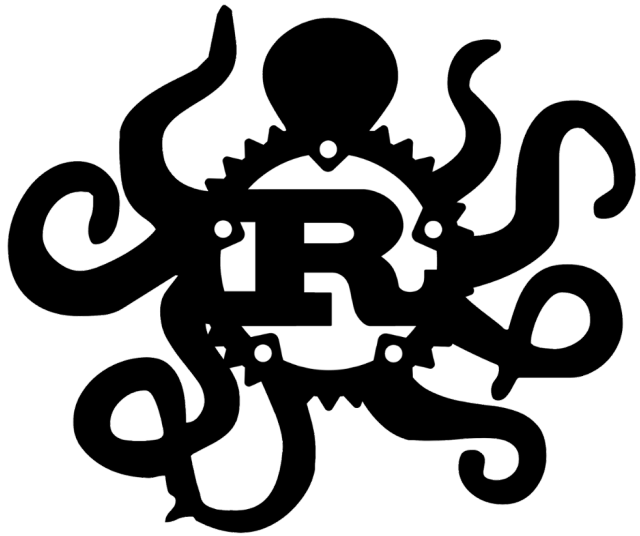
Karlsruher Institut für Technologie

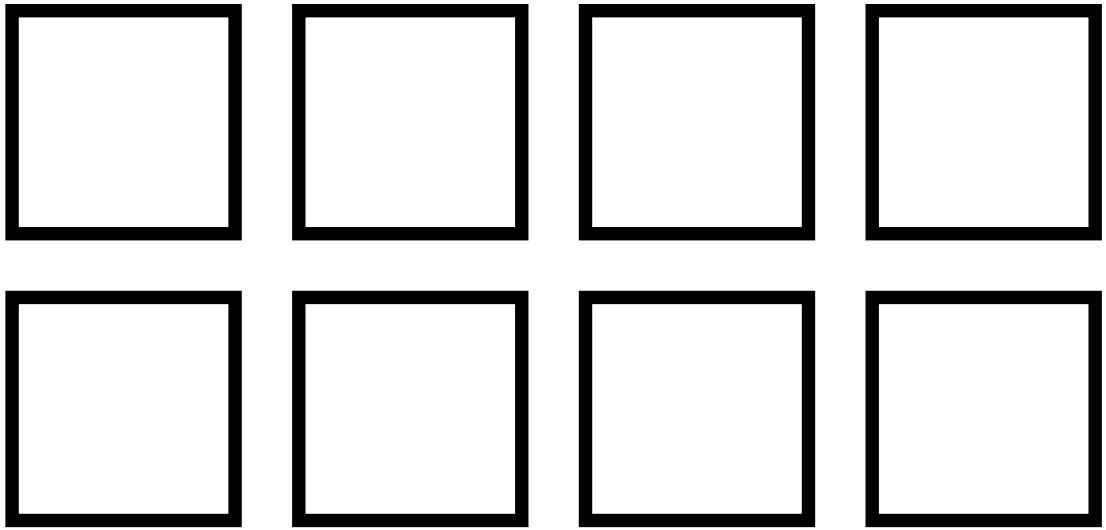
Invasives Rust

Hermann Heinz Erich Krumrey

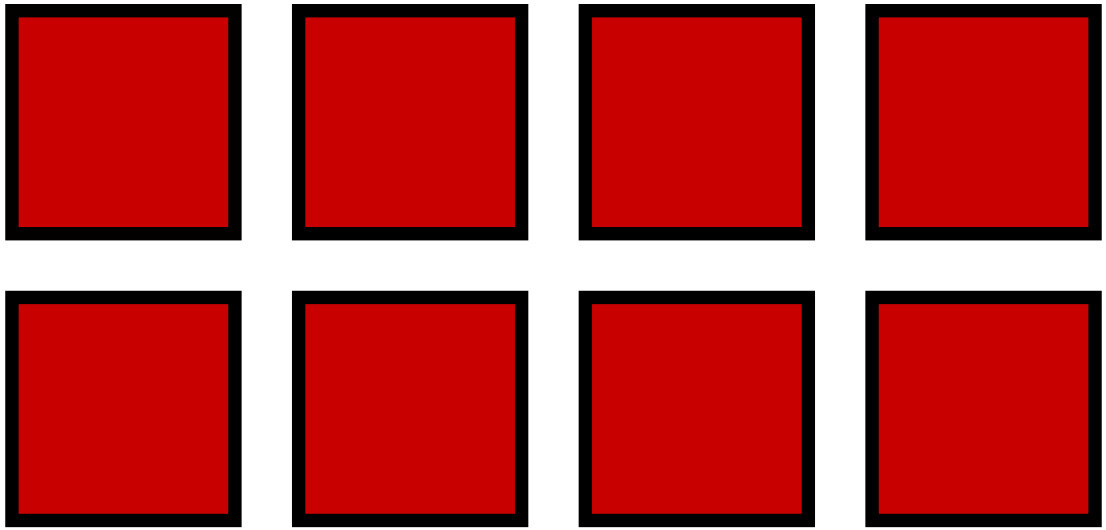
Lehrstuhl Programmierparadigmen, IPD Snelting



Motivation

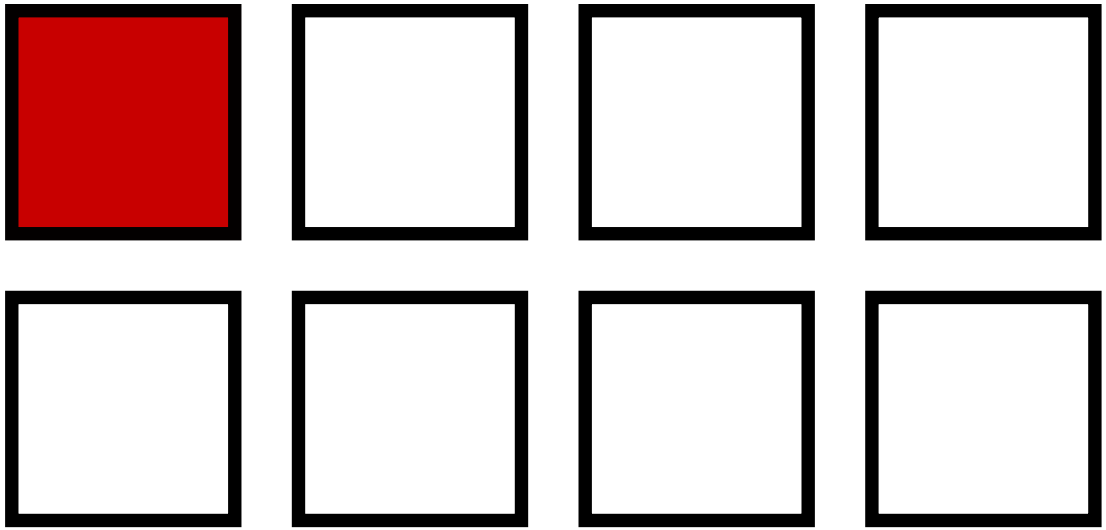


Motivation



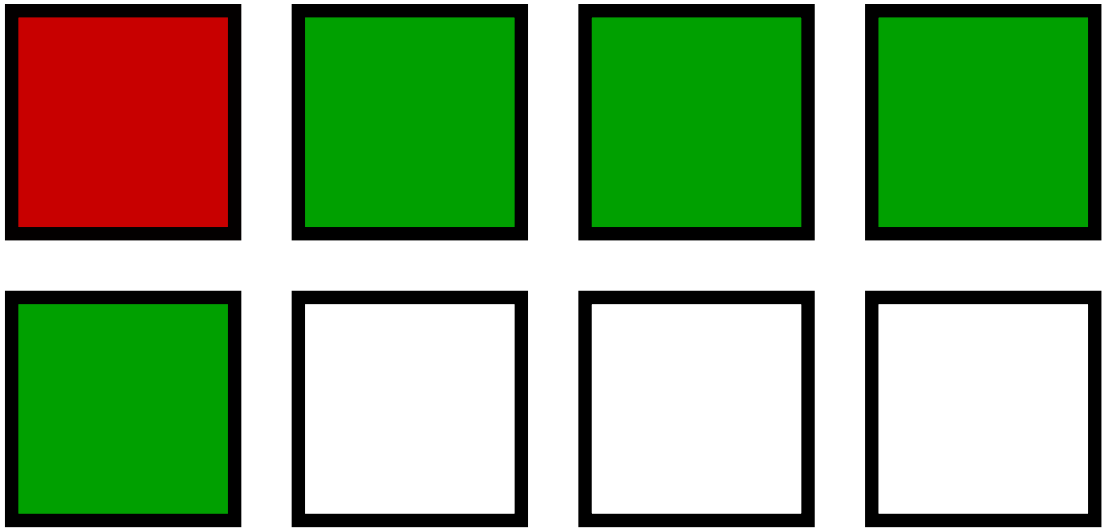
1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten

Motivation



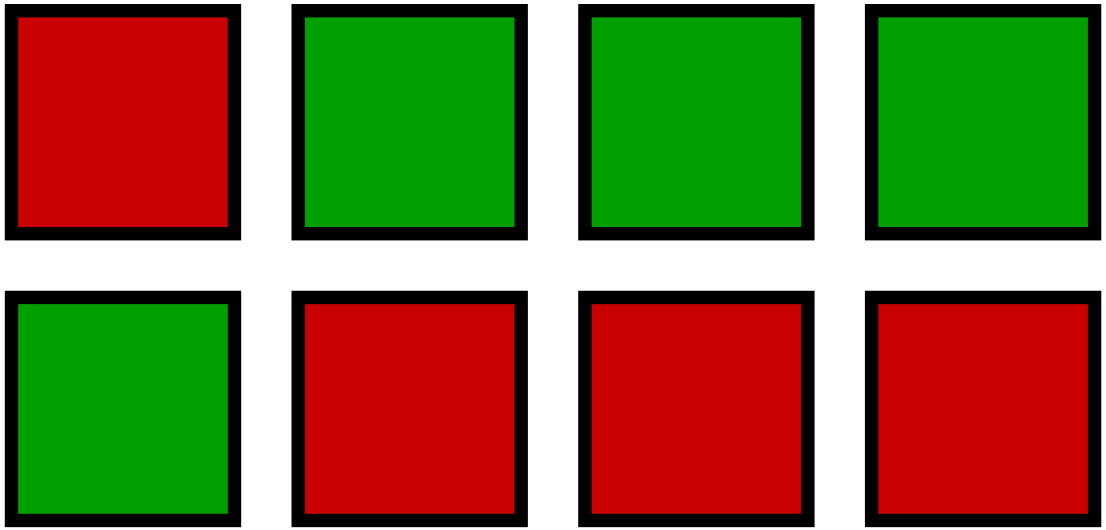
1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten
2. Programm 1 sendet Ergebnisse über das Netzwerk

Motivation



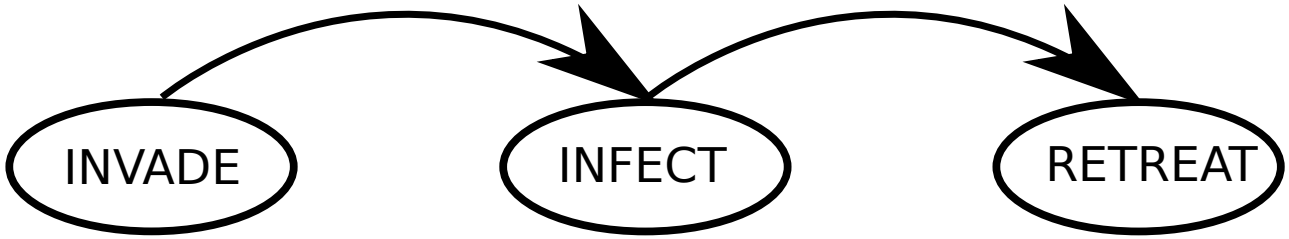
1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten
2. Programm 1 sendet Ergebnisse über das Netzwerk
3. Programm 2 beginnt Ausführung auf 4 Recheneinheiten

Motivation



1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten
2. Programm 1 sendet Ergebnisse über das Netzwerk
3. Programm 2 beginnt Ausführung auf 4 Recheneinheiten
4. Programm 1 führt wieder Berechnungen aus, jetzt auf 4 Recheneinheiten

Invasives Rechnen



- Ressourcenbewusstes Programmieren
- 3 Phasen:
  1. Invade - Ressourcen reservieren
  2. Infect - Ressourcen nutzen
  3. Retreat - Ressourcen freigeben
- OctoPOS und iRTSS bieten Software-Grundlage
- Unterstützung der Programmiersprachen C, C++ und X10

Rust - Motivation

- Sichere Speicherzugriffe
- Effiziente und weniger fehleranfällige Parallelberechnung
- Konzeptionell an C-artige Sprachen angelehnt
- Höhere Abstraktionen, um den Einstieg zu erleichtern
- Speichersicherheit und Abstraktionen sollen nicht auf Kosten der Leistung erreicht werden

Rust - Ownership, Move-Semantik und Referenzen

- Ownership
  - Das zentrale Alleinstellungsmerkmal der Programmiersprache
  - Jeder Speicherbereich wird nur einer einzigen Variable zur Verfügung gestellt
  - Beim Verlassen des Geltungsbereichs wird der Speicherbereich freigegeben
- Move-Semantik
  - Ownership kann auf andere Variablen übertragen werden
  - Ursprüngliche Variable ist nach einem „Move“ nicht mehr verwendbar
- Referenzen
  - Unendliche unveränderliche Referenzen
  - Nur eine veränderliche Referenz
  - Move einer Variable nicht möglich wenn Referenzen existieren

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");



...
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;



...
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(a);


...
```

```
error[E0382]:
use of moved value: ‘a‘
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(b);


...
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(b);
  f(b);

...
```

```
error[E0382]:
use of moved value: ‘b‘
```

Rust - Beispiel 2

```
fn g(s: &String) { ... }
fn h(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");




...
```

Rust - Beispiel 2

```
fn g(s: &String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;



...
```

Rust - Beispiel 2

```
fn g(s: &String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;
  let x_ref2 = &x;


...
```

Rust - Beispiel 2

```
fn g(s: &String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &mut x;
  let x_ref2 = &x;


...
```

Rust - Beispiel 2

```
fn g(s: &String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &mut x;
  let x_ref2 = &mut x;


...
```

Rust - Traits

- Wie Interfaces in anderen Sprachen
- Können Verhalten bezüglich Ownership und Move-Semantik beeinflussen
- Drop
  - Destruktor
  - drop()-Methode wird beim Verlassen des Geltungsbereichs aufgerufen
- Copy
  - Erlaubt Copy-Semantik statt Move-Semantik
  - Jedes mal, wenn ein Move geschehen würde, wird eine bitweise Kopie ausgeführt
  - Primitve Datentypen wie i32 implementieren dieses Trait

octorust

- Hilfsprogramm zum Kompilieren von invasiven Rust-Programmen
- Unetrstützt die SPARC-V8 Architektur
- Automatische Verlinkung mit iRTSS/OctoPOS
- ca. 650 Zeilen Code (Python)

octolib

- Bibliothek mit invasiven Strukturen und Funktionen
- Direkte C-Rust Bindings
- Rust-spezifische Anpassungen
- ca. 750 Zeilen Code (Rust)

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");




...
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;



...
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(a);


...
```

```
error[E0382]:
use of moved value: ‘a‘
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(b);


...
```

Rust - Beispiel

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(b);
  f(b);

...
```

```
error[E0382]:
use of moved value: ‘b‘
```

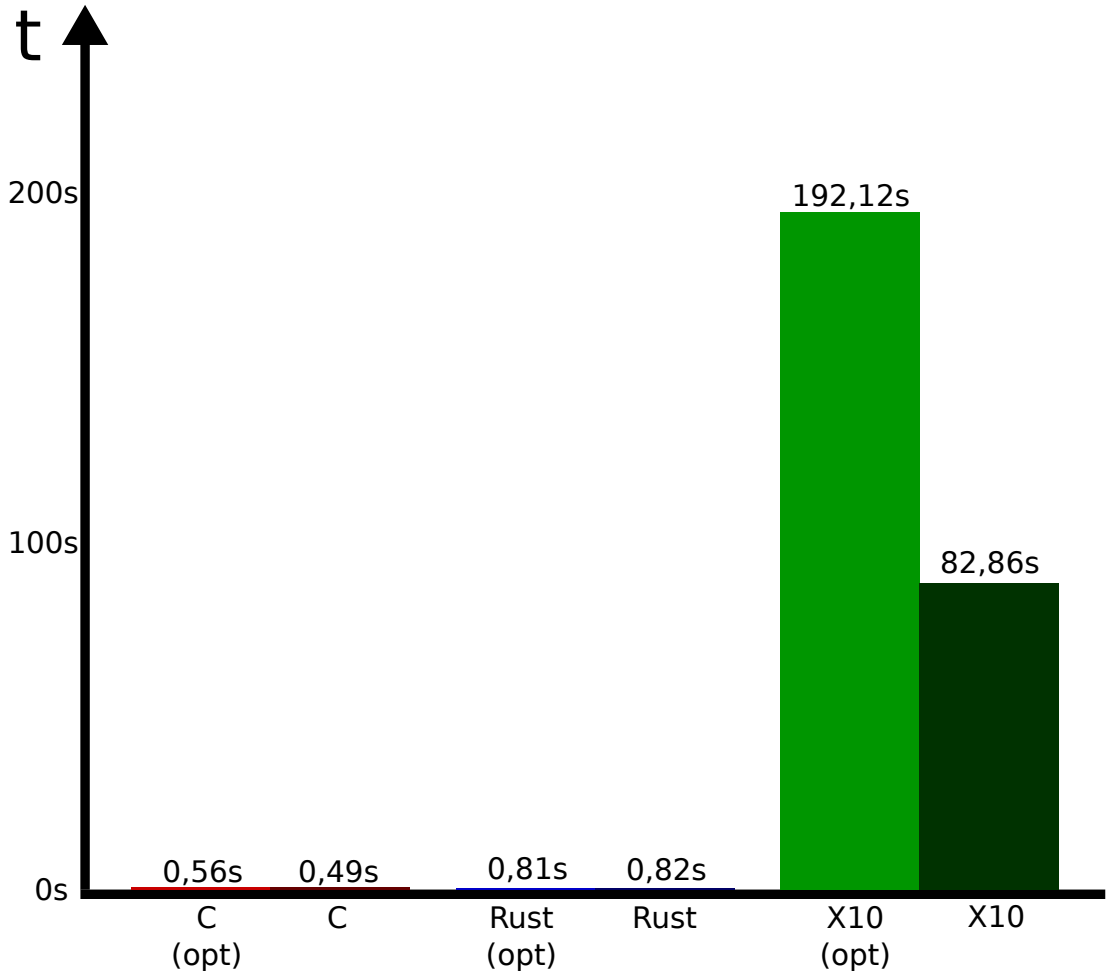
Beispiel eines invasiven Rust-Programms

```
    pub extern "C" fn rust_main_ilet (tile_id: u8) { // TODO
Check tile_id name
    //Invade
    let constraints = Constraints::new(4, 8);
    let claim = AgentClaim::new(constraints);

    //Infect
    let ilet_fn = |param: *mut c_void| {
        print("Hello␣World!\n\0");
    }
    claim.infect(ilet_fn, None);

  } // Retreat & Shutdown
```

Evaluation

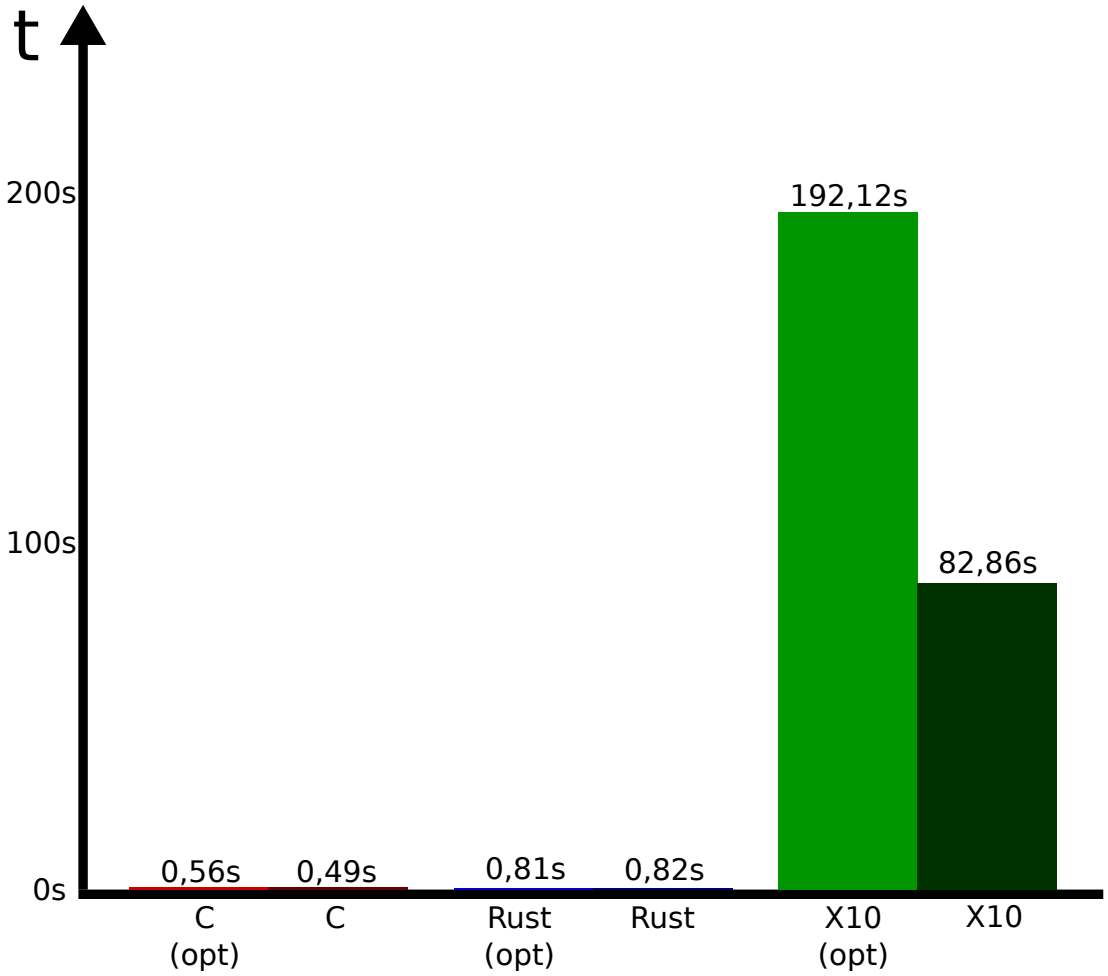
- Wiederholte Laufzeitmessungen
- Betrachtung von Programmen mit und ohner Compiler-Optimierungen
- Unterschiedliche Programme
  - Kompilierungsdauer
  - Anlaufzeit
  - Parallele Primzahlenberechnung
  - Allokation von Objekten

Kompilierungsdauer



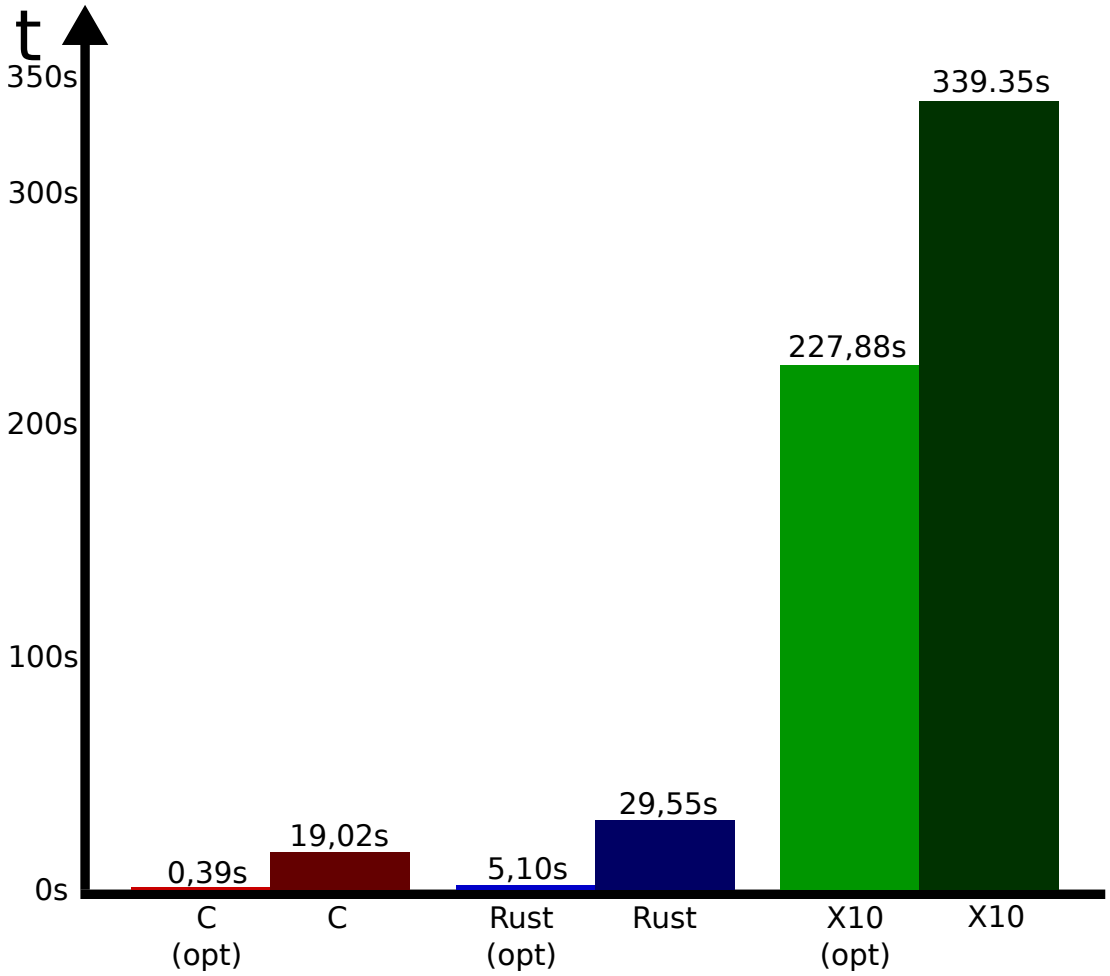
- X10 langsam

Anlaufzeit



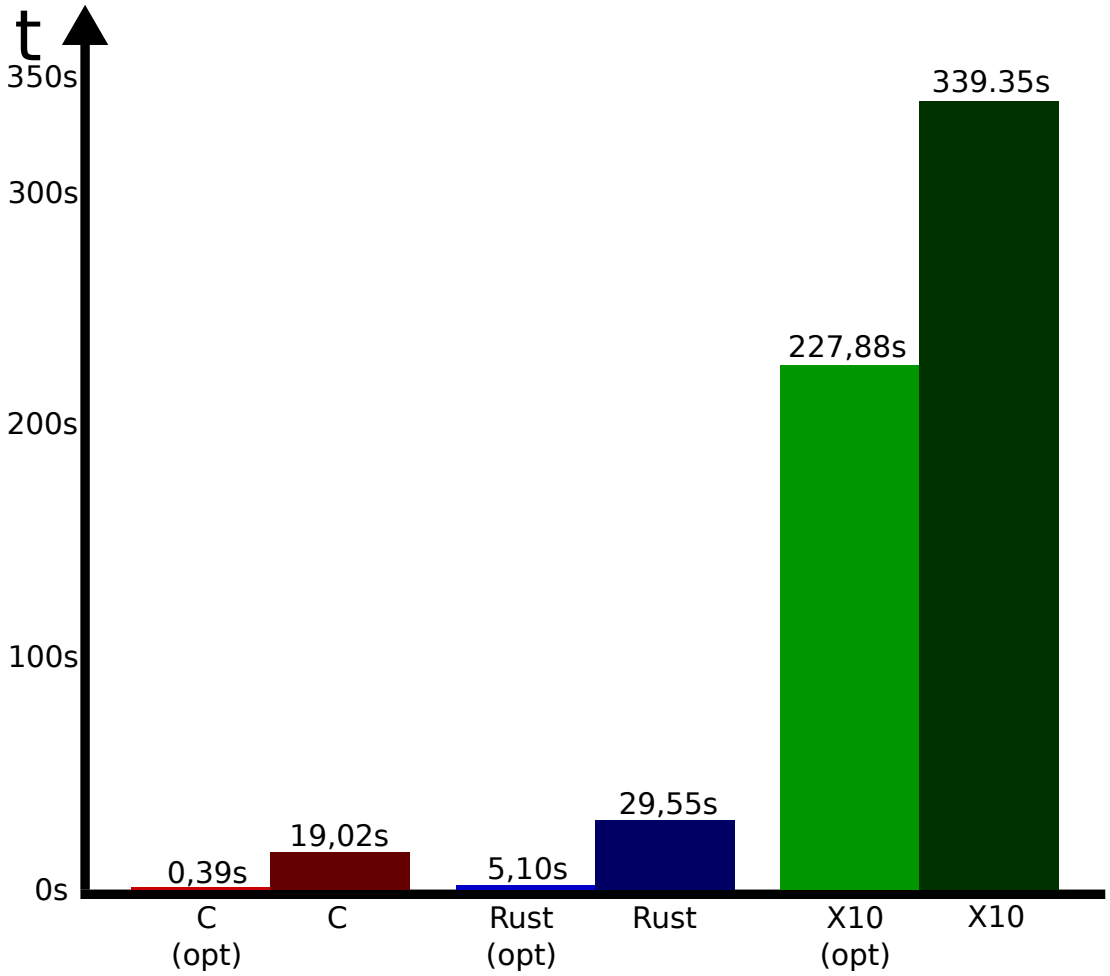
- X10 langsam

Parallele Primzahlenberechnung



- Rust langsam

Allokation von Objekten



- X10 langsam

Zusammenfassung und Fazit

Vielen Dank für ihre Aufmerksamkeit!