When is Parallelism Fearless and Zero-Cost with Rust?

Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, Mark C. Jeffrey

SPAA 2024
Rust claims to solve the arduous task of parallel programming. Rust provides fearless concurrency.

Does Rust make expressing all parallel programs easy?

As it is,

- It makes expressing easy parallelism fearless
- Its support is limited for hard cases of parallelism
Contributions

A case study of Rust’s support for regular and irregular parallelism
- Classify parallel patterns found in PBBSv2 and MultiQueue-based algorithms
- Qualitatively evaluates the level of support for each pattern

Rust Parallel Benchmark Suite (RPB):
A benchmark suite with regular and irregular parallelism in Rust
- Performance evaluation of Rust vs. C++ benchmarks
Background

Types of parallelism
The regularity of algorithms affects ease of parallelization

Parallelism irregularity index

Operator

Set of tasks

Structure of data

Structured

Unstructured

Structure

Array

Matrix

Graph

[Pingali et al., PLDI 2011]
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]
The regularity of algorithms affects ease of parallelization

Parallel array summation

[Pingali et al., PLDI 2011]
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]
Background

Rust
Rust is gaining popularity

Rust is on its seventh year as the most loved language...
Rust is gaining popularity

Program with Rust!
Rust is gaining popularity because of its safety guarantees

Rust is gaining popularity because of its safety guarantees.

Rust catches all type and memory safety errors.

Rust claims to enable *fearless concurrency*!!!
Rust claims to enable **fearless concurrency!!!**

Does Rust cover all these branches?
Rust claims to enable **fearless concurrency!!!**

Does Rust cover all these branches?
Rust claims to enable **fearless concurrency**!!!

Does Rust cover all these branches?

---

**Fearless Concurrency with Rust (AM)**
Herbert Wolverson

---

Transaction Memory

---

Do fear conflicts occur, even if they rarely or never happen?

---

Do lens understand and need concurrent profiles for concurrent code. ACM

---

Does the hope for mastering the content of concurrency widespread because

---

"Fear of conflict, even if they rarely or never happen, is an alternative and a block of code. ACM..."
Rust claims to enable **fearless concurrency!!!**

```
Chapter 16: Fearless Concurrency

Chapter 16 | Object-oriented Programming Features of Rust

Fearless Concurrency with Rust (AM)
Herbert Wolverson

Does Rust cover all these branches?
```

---

**Transaction Memory**

Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel

Is Transactional Programming Actually Easier?

---

**Parallelism Irregularity Index**

- Structure of data
  - Structured
  - Unstructured
- Read-only
  - Local read-write
  - Arbitrary read-write
- Dispatch
  - Static
  - Dynamic
- Order
  - Unordered
  - Ordered

---

**ACM Publications**

- 1993
- 1994
- 1995
- 1996
- 1997
- 1998
- 1999
- 2000
- 2001
- 2002
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
- 2014
- 2015
- 2016
- 2017
- 2018
- 2019
- 2020
- 2021
- 2022
- 2023

**Google Trends**

- 1993
- 1994
- 1995
- 1996
- 1997
- 1998
- 1999
- 2000
- 2001
- 2002
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
- 2014
- 2015
- 2016
- 2017
- 2018
- 2019
- 2020
- 2021
- 2022
- 2023
Rust claims to enable **fearless concurrency**!!!
Rust claims to enable **fearless concurrency!!!**

The Dispatch Operator is a key concept in parallelism. It relates to the structure of data and its parallelism irregularity index. The structure of data can be either structured or unstructured, and the order of tasks can be either ordered or unordered.

Does Rust cover all these branches?

It’s time to study Rust’s capabilities and limitations for parallelism.
Clarifying “fearless concurrency”

**Fear**: Anticipation of concurrency errors that manifest at run time.

- **Fearless**: Concurrency errors get caught at compile time
- **Comfortable**: Errors get caught at run time (symptoms close to cause)
- **Scared**: Concurrency errors may happen without being detected
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
- Read-only accesses
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
- Read-only accesses
- Local read-writes on structured data
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
• Read-only accesses
• Local read-writes on structured data

Assume a static unordered set of tasks
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
Read-only accesses
Local read-writes on structured data

```rust
fn par_increment(v: &mut [u32]) {
    v.par_iter_mut().for_each(|vi| *vi+=1);
}
```
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
Read-only accesses
Local read-writes on structured data

```rust
fn par_increment(v: &mut [u32]) {
    v.par_iter_mut().for_each(|vi| *vi+=1);
}
```

Stride pattern on `v`
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
- Read-only accesses
- Local read-writes on structured data

```rust
fn par_increment(v: &mut [u32]) {
    v.par_iter_mut().for_each(|vi| *vi += 1);
}
```

Can only mutate `vi` inside a task.
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
- Read-only accesses
- Local read-writes on structured data

```rust
fn par_increment(v: &mut [u32]) {
    v.par_iter_mut().for_each(|vi| *vi+=1);
}
```

Task: Can only mutate `vi` ⇒ No errors
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
Read-only accesses
Local read-writes on structured data

```rust
fn par_increment(v: &mut [u32]) {
    v.par_iter_mut().for_each(|vi| *vi+=1);
}
```

Can only mutate `vi`:
No errors
Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:
- Read-only accesses
- Local read-writes on structured data

Rust provides fearlessness for regular accesses

```rust
fn par_increment(v: &mut [u32]) {
    v.par_iter_mut().for_each(|vi| *vi += 1);
}
```

- task: Can only mutate `vi` ⇒ No errors

Stride pattern on `v`
Rust rejects data-race-prone irregular accesses

Irregular accesses:

Assume a static unordered set of tasks
Rust rejects data-race-prone irregular accesses

Irregular accesses:
- **Arbitrary read-writes**

Assume a static unordered set of tasks
Rust rejects data-race-prone irregular accesses

Irregular accesses:
• Arbitrary read-writes
• Local read-writes on unstructured data

Assume a static unordered set of tasks
Rust rejects data-race-prone irregular accesses

```rust
// serial indirect increment
(0..n)
  .into_iter()
  .for_each(|i| {
    v[offsets[i]] += 1;
  })
```
Rust rejects data-race-prone irregular accesses

// serial indirect increment
(0..n)
  .into_iter()
  .for_each(|i| {
    v[offsets[i]] += 1;
  })

// parallel
(0..n)
  .into_par_iter()
  .for_each(|i| {
    v[offsets[i]] += 1;
  })
Rust rejects data-race-prone irregular accesses

// serial indirect increment (0..n)
.\texttt{into\_iter()}
.\texttt{for\_each(|i| \{}
  \texttt{v[offsets[i]] += 1;}
\texttt{\})}

// parallel (0..n)
.\texttt{into\_par\_iter()}
.\texttt{for\_each(|i| \{}
  \texttt{v[offsets[i]] += 1;}
\texttt{\})

\textbf{Risky}

---

(SngInd)

\begin{tikzpicture}
\end{tikzpicture}

(RngInd)

\begin{tikzpicture}
\end{tikzpicture}

(AW)
Rust rejects data-race-prone irregular accesses

// serial indirect increment
(0..n)
    .into_iter()
    .for_each(|i| {
        v[offsets[i]] += 1;
    })

// parallel
(0..n)
    .into_par_iter()
    .for_each(|i| {
        v[offsets[i]] += 1;
    })

error[E0596]: cannot borrow `v` as mutable, as it is a captured variable in a `Fn` closure

Compile error
Rust rejects data-race-prone irregular accesses

// serial indirect increment
(0..n)
    .into_iter()
    .for_each(|i| {
        v[offsets[i]] += 1;
    })

// parallel
(0..n)
    .into_par_iter()
    .for_each(|i| {
        v[offsets[i]] += 1;
    })

Offsets

Unique

Duplicates

Compile error

error[E0596]: cannot borrow `v` as mutable, as it is a captured variable in a `Fn` closure
Programmers face a conundrum expressing irregular accesses

Synchronization

\[
(0..n) \\
\text{.into\_par\_iter()} \\
\text{.for\_each(|i| \{ \\
\text{unsafe { \\
\hspace{1em}*(v.as\_ptr().add(i) as *mut T)+=1;
\text{}}
\text{}}
\text{\})}
\]

Old = v[offsets[i]].load(Order);

v[offsets[i]].store(old+1, Order);

v[offsets[i]].fetch_add(1, Order);

*v[offsets[i]].lock().unwrap()+=1;

v.lock().unwrap()[offsets[i]]+=1;
Programmers face a conundrum expressing irregular accesses

Synchronization

- Locks
- Coarse

```rust
(0..n)
.into_par_iter()
.for_each(|i| {
    unsafe {
        (*v.as_ptr().add(i) as *mut T)+=1;
    }
}
```

Old = v[offsets[i]].load(Order);

v[offsets[i]].store(old+1, Order);

v[offsets[i]].fetch_add(1, Order);

*v[offsets[i]].lock().unwrap()*+=1;

v.lock().unwrap()[offsets[i]]+=1;

}}

\( 0 \leq i \leq n-1 \)

\( 1 \leq n \leq 2^m \)

\( 0 \leq \text{offsets}[i] < n \)

Synchronization

Locks

Coarse

500x
Programmers face a conundrum expressing irregular accesses

```rust
(0..n)
    .into_par_iter()
    .for_each(|i| {
        *v[offsets[i]].lock().unwrap()+=1;
    })
```
Programmers face a conundrum expressing irregular accesses

```rust
(0..n)
.into_par_iter()
.for_each(|i| {
    unsafe {
        *v.as_ptr().add(i) as *mut T) += 1;
    }
    Old = v[offsets[i]].load(Order);
    v[offsets[i]].store(old+1, Order);
    v[offsets[i]].fetch_add(1, Order);
})
```

Synchronization

- Locks
  - Coarse 500x
  - Fine 5x
- Atomics

Atomics

(SngInd)

(RngInd)

(AW)
Programmers face a conundrum expressing irregular accesses

Synchronization

- Locks
  - Coarse  500x
  - Fine  5x
- Atomics
  - Relaxed  1x

\[
(0..n)\]

```python
.into_par_iter()
.for_each(|i| {

v[offsets[i]].fetch_add(1, Order);

})
```
Programmers face a conundrum expressing irregular accesses

```
(0..n)
    .into_par_iter()
    .for_each(|i| {
        v[offsets[i]].fetch_add(1, Order);
    })
```
Programmers face a conundrum expressing irregular accesses

Synchronization
- Locks
  - Coarse: 500x
  - Fine: 5x
- Atomics
  - Relaxed: 1x
  - SeqCst: 2x

(0..n)

```rust
    .into_par_iter()
    .for_each(|i| {
        Old = v[offsets[i]].load(Order);
        v[offsets[i]].store(old+1, Order);
        Atomicity violation -> compiles!!
    })
```
Programmers face a conundrum expressing irregular accesses

Synchronization
- Locks
  - Coarse: 500x
  - Fine: 5x
- Atomics
  - Relaxed: 1x
  - SeqCst: 2x
- Unsafe

(0..n)
    .into_par_iter()
    .for_each(|i| {
        Old = v[offsets[i]].load(Order);
        v[offsets[i]].store(old+1, Order);
        Atomicity violation -> compiles!!
    })

Unsafe

21
Programmers face a conundrum expressing irregular accesses

### Synchronization

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks</td>
<td></td>
</tr>
<tr>
<td>Coarse</td>
<td>500x</td>
</tr>
<tr>
<td>Fine</td>
<td>5x</td>
</tr>
<tr>
<td>Atomics</td>
<td></td>
</tr>
<tr>
<td>Relaxed</td>
<td>1x</td>
</tr>
<tr>
<td>SeqCst</td>
<td>2x</td>
</tr>
</tbody>
</table>

### Unsafe

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>unchecked</td>
<td>1x</td>
</tr>
</tbody>
</table>

```rust
unsafe {
    *(v.as_ptr().add(i) as *mut T)+=1;
}
```
Programmers face a conundrum expressing irregular accesses

<table>
<thead>
<tr>
<th>Synchronization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks</td>
<td></td>
</tr>
<tr>
<td>Coarse</td>
<td>500x</td>
</tr>
<tr>
<td>Fine</td>
<td>5x</td>
</tr>
<tr>
<td>Atomics</td>
<td></td>
</tr>
<tr>
<td>Relaxed</td>
<td>1x</td>
</tr>
<tr>
<td>SeqCst</td>
<td>2x</td>
</tr>
</tbody>
</table>

unsafe {
  *(v.as_ptr().add(i) as *mut T)+=1;
}

check_duplicates(offsets);
(0..n)
  .into_par_iter()
  .for_each(|i| {
    unsafe {
      *(v.as_ptr().add(i) as *mut T)+=1;
    }
  })

21
Programmers face a conundrum expressing irregular accesses

\[
(0..n)
\]

\[
\text{for_each}(\mid i \mid \{ \\
\text{unsafe} \{ \\
\text{*}(\text{v.as_ptr().add}(i) \text{ as } \ast\text{mut T})+=1;
\}
\})
\]

check_duplicates(offsets);

Synchronization

<table>
<thead>
<tr>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
</tr>
<tr>
<td>Fine</td>
</tr>
</tbody>
</table>

Offsets

<table>
<thead>
<tr>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>S</td>
</tr>
</tbody>
</table>

Unsafe

<table>
<thead>
<tr>
<th>unchecked</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>with check</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x</td>
</tr>
</tbody>
</table>

\[
\text{(SngInd)}
\]

\[
\text{RngInd}
\]

\[
\text{(AW)}
\]
Progammers face a conundrum expressing irregular accesses (0..n).

```rust
unsafe {
    *(v.as_ptr().add(i) as *mut T)+=1;
}
```

```rust
check_duplicates(offsets);
```

**Synchronization**
- **Locks**
  - Coarse: 500x
  - Fine: 5x
- **Atomics**
  - Relaxed: 1x
  - SeqCst: 2x

**Unsafety**
- unchecked: 1x
- with check: 3x

**Offsets**
- Unique: C/S
- Duplicates: C/S

---

**Synchronization**
- Locks
  - Coarse: 500x
  - Fine: 5x
- Atomics
  - Relaxed: 1x
  - SeqCst: 2x

**Unsafe**
- unchecked: 1x
- with check: 3x

---

```
check_duplicates(offsets);

(0..n)
    .into_par_iter()
    .for_each(|i| {
        unsafe {
            *(v.as_ptr().add(i) as *mut T)+=1;
        }
    })
```
Programmers face a conundrum expressing irregular accesses

Synchronization
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

Unsafe
- unchecked 1x
- with check 3x

Offsets
- Unique
- Duplicates

check_duplicates(offsets);
(0..n)
 .into_par_iter()
 .for_each(|i| {
   unsafe {
     *(v.as_ptr().add(i) as *mut T)+=1;
   }
 })

21
Programmers face a conundrum expressing irregular accesses

<table>
<thead>
<tr>
<th>Synchronization</th>
<th></th>
<th></th>
<th>Offsets</th>
<th>Unique</th>
<th>Duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks</td>
<td></td>
<td></td>
<td></td>
<td>C/S</td>
<td>S</td>
</tr>
<tr>
<td>Coarse</td>
<td>500x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fine</td>
<td>5x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomics</td>
<td></td>
<td></td>
<td></td>
<td>C/S</td>
<td>S</td>
</tr>
<tr>
<td>Relaxed</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SeqCst</td>
<td>2x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsafe</td>
<td></td>
<td></td>
<td></td>
<td>C/S</td>
<td>S</td>
</tr>
<tr>
<td>unchecked</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with check</td>
<td>3x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```rust
check_duplicates(offsets); (0..n)
    .into_par_iter()
    .for_each(|i| {
        unsafe {
            *(v.as_ptr().add(i) as *mut T)+=1;
        }
    })
```

<table>
<thead>
<tr>
<th>S</th>
<th>C/S</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SngInd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(RngInd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(AW)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21

Synchronization

Locks

- Coarse: 500x
- Fine: 5x

Atomics

- Relaxed: 1x
- SeqCst: 2x

Unsafe

- unchecked: 1x
- with check: 3x
Programmers face a conundrum expressing irregular accesses

```rust
fn check_duplicates(offsets: &Vec<[T; n]>) {
    for_each(|i| {
        unsafe {
            *(v.as_ptr().add(i) as *mut T) += 1;
        }
    })
}
```

## Synchronization
- **Locks**
  - **Coarse**: 500x
  - **Fine**: 5x
- **Atomics**
  - **Relaxed**: 1x
  - **SeqCst**: 2x

## Unsafe
- **unchecked**: 1x
- **with check**: 3x

### Synchronization
- **S**: Unique
- **C/S**: Duplicates

### Offsets
- **SC**: Unique
- **C/S**: Duplicates

### Examples
- **(SngInd)**
- **(RngInd)**
- **(AW)**
Programmers face a conundrum expressing irregular accesses

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>S</th>
<th></th>
<th>C/S</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coarse</td>
<td>500x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fine</td>
<td>5x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relaxed</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SeqCst</td>
<td>2x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsafe</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unchecked</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with check</td>
<td>3x</td>
<td>S</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

```rust
check_duplicates(offsets);
(0..n)
.into_par_iter()
.for_each(|i| {
    unsafe {
        *(v.as_ptr().add(i) as *mut T)+=1;
    }
})
```

21
Programmers face a conundrum expressing irregular accesses

Rust solutions for irregular accesses are not fearless
So far, static unordered scheduling
Let’s consider other cases
Task scheduling

**Static:** E.g., bulk-synchronous parallelism (split-merge)

**Dynamic:**
- Tasks spawn new tasks
- Workers pop a task, execute, push new tasks (if exist)

E.g., MultiQueue
Task scheduling

**Static:** E.g., bulk-synchronous parallelism (split-merge)

**Dynamic:**
- Tasks spawn new tasks
- Workers pop a task, execute, push new tasks (if exist)

E.g., MultiQueue

User: depends on access types \( F/C/S \)

Workers:

MQ:
Task scheduling

**Static:** E.g., bulk-synchronous parallelism (split-merge)

**Dynamic:**
- Tasks spawn new tasks
- Workers pop a task, execute, push new tasks (if exist)

E.g., MultiQueue

User: depends on access types

Implementer: Workers and Queue

Workers: MQ: F/C/S

S
Task scheduling

**Static:** E.g., bulk-synchronous parallelism (split-merge)

**Dynamic:**

- Tasks spawn new tasks
- Workers pop a task, execute, push new tasks (if exist)

User: depends on access types

Implementer: Workers and Queue

The dynamism of task scheduling does not affect fearlessness
Evaluation of RPB
Irregular accesses are common in RPB

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Tasks’ Accesses</th>
<th>Task dispatch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Regular</td>
<td>Irregular</td>
</tr>
<tr>
<td></td>
<td>Stride</td>
<td>Block</td>
</tr>
<tr>
<td>bw</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dedup</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dr</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>hist</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>isort</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lrs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mm</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>msf</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sa</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sf</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sort</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bfs</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>sssp</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Irregular accesses are common in RPB

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Regular</th>
<th>Irregular</th>
<th>Task dispatch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RO</td>
<td>Stride</td>
<td>Block</td>
</tr>
<tr>
<td>bw</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dedup</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dr</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>hist</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>isort</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lrs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>msf</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sa</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sf</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sort</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bfs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sssp</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Regular accesses: F
Irregular accesses: C/S
Irregular accesses are common in RPB

Expressing RPB in Rust is not fearless
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses

(24 cores)
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses (24 cores)
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses

(24 cores)
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses

Rust benchmarks with unsafe irregular accesses are 1.44x slower (gmean) than C++

(24 cores)
See paper for more evaluations on ...

Runtime checks with unsafe blocks

Unnecessary synchronization to placate rustc
Conclusions

To replace C++, Rust’s claims should be verified, including fearless concurrency

Rust grants fearless concurrency for regular accesses but not irregular ones

The task scheduling scheme does not affect fearlessness

Using (current) Rust can have a performance penalty

Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, Mark C. Jeffrey
Conclusions

To replace C++, Rust’s claims should be verified, including fearless concurrency

Rust grants fearless concurrency for regular accesses but not irregular ones

The task scheduling scheme does not affect fearlessness

Using (current) Rust can have a performance penalty