Rust als sichere Programmiersprache für systemnahe und parallele Software

Wettlaufoperationen

Jens Breitbart, Stefan Lankes

Softwareprobleme wie Race Conditions können parallele und nebenläufige Anwendungen unsicher machen. Rust kommt Entwicklern mit Überprüfungen zur Kompilierzeit bei der Fehlervermeidung entgegen.

Die Entwicklung von Software mit parallelen oder nebenläufigen Aspekten gilt häufig als eine der schwierigsten Aufgaben im Bereich der Softwareentwicklung, da Fehler nur schwer auffindbar sind. Zu den bekannteren Problemen, die im praktischen Einsatz zum Teil tragische Folgen hatten (s. Kasten „Therac-25"), zählen Softwarefehler durch Wettlaufsituation (Race Condition). Diese Fehler können in verschiedenen Programmiersprachen auftreten. Wie der folgende Artikel zeigt, bietet Rust eine Reihe von Überprüfungen zur Kompilierzeit, mit denen sich Wettlaufsituationen verhindern lassen.

Vorteile von Rust

Rust [1][2] ist eine junge, erstmals 2010 vorgestellte, Programmiersprache. Sie entstand aus einem persönlichen Projekt des Mozilla-Mitarbeiters Graydon Hoare. Im Jahr 2015 erschien eine erste stabile Version. Die Weiterentwickelung treibt hauptsächlich Mozilla Research voran. In der Zwischenzeit sind einige Komponenten des Firefox Browsers in Rust programmiert.

Rust verfolgt das Ziel, verschiedene Programmierparadigmen (funktionale, objektorientierte etc.) zu vereinen. Dabei soll sie so effizient wie C/C++ und so sicher wie eine Interpretersprache sein. Der sichere Umgang mit Speicher sowie die Vermeidung von Wettlaufsituationen stellen ein Alleinstellungsmerkmal gegenüber anderen Programmiersprachen dar. So kann Rust eine sichere Speicherverwaltung ohne Garbage Collection garantieren – dies verspricht eine hohe Effizienz.

Wichtig für den Erfolg einer neuen Programmiersprache ist die Integration von existierendem Code. Mit Rust ist es möglich, C und alle kompatiblen Programmiersprachen einzubinden. Das Programm in Listing 1 verdeutlicht, wie sich aus Rust eine C-Funktion aufrufen lässt.

In diesem Beispiel ist unterstellt, dass eine Shared Library existiert, die `libhello.so` heißt und die C-Funktion `c_hello` enthält. Diese Funktion könnte zum Beispiel einen einfachen Text auf der Konsole ausgeben. Innerhalb des Kontrollblocks `extern` ist definiert, wie die Funktion für Rust aussieht. Sie heißt `c_hello` und gibt eine Ganzzahl zurück, `c_int` definiert explizit, dass der verwendete Datentyp in C ein `int` ist. Die Zeile vor dem Kontrollblock extern gibt an, in welcher Shared Library die Funktion zu suchen ist.

Ganzzahlen sind recht simple Datentypen und lassen sich einfach zwischen den Programmiersprachen austauschen. Aber auch komplexere Datenstrukturen, beispielsweise ein Struct, lassen sich zwischen Rust und C austauschen. Dabei ist lediglich zu gewährleisten, dass beide Programmiersprachen die gleiche Darstellungsform verwenden. Im Beispiel in Listing 2 legt die Zeile oberhalb der Struct-Definition fest, dass der Struct das gleiche (Speicher-)Layout wie C verwendet:

Da C aus Sicht von Rust eine unsichere Sprache darstellt, sind C-Funktionsaufrufe nur in unsafe-Blöcken erlaubt. Falls das Programm ein Speicherproblem (zum Beispiel durch Nutzung uninitialisierter Zeiger) besitzt, kann nur in solchen Blöcken das Problem auftreten. In allen anderen Bereichen garantiert der Rust-Compiler, dass dies nicht passiert. Dadurch reduziert sich die Fehlersuche auf wenige Zeilen Rust-Code.

„Therac-25"

Durch eine Wettlaufsituation (Race Condition) in der Software des Linearbeschleunigers Therac-25, der sowohl zum Röntgen als auch für therapeutische Bestrahlung in der Krebstherapie zum Einsatz kam, sind einige Patienten mit dem 100fachen der geplanten Strahlendosis behandelt und dadurch schwer verletzt, einzelne sogar getötet worden. Ursächlich für den Fehler war jedoch nicht die verwendete Programmiersprache.

Ownership/Borrowing

Wie bereits erwähnt, bietet Rust ein sicheres Speichermodell an, das auf Garbage Collection (GC) verzichtet. GC versucht üblicher-

Listing 1: C-Funktion in Rust aufrufen

```
#[link(name = "hello")]
extern {
    fn c_hello() -> c_int;
}

fn main() {
    println!("Hello world from Rust!");

    unsafe {
        c_hello();
    }
}
```

Listing 2: Struct mit C-Layout

```
#[repr(C)]
struct RustObject {
    number: c_int
}
```

Listing 3: Konzept des Ownership

```
std::vector<std::string>* x = nullptr;

{
    std::vector<std::string> z;

    z.push_back("Hello World!");
    x = &z;
}

std::cout << (*x)[0] << std::endl;
```

Listing 4: Rust-Compiler lehnt Code als fehlerhaft ab

```
let x;

{
    let z = vec!("Hello World!");

    x = &z;
}

println!("{}", x[0]);
```

weise zur Laufzeit nicht länger benötigte Speicherbereiche zu identifizieren und freizugeben. Bei Rust wird dies bereits zur Kompilierzeit bestimmt. Voraussetzung dafür ist das Konzept des Ownerships.

Das einfache Beispiel in Listing 3 erläutert das Konzept des Ownerships und macht die Unterschiede zu C/C++ deutlich. Das Beispiel ist zwar konstruiert, zeigt aber dennoch, dass es möglich ist, gültigen C++-Code zu schreiben, den der Compiler anstandslos kompiliert, obwohl er ungültige Speicherzugriffe enthält. Der Vektor `z` wird nach Verlassen des Code-Blocks zwischen den geschweiften Klammern gelöscht, obwohl noch die Referenz `x` auf diesen existiert.

Natürlich lässt sich ein entsprechender Fehler – wie das Beispiel in Listing 4 zeigt – auch in Rust programmieren. Im Unterschied zu einem C/C++-Compiler übersetzt ein Rust-Compiler diesen Code allerdings nicht. Er meldet stattdessen den Fehler, dass `z` nicht lang genug lebt, und akzeptiert das dortige Programm nicht als gültig, obwohl es syntaktisch korrekt ist.

Um diese Schutzmechanismen zu gewährleisten, führt Rust das Konzept des Ownerships (Besitzer) sowie des Borrowings (Ausleihen) ein. In Rust gibt es immer nur einen Besitzer, der das Recht hat, das Objekt freizugeben. Erst durch das Erstellen von Referenzen lässt sich das Objekt verleihen (siehe Listing 5).

Listing 5: Der Besitzer muss das Objekt freigeben

```
let mut x = vec!("Hello World!");

{
    let z = &mut x;
    // Do something with z...
}

println!("{}", x[0]);
```

Durch das Erstellen der Referenz auf `x` steht das Objekt zum Ausleihen bereit. Das Schlüsselwort `mut` beschreibt in diesem Zusammenhang, dass die Variable veränderbar ist. Im Unterschied zu anderen Programmiersprachen sind in Rust per Default alle Variablen konstant. Wichtiger ist aber auch die Feststellung, dass die geschweiften Klammern in diesem Beispiel essenziell sind. Ohne sie verweigert der Compiler in der letzten Zeile das Ausleihen des Vektors an `println`, da nicht `x`, sondern `z` der Besitzer des Vektors ist. Es gilt die Grundregel, dass nur der aktuelle Besitzer etwas verleihen darf.

Das Ownership-/Borrowing-Konzept ist gewöhnungsbedürftig und erfordert ein Umdenken. So stellt eine doppelte verkettete Liste, die per se mehrere Besitzer kennt, in Rust eine kleine Herausforderung dar. Allerdings bietet das Konzept auch deutliche Vorteile. Die Speicherverwaltung ist sicher und nebenläufiger Code lässt sich bei nur einem Besitzer einfach realisieren.

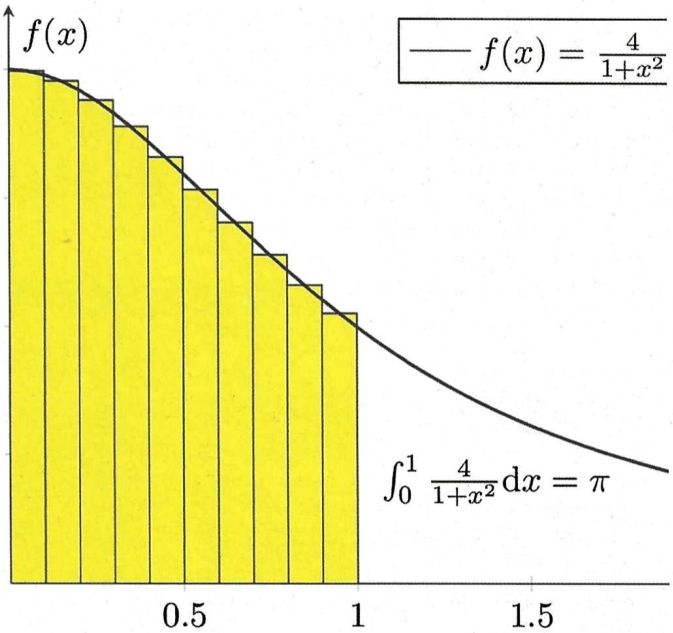
Unterstützung beim Erstellen nebenläufigen Codes

Um zu veranschaulichen, wie sich mit Rust nebenläufiger Code erstellen lässt, wird die Zahl π parallel bestimmt. Im Prinzip geht es darum, das Integral $f(x) = \frac{4}{1+x^2}$ von 0 bis 1 zu bestimmen. Eine klassische Annäherung stellt das Zeichnen von Rechtecken unterhalb der Funktion `f` dar. Die Flächeninhalte sind einfach zu bestimmen, und je schmaler die Rechtecke ausfallen, umso genauer ist das Ergebnis.

In Sprachen wie C/C++ oder auch Java ist es relativ einfach, fehlerhaften Code mit Wettlaufsituationen zu erzeugen. Der Beispielcode in Listing 6 versucht π mit Threads zu berechnen. Er beinhaltet allerdings eine Wettlaufsituation bezüglich der Variable `sum` und je nach konkreter Ausführung des Programms auf der CPU liefert diese das falsche oder korrekte Ergebnis zurück.

Portiert man diesen Code direkt nach Rust, sieht er ähnlich wie der in Listing 7 aus. Der Rust-Compiler übersetzt diesen Code aber nicht, sondern meldet vielmehr verschiedene Fehler.

In Rust dürfen Threads nur mit Variablen arbeiten, deren Laufzeit mindestens so lange ist wie die des Threads, in dem sie genutzt werden. Für die Threads der Rust-Standardbibliothek ergibt sich dadurch, dass die Variable entweder dem Thread selbst gehören muss oder aber die Variable `static lifetime` besitzt, also über die gesamte Laufzeit des Programms hinweg existiert. Diese Regel verhin-



Schematische Darstellung der π-Approximation (Abb. 1).

dert, dass ein Thread auf Variablen zugreift, die zwar zum Start des Threads existiert haben, anschließend aber gelöscht wurden, bevor der Thread beendet war. Die Variablen `step` und `sum` verletzen diese Regel, da sie auf dem Stack angelegt wurden – die erzeugten Threads können länger leben, als die Funktion, die sie erzeugt hat.

Für die Variable `step` lässt sich dieses Problem einfach lösen, da sie eine unveränderliche Variable ist und jeder Thread eine Kopie dieser Variablen bekommen kann. Rust benutzt dafür das Schlüsselwort `move` (`thread::spawn(move ||...)`, das dem neu erzeugten Thread eine unveränderliche Kopie aller Variablen mitgibt, auf die im Thread zugegriffen wird. Im Unterschied zur unveränderlichen Variablen `step` verändern die erzeugten Threads `sum`, sodass sich in deren Fall auch durch das Hinzufügen des Schlüsselworts `move` nicht alle Compilerfehler auflösen lassen.

Listing 6: Berechnung von π mit Threads

```
const double step = 1.0 / NUM_STEPS;
 double sum = 0.0;

 for (int tid = 0; tid < nthreads; ++tid) {
    std::thread t([&](int start, int end){

        for (int i = start; i < end; i++) {
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
    }, (NUM_STEPS / nthreads) *  tid
     , (NUM_STEPS / nthreads) * (tid + 1));
 }
```

Listing 7: Rust meldet Fehler wegen zu kurzer Laufzeit der Variablen.

```
let step = 1.0 / NUM_STEPS as f64;
 let mut sum = 0.0 as f64;

 let threads: Vec<_> = (0..nthreads)
    .map(|tid| {
        thread::spawn(|| {
            let start = (NUM_STEPS / nthreads) * tid;
            let end = (NUM_STEPS / nthreads) * (tid+1);

            for i in start..end {
                let x = (i as f64 + 0.5) * step;
                sum += 4.0 / (1.0 + x * x);
            }
        })
    }).collect();
```

Listing 8: Mutex schützt Variable.

```
static readonly_number: u64 = 42;
 static counter: Mutex<u64> = Mutex::new(0);

 pub fn init() {
    let guard = counter.lock().unwrap();
    guard = readonly_number;
 }
```

Listing 9: Arc verwaltet den Mutex

```
let sum = Arc::new(Mutex::new(0.0 as f64));

 let threads: Vec<_> = (0..nthreads).map(|tid| {
    let sum = sum.clone();

    thread::spawn(move || {
        let start = (NUM_STEPS / nthreads) *  tid;
        let end =   (NUM_STEPS / nthreads) * (tid+1);
        for i in start..end {
            let x = (i as f64 + 0.5) * step;
            *sum.lock().unwrap() += 4.0 / (1.0 + x * x);
        }
    })
 }).collect();
```

Mutexe als Variablen-Schutz

Eine Möglichkeit, den Code fehlerfrei zu implementieren, sind Mutexe. Ein Mutex in Rust nimmt ähnlich wie ein Mutex in C++ die zu schützende Variable auf und synchronisiert alle Zugriffe auf diese Variable. Der Quellcode in Listing 8 zeigt ein einfaches Beispiel, wie sich ein Mutex nutzen lässt.

In der Funktion `init` erfolgt der Versuch, über die Funktion `lock()` Zugriff auf die Variable zu bekommen, die der Mutex schützt. Sollte dabei kein Fehler ausgelöst werden, blockiert die Funktion `lock()` so lange, bis der aufrufende Thread exklusiven Zugriff auf diese Variable hat. Der Thread gibt die geschützte Variable automatisch wieder frei, sobald die entsprechende Referenz (im Beispiel `guard`) zerstört ist.

Gemeinsame Variable (Arc & Co.)

Mutexes lösen das Problem der Wettlaufsituation der Variable `sum` aus dem π-Beispiel, allerdings ist dabei sicherzustellen, dass der Mutex mindestens so lange lebt wie die entsprechenden Threads. Eine Möglichkeit, das Lebenszeitproblem von `sum` zu lösen, ist explizit einen global veränderlichen Mutex anzulegen. Dies schränkt allerdings die Struktur des Codes deutlich ein, da zum Beispiel jeder Versuch, π zu berechnen, neue Threads erzeugt, die alle denselben Mutex (und entsprechend dieselbe Summe) benutzen. Eine Alternative dazu sind `Rc` beziehungsweise der Thread-sichere Gegenpart `Arc`, die es ermöglichen Variablen auf dem Heap anzulegen. Rust benutzt Referenzzählung um sicherzustellen, dass die über `Rc` und `Arc` angelegten Variablen so lange existieren, bis keine Referenz auf die entsprechende Variable im Programm existiert.

Parallele Berechnung der Zahl π

Um die Zahl π berechnen zu können, gilt es, die Wettlaufsituation aus dem ursprünglichen Beispiel zu vermeiden. Es liegt daher nahe, die Variable `sum` durch ein Mutex zu schützen. Allerdings lebt der Mutex weiterhin nicht so lange wie die Threads, die ihn benutzen. Aus diesem Grund kommt ein `Arc` zur Verwaltung des Mutex zum Einsatz. Jeder Thread erhält durch die `clone`-Funktion einen eignen `Arc`, die alle auf denselben Mutex zeigen. Der Mutex wird erst wieder freigegeben, wenn keine Referenz auf dieses Objekt existiert. Im Beispiel in Listing 9 bedeutet das, dass alle Threads beendet sind und die Referenz wieder freigibt.

Das Anfordern eines Schreibzugriffs ist bei Mutexes eine recht zeitaufwendige Funktion. Insbesondere im vorliegenden Beispiel ist die Verwendung eines Mutex eine schlechte Wahl, da der synchronisierte Zugriff im Verhältnis zur restlichen Berechnung recht umfangreich ausfällt. Effizienter wäre die Berechnung von Teilergebnissen, die sich später zum Gesamtergebnis zusammenführen lassen (siehe Listing 10).

In diesem Beispiel berechnet jeder Thread seine Teilsumme (`partial_sum`) lokal und liefert sie beim Beenden als Ergebnis zurück. Einen Rust-Einsteiger dürfte an dieser Stelle verwundern, dass das Teilergebnis nicht per `return`-Anweisung kommt. Die Variable ohne abschließendes Semikolon am Schluss eines Blocks stellt hierfür die verkürzte Schreibweise dar. Die Ergebnisse sind im obigen Beispiel in einer `Map` abgelegt und mit der Methode `collect` aufsummiert. Diese Vorgehensweise macht die Verwendung von Mutexes überflüssig und garantiert darüber hinaus eine bessere Skalierbarkeit.

Listing 10: Vom Teil- zum Gesamtergebnis

```
let step = 1.0 / NUM_STEPS as f64;
let sum = 0.0 as f64;

let threads: Vec<_> = (0..nthreads)
    .map(|tid| {
        thread::spawn(move || {
            let mut partial_sum = 0 as f64;
            for i  in start..end {
                let x = (i as f64 + 0.5) * step;
                partial_sum += 4.0 / (1.0 + x * x);
            }
            partial_sum
        })}).collect();
```

Listing 11: Lastverteilung über Threads

```
let step = 1.0 / NUM_STEPS as f64;

let sum: f64 = (0..NUM_STEPS).into_par_iter()
    .map(|i| {
        let x = (i as f64 + 0.5) * step;
        4.0 / (1.0 + x * x)
    }).sum();
```

Listing 12: Parallelisierung mit Nachrichten

```
let (tx, rx) = mpsc::channel();
let mut sum = 0.0;

for id in 0..nthreads {
    // The sender endpoint can be copied
        let thread_tx = tx.clone();
    let start = (NUM_STEPS / nthreads as u64) * id;
    let end = (NUM_STEPS / nthreads as u64) * (id+1);

    // Each thread will send its partial sum via the channel
        thread::spawn(move || {
        let partial_sum = /* calculate the surface area */
thread_tx.send(partial_sum).unwrap();
    });
};

for _ in 0..nthreads {
    // The `recv` method picks a message from the channel
    // `recv` will block the current thread if there no messages
available
    sum = sum + rx.recv().unwrap();
}
```

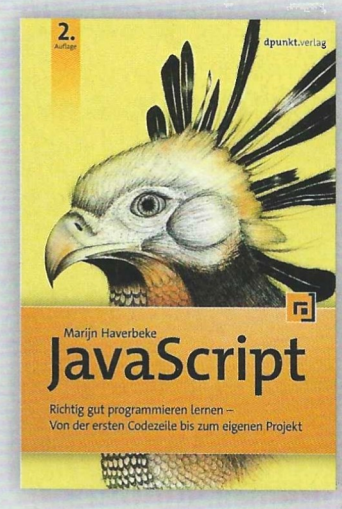
Datenparallelität mit Rayon

Bisher kamen allein Standardkomponenten von Rust zum Einsatz, um nebenläufigen oder parallelen Code zu schreiben. Softwareentwicklern kommt dabei die Verantwortung zu, beispielsweise die Verteilung der Last über die Threads selbst durchzuführen. In C/C++ ließe sich dieser Schritt zum Beispiel durch OpenMP vereinfachen. Mit Rayon existiert eine Bibliothek für Rust, die vergleichbare Ziele verfolgt und sowohl Task- als auch Datenparallelität unterstützt (siehe Listing 11).

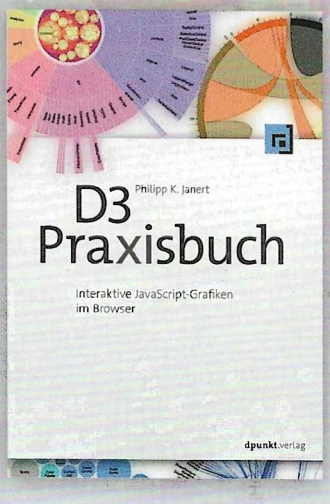
Die Funktion `into_par_iter` durchläuft parallel eine Schleife von 0 bis `NUM_STEPS`. Für die Berechnung automatisch erzeugte Threads bekommen einen Teil von der Schleife zugewiesen. Wie groß diese Teilbereiche sind, entscheidet Rayon eigenständig. Wie bei OpenMP haben Softwareentwickler aber die Freiheit, die Zuweisung zu beeinflussen oder sie komplett der Bibliothek zu überlassen.

Nachrichtenaustausch als Parallelisierungskonzept

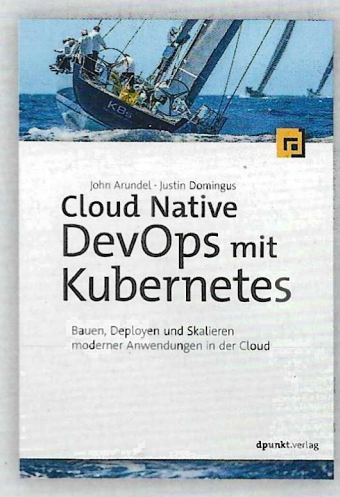
Im Allgemeinen gilt die Parallelisierung über den gemeinsamen Speicher als fehleranfällig. Zudem lässt sich die Korrektheit einer solchen Vorgehensweise schwierig belegen. Wie gezeigt, bie-



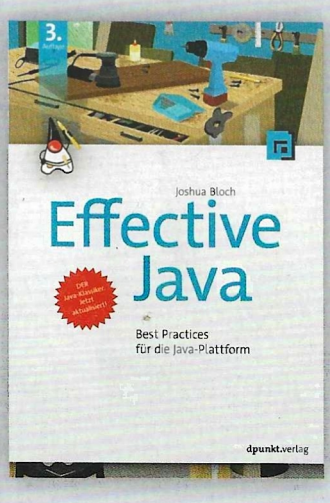
2. Auflage
2020, 488 Seiten
€ 32,90 (D)
ISBN 978-3-86490-728-9



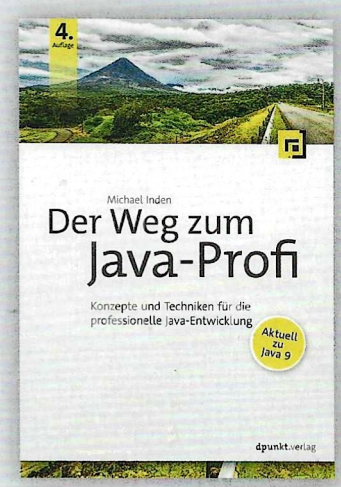
2020, 268 Seiten
€ 32,90 (D)
ISBN 978-3-86490-725-8



2019, 368 Seiten
€ 39,90 (D)
ISBN 978-3-86490-698-5



3. Auflage
2018, 410 Seiten
€ 36,90 (D)
ISBN 978-3-86490-578-0



4. Auflage
2018, 1416 Seiten
€ 49,90 (D)
ISBN 978-3-86490-483-7

plus
Buch + E-Book:
www.dpunkt.plus

dpunkt.verlag
www.dpunkt.de
Wieblinger Weg 17 · D-69123 Heidelberg · fon: 0 62 21 / 14 83 40
fax: 0 62 21 / 14 83 99 · e-mail: bestellung@dpunkt.de

Interview mit Jan-Erik Rediger, Telemetry Engineer bei Mozilla

heise Developer: *Woher kommen die meisten Rust-Entwickler und was ist ihre Motivation für Rust?*

Jan-Erik Rediger: Wir sehen Entwickler aus allen Programmiersprachen zu Rust kommen. Natürlich beobachten wir einige C/C++-Entwickler, da Rust in eine ähnliche Nische vordringt. Wir sehen aber auch viele Leute, die von High-Level-Programmiersprachen wie Ruby, Python oder JavaScript kommen.

Für viele sind die Security-Features ein Grund für Rust, aber auch die Paradigmen, die Rust bietet. Für die meisten ist es eher die zweite oder dritte Programmiersprache, die sie lernen.

heise Developer: *Können Sie ein paar Anwendungsbereiche nennen, für die sich Rust bei neuen Projekten besonders eignet?*

Rediger: Ein wichtiger Bereich ist die Mobilentwicklung. Die App-Entwicklung gerade für unterschiedliche Betriebssysteme läuft häufig darauf hinaus, dass man Code doppelt und dreifach schreibt.

Ein guter Ansatz ist, zumindest den Code der Applikationslogik in Rust zu bauen und auf mehreren Plattformen gleich nutzen zu können. Der Bereich ist mir persönlich äußerst wichtig. Und es gibt sicherlich ein paar Ecken und Kanten, die wir noch ausbügeln müssen.

Ein anderer Bereich ist Embedded und das Internet der Dinge, bei dem man auf Geräte mit begrenzten Ressourcen stößt, beispielsweise IoT-Gateways oder Sensoren. In dieser Ecke dominiert derzeit nach wie vor C, aber wir sehen eine gute Entwicklung bei Rust auch bezüglich von Libraries.

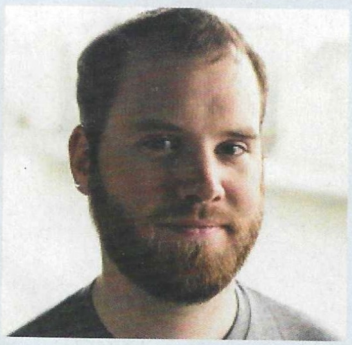
Das stößt auch in der Industrie auf Interesse, zumal C eben nicht die Garantien bezüglich der Sicherheit bietet wie Rust.

heise Developer: *Welche Rolle spielt die Interoperabilität mit C?*

Rediger: Wichtig ist, dass wir mit C-Libraries interagieren können, gerade auch im Embedded-Bereich. Das bedeutet freilich, dass wir das sichere Umfeld verlassen, aber das war von Anbeginn so gedacht. Es war uns immer klar, dass es kaum Entwickler gibt, die ein Projekt komplett umschreiben. Rust kann dazu genutzt werden, iterativ Projekte umzuschreiben, gerade weil man mit anderen Bibliotheken und Sprachen interagieren kann.

Es gibt durchaus noch Probleme von der Rust-Seite, wie das Zusammenspiel funktioniert, aber wir arbeiten daran, die Voraussetzungen zu verbessern.

Das Interview führte Rainald Menge-Sonnentag, Redakteur von heise Developer. Das vollständige Interview lesen Sie auf heise Developer (ix.de/z9jr).



Jan-Erik Rediger, Telemetry Engineer, Mozilla

tet Rust Anwendungsmöglichkeiten, um diese Probleme zu minimieren. Darüber hinaus bietet sich aber auch ein Mechanismus an, der quasi Communicating Sequential Processes (CSP) für Rust darstellt. Die Grundlagen von CSP hat Tony Hoare bereits im Jahr 1978 veröffentlicht. Sie stellen eine Prozessalgebra zur Beschreibung von Interaktion zwischen kommunizierenden Prozessen dar. CSP eignet sich hervorragend, um nebenläufige Anwendungen zu spezifizieren und zu verifizieren. Vereinfacht gesagt, stellt CSP das theoretische Modell für die Parallelisierung mit Nachrichten dar und bildet somit die Grundlage für das Message Passing Interface (MPI).

Das Beispiel in Listing 12 veranschaulicht den Aufbau von Kanälen zwischen den einzelnen Threads. Jeder Thread sendet seine Teilergebnisse zum Hauptthread, der diese einsammelt und aufsummiert. Die Problematik einer Wettlaufsituation lässt sich damit vollständig vermeiden.

Fazit

Die Beispiele verdeutlichen, dass Rust sehr gut für die Entwicklung von parallelen und nebenläufigen Anwendungen geeignet ist, da der Compiler den Softwareentwickler bereits während der Entwicklung vor im Nachhinein nur noch schwer zu findenden Bugs bewahrt. Die Regeln mögen auf den ersten Blick recht kompliziert und aufwendig erscheinen – und sie erhöhen auch die Zeit, bis eine erste lauffähige Software vorhanden ist –, allerdings reduziert sich nach Erfahrung der Autoren der Aufwand bei der Fehlersuche deutlich. Darüber hinaus ist die Ausführungsgeschwindigkeit vergleichbar mit C/C++-Versionen eines Programms. Zumindest der aktuelle Rust-Compiler muss den Vergleich zu GCC und Clang nicht scheuen. (map@ix.de)

Quellen

[1] ix.de/z9jr

[2] J. Blandy and Jason Orendorff, Programming Rust, O'Reilly UK Ltd., 2018

[3] S. Klabnik and C. Nicols, The Rust Programming Language (Manga Guide), No Starch Press, 2018

Dr. Jens Breitbart

arbeitet als Softwarearchitekt bei Driver Assistance, Robert Bosch GmbH. Er forschte fast zehn Jahre im Bereich des High Performance Computing und war bis Oktober vergangenen Jahres Mitarbeiter am Lehrstuhl für Rechnertechnik und Rechnerorganisation der TU München beschäftigt. Er arbeitet privat in verschiedenen Softwareprojekten mit, u. a. HermitCore.

Dr. Stefan Lankes

arbeitet als akademischer Oberrat am Institute of Automation of Complex Power Systems der RWTH Aachen University. Er forscht seit circa 20 Jahren im Bereich der systemnahen Software für Hochleistungsrechner und echtzeitfähige Systeme. Unter anderem ist er Initiator des Open-Source-Projekts HermitCore.