der GPGPU-Wrapper verbirgt heterogene Hardware-Komponenten hinter einer Fassade. Er erstellt Aktoren, die Aufgaben an OpenCL-Kernel weiterleiten. Nachrichten werden zur Laufzeit in OpenCL-kompatible Datentypen gewandelt und umgekehrt.

In den folgenden Abschnitten möchten wir zunächst zwei Algorithmen vorstellen, die für einen effizienten Nachrichtenaustausch essentiell sind. Nachfolgend wird zudem der Scheduler im Detail vorgestellt, da dieser maßgeblich für die Leistungsmerkmale von CAF auf Mehrkernsystemen verantwortlich ist.

A. Mailbox

Die Mailbox-Implementierung ist eine der wichtigsten Komponenten in einem Aktor-Laufzeitsystem. Empfängt ein Aktor von mehreren Sendern gleichzeitig Nachrichten, so kann die Mailbox zu einem Flaschenhals werden und somit die gesamte Systemperformance negativ beeinflussen. Eine hohe Performance bei nebenläufiger Ausführung ist nur möglich, wenn parallele Lese- und Schreibzugriffe keinen signifikanten Synchronisations-Aufwand verursachen. Die Wahl des richtigen Algorithmus ist daher entscheidend. Die Mailbox in CAF ist eine “Single-Reader-Many-Writer-Queue”. Diese Implementierung erlaubt paralleles Schreiben in die Mailbox, wobei nur der Besitzer der Queue Nachrichten aus dieser Mailbox entnehmen kann. Unsere Implementierung basiert auf einem nicht-blockierenden Stack (LIFO), für den alle Zugriffe mit Hilfe einer einzigen atomaren compare-and-swap (CAS)-Operation realisiert sind. Um die Nachrichten in der korrekten Reihenfolge auszulesen ist dieser Stack mit einem internen Cache kombiniert, der Nachrichten in FIFO-Reihenfolge übersetzt und nur dem Besitzer zugänglich ist. Die Implementierung der Mailbox leidet nicht unter dem ABA-Problem bei parallelem Zugriff in CAS-basierten Systemen [10], da durch die Entkopplung über den internen Cache jedes Element maximal einem Thread zur Zeit zugänglich ist. Die Komplexität der Mailbox liegt bei $O(1)$ für Einfüge-

Operationen. Bei der Entnahme liegt der Durchschnitt bei $O(1)$, jedoch im schlechtesten Fall durch das umsortieren der Nachrichten in den internen Cache bei $O(n)$, wobei n die Anzahl der Nachrichten ist. Eine detailliertere Veranschaulichung der Mailbox wurde in vorherigen Publikationen gezeigt [11].

B. Copy-On-Write Nachrichten

Die Implementierung der Nachrichtenschicht von CAF verwendet Tupel mit Wertesemantik. Jedoch kann ein Tupel zwischen mehreren Aktoren implizit geteilt sein, so lange keiner der Aktoren schreibend auf das Tupel zugreift. Sobald ein Aktor Schreibzugriff anfordert, wird eine Kopie des Tupel erstellt. Unsere Implementierung verwendet atomare Referenzzähler, die Race-Conditions ausschließt. Die dadurch entstehenden Laufzeitkosten sind minimal und dadurch zu vernachlässigen.

C. Scheduler

CAF hat zum Ziel, Millionen von Aktoren auf hunderten von Prozessorkernen effizient auszuführen. Das naive Verfahren jedem Aktor einen dedizierten Thread zuzuweisen würde zu hohem Aufwand bei geringer Skalierbarkeit führen, da jeder Thread seinen eigenen Stack allokiert und Ressourcen im Betriebssystemkern belegt. Um eine bessere Ausnutzung von Speicher und CPU-Zeit zu erreichen nutzt CAF eine beim Starten festgelegte Anzahl an Worker-Threads, über die dynamisch Aktoren verteilt werden. Die Anzahl der Worker wird zur Laufzeit bestimmt und richtet sich nach der lokalen Hardware-Nebenläufigkeit, wobei ein CPU-Thread einem Worker entspricht. Da das Scheduling der Aktoren oberhalb des Betriebssystems stattfindet, kann der Scheduler von CAF Aktoren nicht unterbrechen. Ein kooperativer Scheduler kann keine Fairness garantieren. Aktoren, die sich nicht zu einer kooperativen Ausführung eignen, beispielsweise weil sie auf blockierende I/O Funktionen zurückgreifen, können durch den Programmierer in separate Threads ausgelagert werden.

Um die Skalierbarkeit auch auf Mehrkern-CPUs sicherzustellen ist es wichtig, die Worker möglichst gleichmäßig auszulasten. Der Work-Stealing-Ansatz [12] ist ein grundlegendes Verfahren für Scheduling oberhalb des Betriebssystems für Fälle, in denen die Laufzeit keine Vorhersagen über das Verhalten der Anwendung treffen kann. Work-Stealing ist beispielsweise in Cilk [13], Intel Thread Building Blocks [14], sowie in Javas Fork/Join [15] implementiert. Jeder Worker hat seine eigene Job-Queue, die er abarbeitet solange Einträge in der Queue existieren. Sobald ein Worker keine Einträge mehr in seiner Job-Queue hat, nimmt er ("stiehlt") Einträge aus Job-Queues anderer Worker. Die Auswahl, bei welchem Worker gestohlen wird, geschieht in der Regel per Zufall.

III. Leistungsmessung in nebenläufigen Systemen

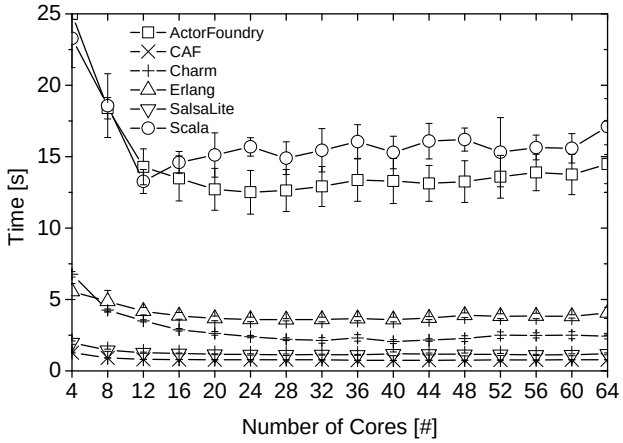
In diesem Abschnitt präsentieren und analysieren wir Messungen zu Laufzeit, Speicherverbrauch und Skalierbarkeit von CAF auf Mehrkernsystemen. Speicherverbrauch und Laufzeit wurden im Vergleich zu ActorFoundry, Erlang, SalsaLite, Scala (Akka) und Charm++ gemessen. Folgende Versionen wurden im Detail verwendet: (1) C++ mit CAF 0.13 (CAF) sowie Charm++ 6.5.1 (Charm), (2) Java mit dem auf Kilim [16] basierenden ActorFroundy 1.0 (ActorFoundry), (3) Erlang 5.10.2 mit HiPE für native Code Erzeugung und Optimierungslevel O3 (Erlang), (4) SALSA Lite 0.0.3 und (5) Scala 2.10.3 mit Akka. CAF und Charm++ wurden als Release-Version mit Clang 3.5.2 kompiliert. Scala, SALSA Lite und ActorFoundry liefen auf einer JVM mit maximal 10 GB RAM. Für die Kompilierung von ActorFoundry wurde der Java-Compiler in Version 1.6.0-38 verwendet, da aktuellere Compiler nicht unterstützt werden. Alle Graphen zeigen die Laufzeit als Mittelwert aus zehn unabhängigen Läufen. Der Speicherverbrauch wird als Boxplot dargestellt, um den stark fluktuierenden Speicherverbrauch über die Gesamtzeit der Ausführung zusammenzufassen. Das Hostsystem besitzt vier Opteron CPUs mit je 16 Kernen (insgesamt 64 CPU-Kerne). Um die Last stets gleichmäßig über alle CPUs zu verteilten wurde pro Durchlauf die Anzahl aktivierter CPU-Kerne um vier erhöht, beginnend bei 4. Der Programmcode zu allen Benchmarks ist online verfügbar unter `https://github.com/actor-framework`.

A. Erzeugung von Aktoren

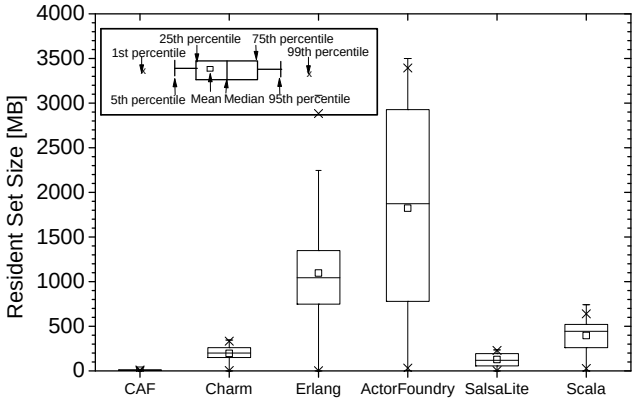
In unserem ersten Benchmark soll erfasst werden, wie effizient das Erzeugen vieler Aktoren in einem klassischen Divide-and-Conquer-Arbeitsablauf ist. Primär misst dieser Versuchsaufbau folglich die Kosten zum Starten eines Aktors. Dazu wurden rekursiv 2^{20} (etwa eine Million) Aktoren erstellt, die eine Baumstruktur bilden.

Die Laufzeitmessungen in Abbildung 2(a) zeigen, dass CAF die niedrigste Laufzeit benötigt. Nachdem ein globales Minimum erreicht wurde, bleibt die Laufzeit konstant. Salsa skaliert in diesem Testfall ähnlich wie CAF, hat jedoch eine höhere Laufzeit. Bei Charm nimmt die Laufzeit von vier bis 32 CPU-Kernen ab, stagniert dann, um bei 36 CPU-Kernen einen leichten Anstieg zu verzeichnen. Danach verschlechtert sich die Laufzeit von Charm. Erlang weist ähnliches Verhalten auf. Bei der Erhöhung der Anzahl der CPU-Kerne von vier auf 24 verbessert sich die Laufzeit, um dann zu stagnieren bzw. leicht anzusteigen. Scala skaliert von vier auf zwölf CPU-Kerne sehr gut, zeigt dann aber überraschenderweise eine signifikante Verschlechterung der Laufzeit. ActorFoundry erreicht sein Optimum bei 24 CPU-Kernen und hat bis dahin kontinuierlich seine Laufzeit verbessert. Ab 28 CPU-Kernen verschlechtert sich die Laufzeit von ActorFoundry jedoch wieder. Das erwartete Verhalten für alle Implementierungen in diesem Test ist ein konstanter Abfall in der Laufzeit, bis ein globales Minimum erreicht wird. Dieses Verhalten ist jedoch nicht bei allen Implementierungen zu beobachten. In absoluten Werten schneiden CAF und Salsa am besten ab.

Die Speichermessungen in Abbildung 2(b) zeigen, dass CAF mit ca. 10 MB den bei weitem niedrigsten Speicherverbrauch aufweist, gefolgt von Salsa und Charm mit jeweils ca. 250 MB, Scala mit 500 MB, Erlang mit gut einem GB Speicherverbrauch und ActorFoundry mit knapp zwei GB. Im Zusammenhang mit den vorherigen Laufzeitmessungen

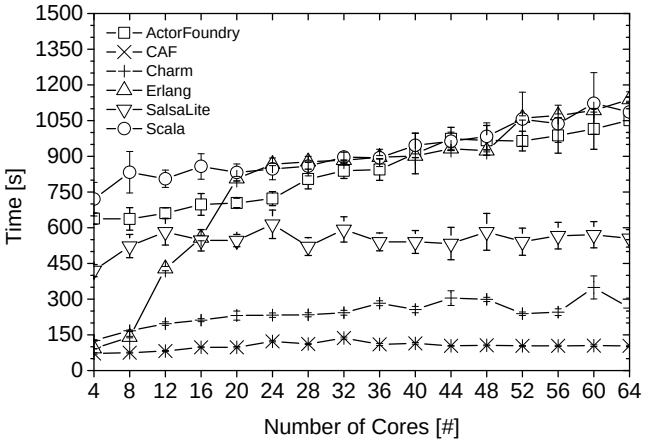


(a) Laufzeit

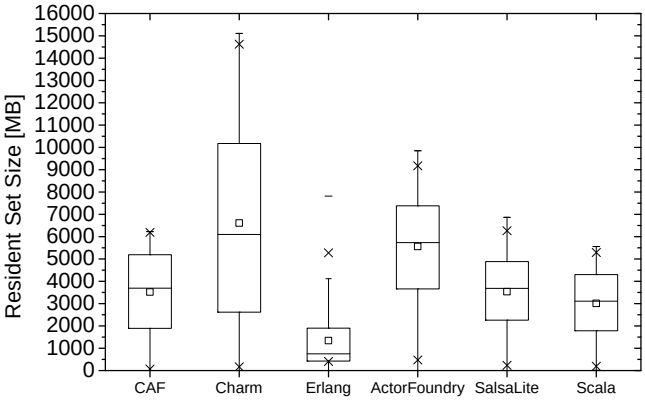


(b) Speicherverbrauch

Abbildung 2. Aktor spawn performance für 2^{20} Aktoren

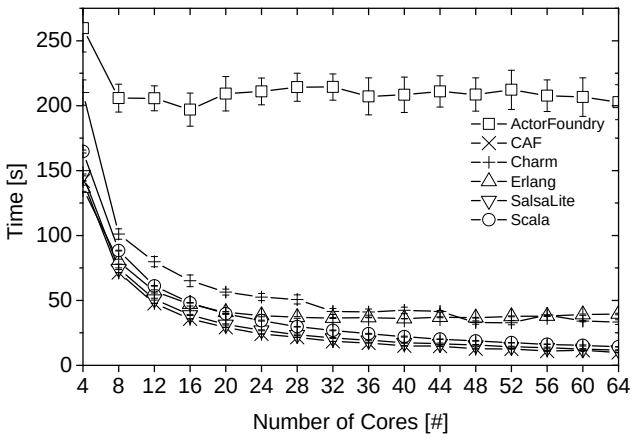


(a) Laufzeit

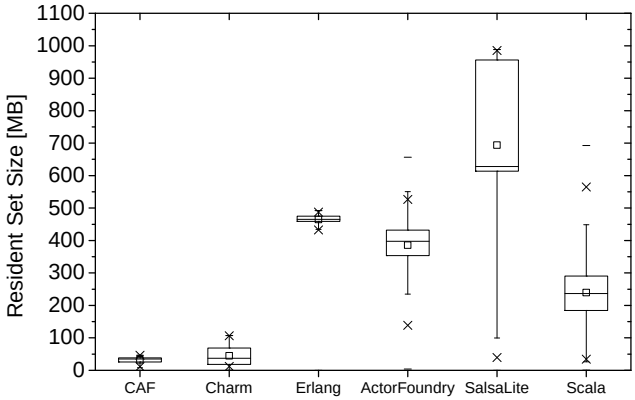


(b) Speicherverbrauch

Abbildung 3. Mailbox performance im N:1 Kommunikations Szenario



(a) Laufzeit



(b) Speicherverbrauch

Abbildung 4. Performance in einem gemischten Szenario mit zusätzlichem Work-Load

lässt sich vermuten, dass bei den langsameren Implementierungen ineffizientere Speicherverwaltung ausschlaggebend sein könnte. Dies würde auch die langsamere Laufzeit mit steigender Nebenläufigkeit erklären.

B. Mailbox unter Last

Mit unserem zweiten Testaufbau möchten wir das Skalierungsverhalten der Mailbox untersuchen. Dazu betrachten wir ein Szenario, bei dem ein zentraler Aktor Nachrichten von mehreren Sendern parallel empfängt, wodurch die Mailbox zum Synchronisationspunkt wird. Hierfür senden 100 Aktoren jeweils eine Million Nachrichten an einen einzigen Empfänger.

Das erwartete Verhalten für alle Implementierungen in diesem Test ist eine konstante oder leicht steigende Laufzeit, da mehr Kerne nicht das abarbeiten der 10^8 Nachrichten beschleunigen, sondern zu mehr Synchronisationsaufwand in der Mailbox führen. Abbildung 3(a) zeigt, dass CAF die niedrigste Laufzeit hat, welche beim Zuschalten mehrerer CPU-Kerne nur marginal ansteigt. Charm ist für wenig Nebenläufigkeit noch etwa gleichauf mit CAF, weist jedoch eine konstante Steigung und starke Fluktuation auf. Auch Erlang bietet auf vier CPU-Kernen eine gute Laufzeit, die in etwa der von CAF entspricht. Jedoch steigt die Laufzeit von Erlang ab zwölf CPU-Kernen unverhältnismäßig. Die Laufzeit von Salsa bleibt relativ konstant, benötigt jedoch im Minimum die dreieinhalbfache Laufzeit der schlechtesten Messung für CAF. ActorFoundry sowie Scala zeigen einen stetigen Anstieg der Laufzeit beim Hinzufügen von CPU-Kernen. Auch wenn Erlang für 4 und 8 Kerne noch zur Spitzengruppe gehört, so fällt doch auf, dass alle virtualisierten Lösungen signifikant langsamer als die beiden nativen Implementierungen sind. Die Messwerte des Speicherverbrauchs in Abbildung 3(b) zeigen, dass Erlang den geringsten Speicherverbrauch hat und maximal 8 GB an Arbeitsspeicher verwendet. CAF liegt im Schnitt bei 3,5 GB Speicherverbrauch, wobei maximal 6,5 GB verbraucht werden. Es folgt Scala mit ca. 3,5 GB Speicherverbrauch und maximal 6 GB, sowie Salsa mit 4 GB Speicherverbrauch und maximal 7 GB. ActorFoundry verbraucht im Mittel 6 GB und maximal 10 GB. Charm schlägt im Mittel mit 7 GB zu Buche, hat jedoch einen Maximalverbrauch von ca. 15,5 GB.

C. Gemischte Anwendung

In diesem Benchmark soll eine realistische Auslastung simuliert werden. Er besteht sowohl aus dem Austausch von Nachrichten sowie einer Primfaktor-Zerlegung. Aktoren wurden dabei in Ringen organisiert. Jeden Ring durchläuft eine Nachricht mit einem Zähler, der pro Durchlauf dekrementiert wird. Sobald der Zähler null erreicht, wird der Ring aufgelöst. Pro Ring gibt es einen Worker, der eine Primfaktor-Zerlegung (28.350.160.440.309.881 d.h. 329.545.133 und 86.028.157) vornimmt. Diese Zerlegung wird gestartet sobald ein Ring instanziiert wurde. Im Detail wurden 100 Ringe mit jeweils 100 Aktoren gestartet, der initiale Wert der Nachricht ist 1000, wobei jeder Ring insgesamt vier Mal neu instanziiert wurde. Das ideale Laufzeitverhalten für diesen Testaufbau ist ein konstantes Sinken der Laufzeit, bis ein globales Minimum erreicht ist.

Abbildung 4(a) zeigt, dass alle Implementierungen mit Ausnahme von ActorFoundry zum Ideal tendieren. Scala, Charm sowie ActorFoundry benötigen die längste Laufzeit. Erlang erreicht seine minimale Laufzeit auf 28 Kernen und bleibt danach stabil. Charm zeigt von 52 auf 56 CPU-Kernen einen leichten Anstieg, fällt dann bei 56 auf 60 aber wieder auf sein altes Niveau zurück. CAF hat in diesem Test erneut die kürzeste Laufzeit.

Die Messwerte zum Speicherbrauch in Abbildung 4(b) zeigen, dass CAF mit ca. 50 MB den geringsten Speicherverbrauch und die geringste Fluktuation aufweist. Es folgt Charm mit ca. 60 MB Speicherverbrauch und einem Maximum von knapp 130 MB. Scala folgt mit gut 250 MB und einem Maximum von 700 MB. ActorFoundry benötigt im Durchschnitt 400 MB Speicher und 700 MB als Maximum. Erlang zeigt kaum Fluktuation und verbraucht ca. 500 MB Speicher. Salsa bildet das Schlusslicht mit 700 MB sowie einem GB Maximalverbrauch.

IV. Leistungsmessung in verteilten und heterogenen Systemen

Nach den Messungen in nebenläufigen Szenarien möchten wir im Folgenden die Skalierbarkeit des Nachrichtenaustausches im Netzwerk sowie das Verhalten der Laufzeitumgebung beim parallelen Einsatz einer GPU messen.

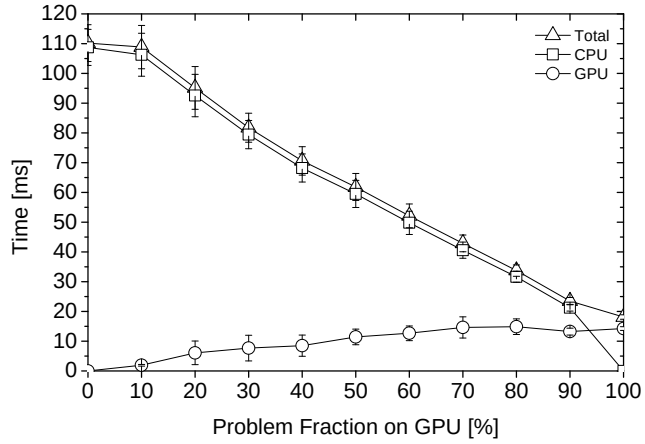
A. Kommunikations-Overhead: CAF vs. MPI

Für diesen Testaufbau haben wir virtuelle Maschinen (VMs) auf einem System mit 64 Kernen mit jeweils 2299 MHz ausgeführt. Zur Virtualisierung kam QEMU [17] zum Einsatz. Jede VM verfügt über einen CPU-Kern und 2GB Arbeitsspeicher. Die Kommunikation zwischen den VMs findet in einem lokalen VLAN statt, welches per Open vSwitch [18] eingerichtet ist. Zu jedem Testlauf gehören 4-64 Worker-VMs (in Viererschritten) sowie eine Master-VM, welche die Aufgaben auf die vorhandenen Worker verteilt. In diesem Test wurde nur die reine Rechenzeit gemessen, da sich die Initialisierungsphase zwischen CAF und OpenMPI stark unterscheidet.

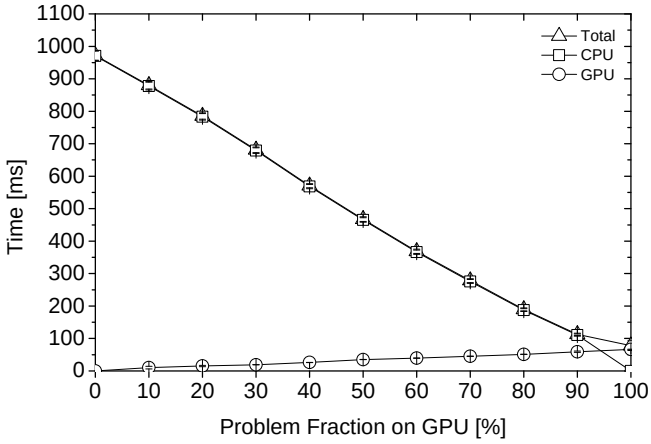
Abbildung 5 zeigt die Laufzeit in Sekunden als Funktion der verwendeten Worker-VMs. Zu beobachten ist ein stetiger Abfall der Laufzeit. Zu erwarten wäre eine Halbierung der Laufzeit bei gleichzeitiger Verdoppelung der Worker. Dieses Optimum wird von CAF und OpenMPI in den ersten beiden Skalierungsstufen (4 auf 8 sowie 16 auf 32) erreicht. Die Messungen zeigen, dass CAF trotz seiner höheren Abstraktion sogar ein leicht besseres Skalierungsverhalten als OpenMPI aufweist.

B. Hybride Berechnungen mit OpenCL-basierten Aktoren

In unserer letzten Leistungsmessung möchten wir zeigen, wie effizient sich eine GPU mit CAF in eine Aktor-basierte Anwendung integrieren lässt. Wir haben dazu die Mandelbrot-Menge als ein leicht zu parallelisierendes Problem gewählt und dieses über eine 12-Kern CPU und eine GPU (Nvidia



(a) Großes Problem: Mandelbrot mit max. 1200 Iterationen



(b) Kleines Problem: Mandelbrot mit max. 120 Iterationen

Abbildung 6. Workload von CPU auf GPU

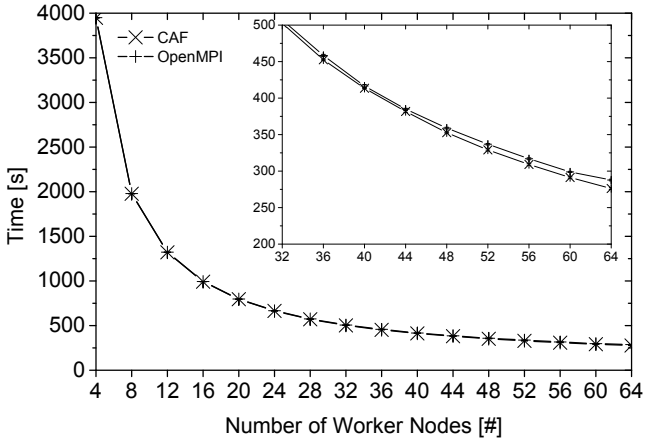


Abbildung 5. CAF vs MPI Mandelbrot

Tesla C2075) aufgeteilt, wobei wir in jedem Schritt 10% des Problems von der CPU auf die GPU verlagern. Das zu generierende Bild hat eine Auflösung von 1920x1080 Pixel und liegt im Bereich [-0.5 - 0.7375i - 0.1375i] des Mandelbrots. Beide Graphen in Abbildung 6 beschreiben die Laufzeit als Funktion über den Anteil an Berechnungen auf der GPU. Zusätzlich ist ein Fehlerbalken eingezeichnet, der die Standardabweichung anzeigt. Die Laufzeitwerte liegen in Millisekunden vor. Die Problemgröße in Abbildung 6(a) ist zehnmal größer als in Abbildung 6(b). Zusätzlich zur Gesamtlaufzeit werden die Zeiten für CPU und GPU (die Zeit zwischen Start und Terminierung) dargestellt, wobei die Gesamtlaufzeit nicht die Summe der CPU- und GPU-Laufzeit ist, da diese nebenläufig sind.

Allgemein ist die CPU-Laufzeit leicht niedriger als die Gesamtlaufzeit, wenn die CPU für Berechnungen verwendet wird. Im Vergleich benötigt die GPU nur einen Bruchteil der Laufzeit der CPU. Für beide Problemgrößen ist ein linearer Abfall der Laufzeit zu beobachten, wenn Teile auf die GPU ausgelagert werden. Die Fehlerbalken sind im Vergleich zur Laufzeit klein. In Abbildung 6(b) zeigt sich, dass durch die insgesamt kleinere Problemgröße die Auslagerungskosten an die GPU mehr ins Gewicht fallen. Zwar verringert sich auch hier die Laufzeit, jedoch weniger signifikant als in Abbildung 6(a) bei der größeren Problemgröße zu sehen ist. Ein Grund hierfür ist, dass sich CPU und GPU keinen Speicher teilen und der Großteil der Laufzeit auf das Kopieren der Daten entfällt.

V. Zusammenfassung und Ausblick

Die zunehmende Parallelisierung und Heterogenisierung der Hardware sowie der verstärkte Einsatz von Cloud-Technologien führen dazu, dass Entwickler für diese Einsatzgebiete hochskalierbare, robuste Lösungen benötigen. Das Aktorenmodell bietet ein integratives Konzept an, diese Teilaspekte im Verbund anzugehen, und es Anwendungen zu ermöglichen, nahtlos von Mehrkern- zu Cluster-Systemen überzugehen. Insbesondere im C++-Umfeld fehlt bisher jedoch ein vollwertiges Programmier-Framework, das die theoretischen Versprechen der Aktoren effizient in die Realität umsetzt.

Das hier vorgestellte C++ Actor Framework – CAF – basiert auf dem Aktorenmodell und richtet sich insbesondere an Infrastruktur-Software mit hohen Anforderungen an Skalierbarkeit und Robustheit. Darüber hinaus ist das Framework auch für Klein- und Kleinstgeräte, wie sie im IoT anzutreffen sind, geeignet. In den hier vorgestellten Leistungsmessungen zeigt sich, dass CAF nebenläufige Hardware sehr gut ausnutzt und insbesondere der Speicherverbrauch im direkten Vergleich zu konkurrierenden Ansätzen sehr gering ist.

Derzeit arbeiten wir an einer Portierung von CAF auf das IoT-Betriebssystem RIOT [19] und einer weiteren Reduktion des Speicherbedarfs. Darüber hinaus arbeiten wir an einer Ausweitung des Schedulings auf verteilte Systeme, um in massiv parallelen Installationen automatische Lastverteilung durch intelligente Migration von Aktoren zur Laufzeit zu gewährleisten.

LITERATUR

[1] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[2] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd IJCAI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[3] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[4] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications," in *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2014.

[5] L. B. Kish, "End of Moore's law: Thermal (Noise) Death of Integration in Micro and Nano Electronics," *Physics Letters A*, vol. 305, no. 3–4, pp. 144–149, 2002.

[6] S. Meyers and A. Alexandrescu, "C++ and the Perils of Double-Checked Locking," *Dr. Dobb's Journal*, July 2004.

[7] J. Torrellas, H. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Comput.*, vol. 43, no. 6, pp. 651–663, Jun. 1994.

[8] R. K. Karmani, A. Shali, and G. Agha, "Actor Frameworks for the JVM Platform: A Comparative Analysis," in *PPPJ*, 2009, pp. 11–20.

[9] L. V. Kale and S. Krishnan, "Charm++: Parallel programming with message-driven objects," *Parallel Programming using C++*, pp. 175–213, 1996.

[10] I.B.M. Corporation, "IBM System/370 Extended Architecture, Principles of Operation," IBM, Tech. Rep. SA22-7085, 1983.

[11] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013.

[12] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995.

[14] Pheatt, Chuck, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[15] Lea, Doug, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 2000, pp. 36–43.

[16] S. Srinivasan and A. Mycroft, "Kilim: Isolation-Typed Actors for Java," in *Proceedings of the 22nd ECOOP*, ser. LNCS, vol. 5142. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 104–128.

[17] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[18] Pfaff, Ben and Pettit, Justin and Amidon, Keith and Casado, Martin and Koponen, Teemu and Shenker, Scott, "Extending Networking into the Virtualization Layer." in *Hotnets*, 2009.

[19] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster*. Piscataway, NJ, USA: IEEE Press, 2013.