When is Parallelism Fearless and Zero-Cost with Rust?

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

UNIVERSITY OF TORONTO

SPAA 2024

Rust claims to solve the arduous task of parallel programming

The Rust Programming Language : 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

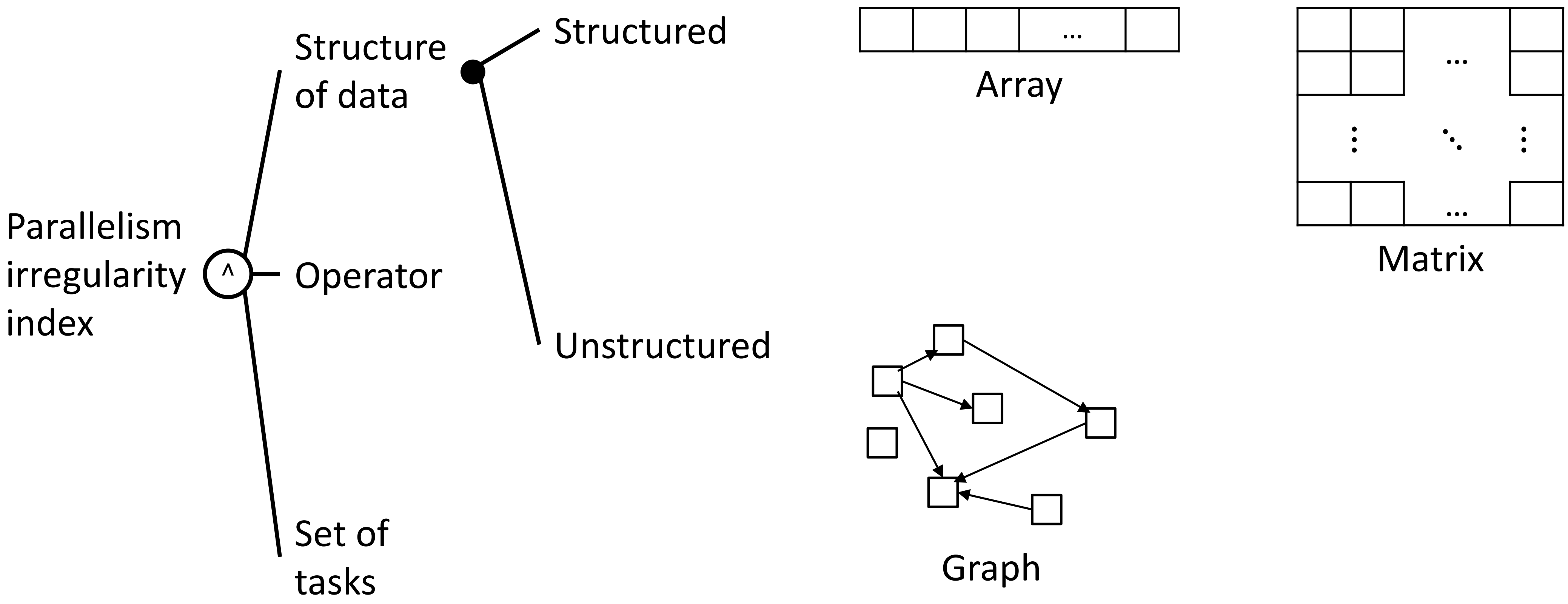
- Performance evaluation of Rust vs. C++ benchmarks

Background

Types of parallelism

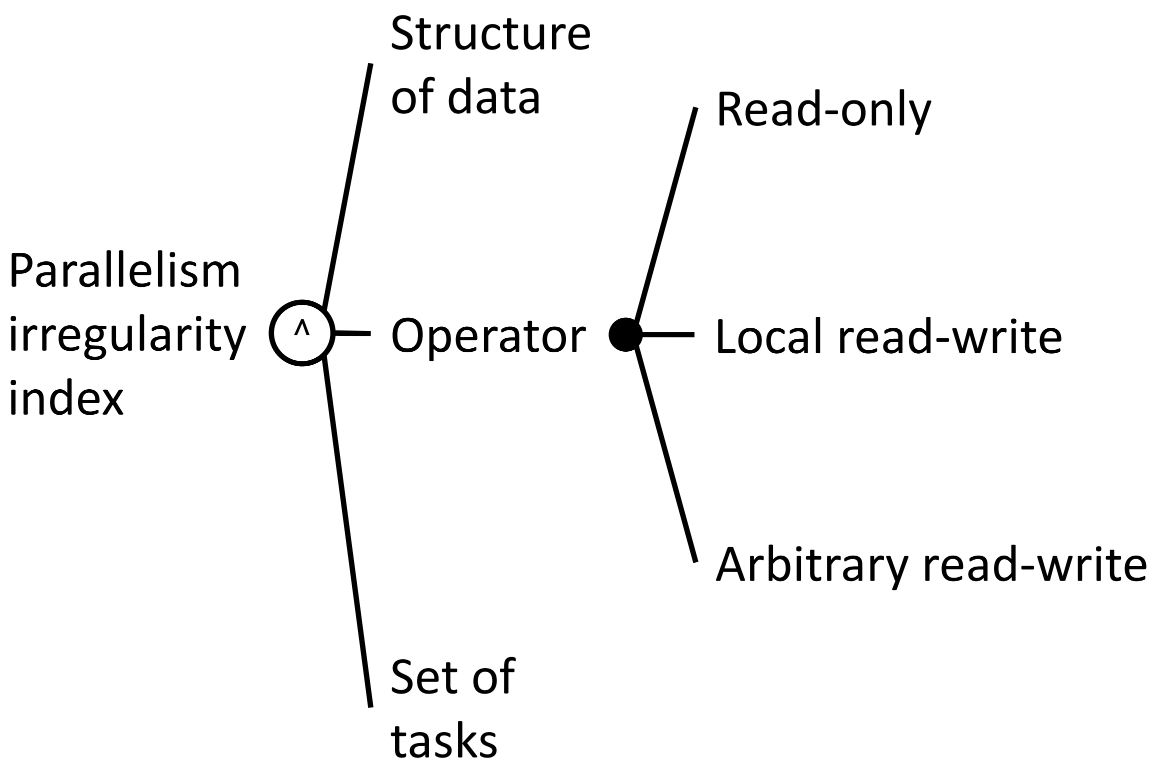
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]



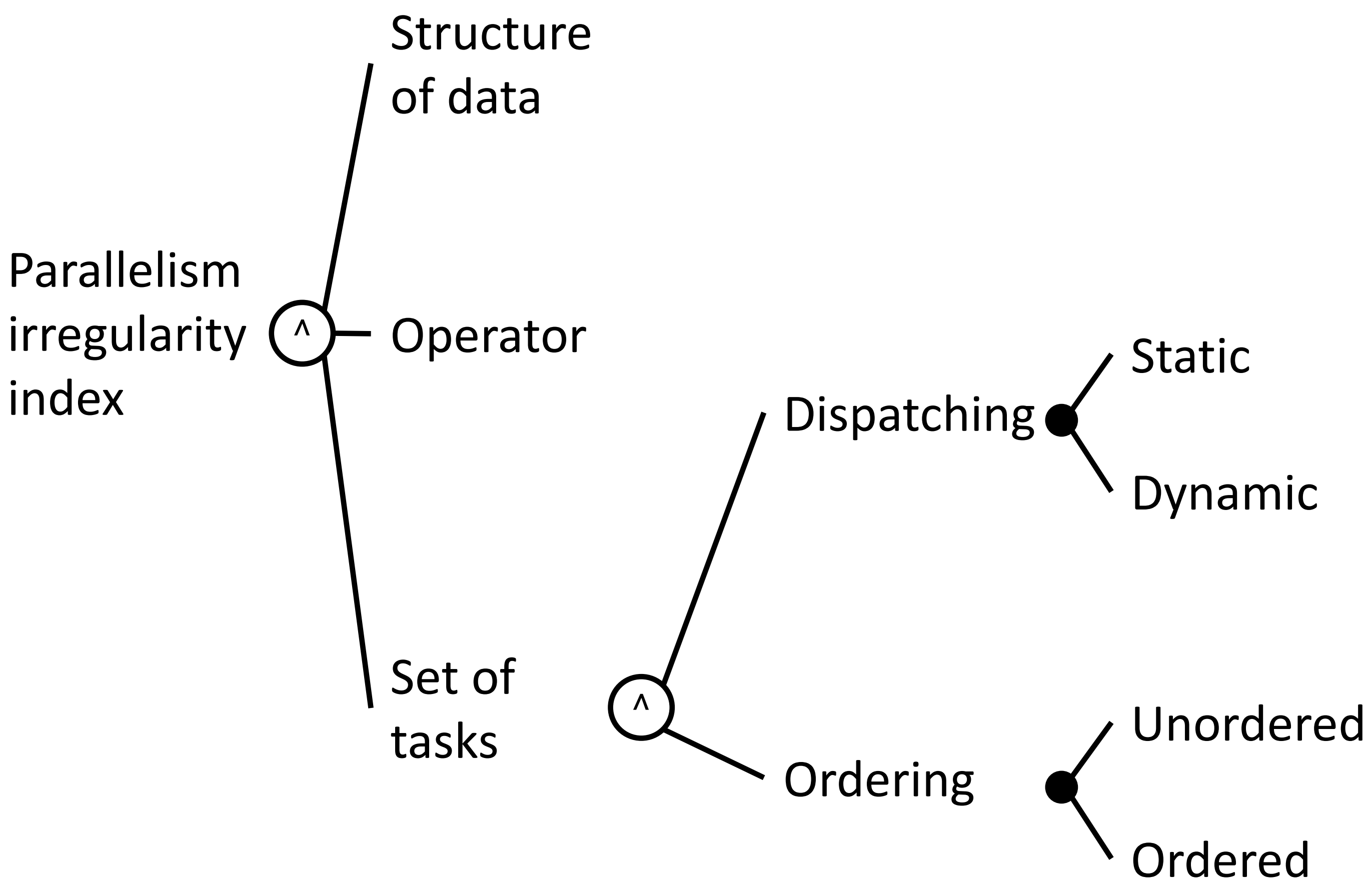
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]



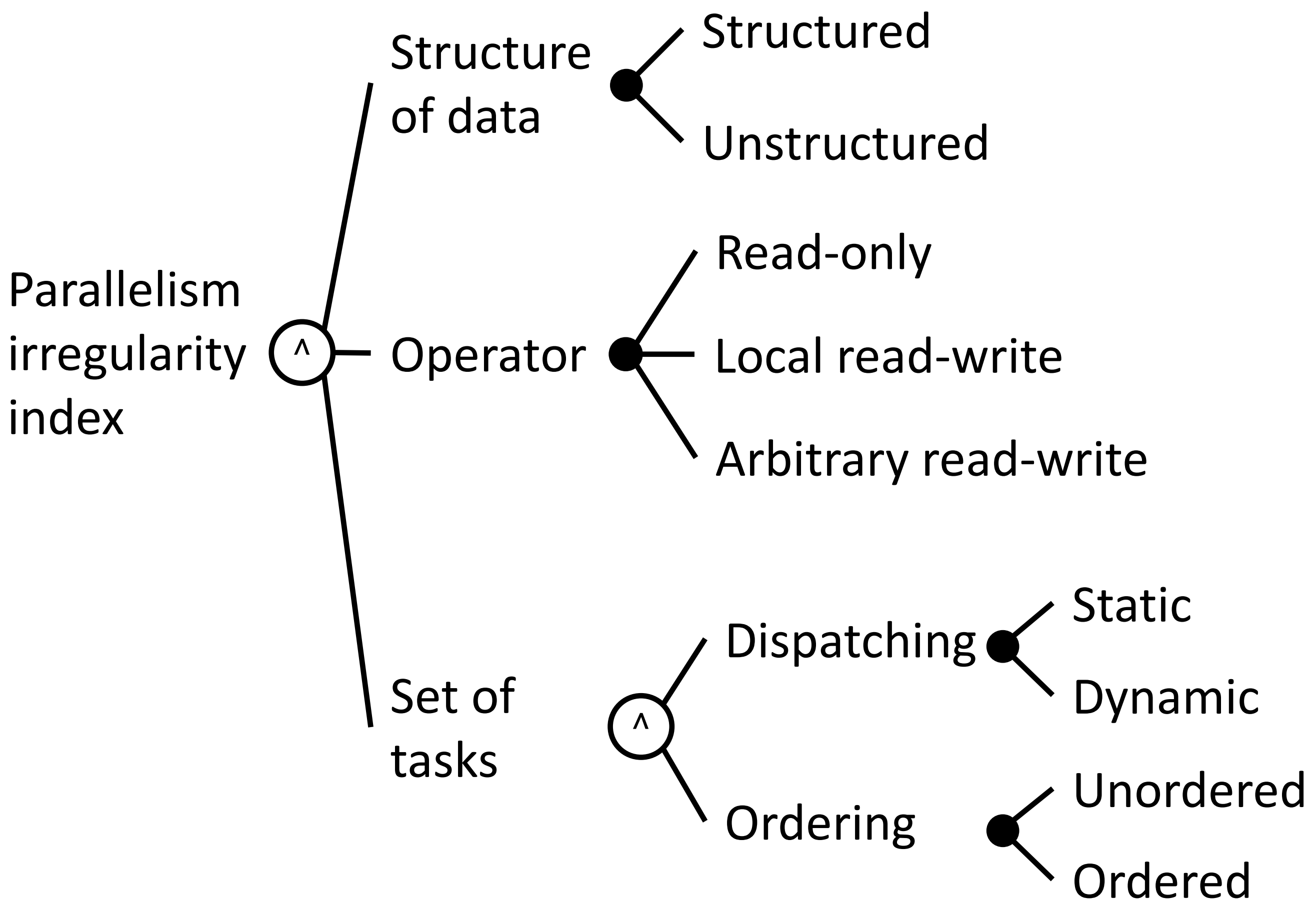
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]



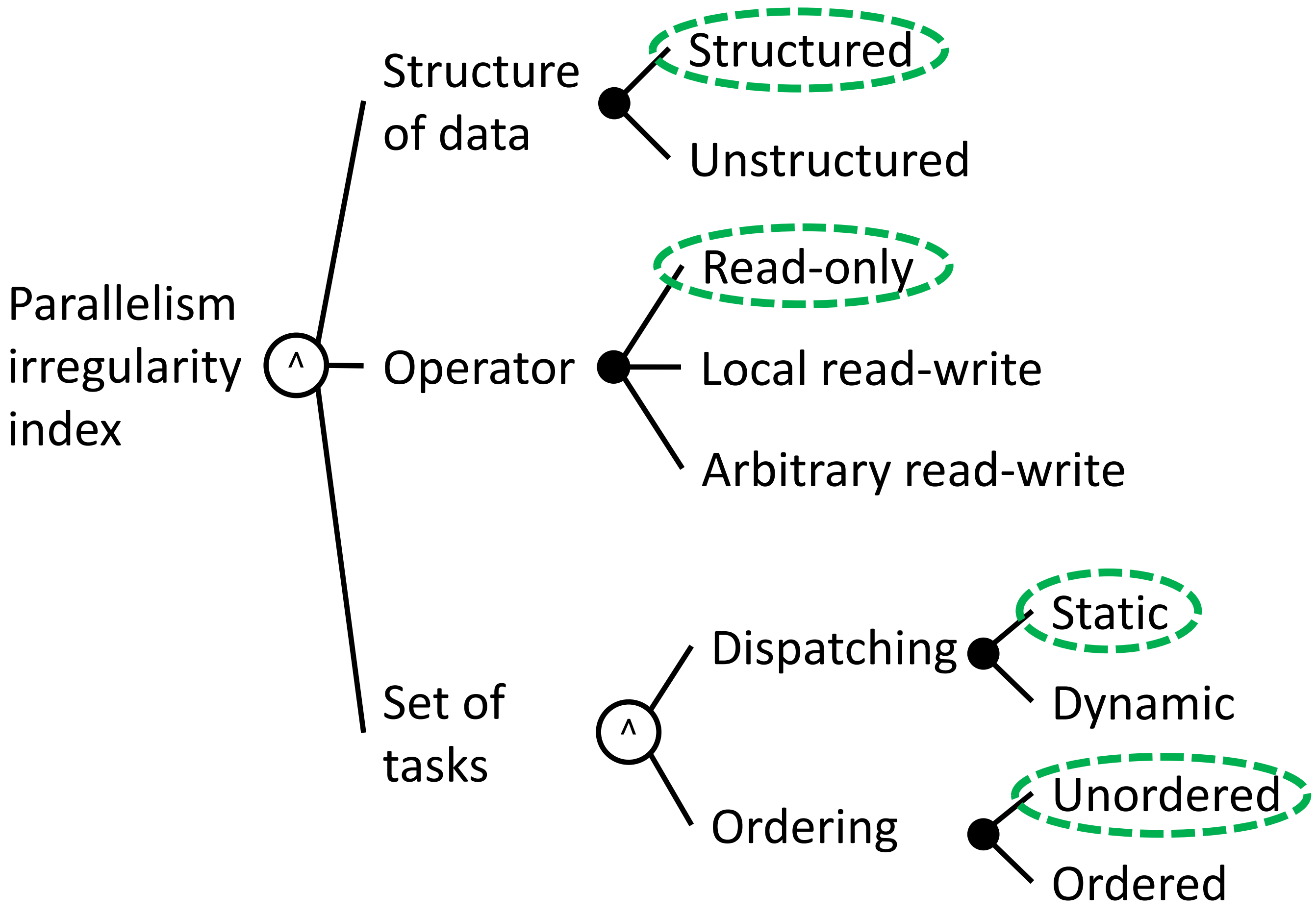
The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]



The regularity of algorithms affects ease of parallelization

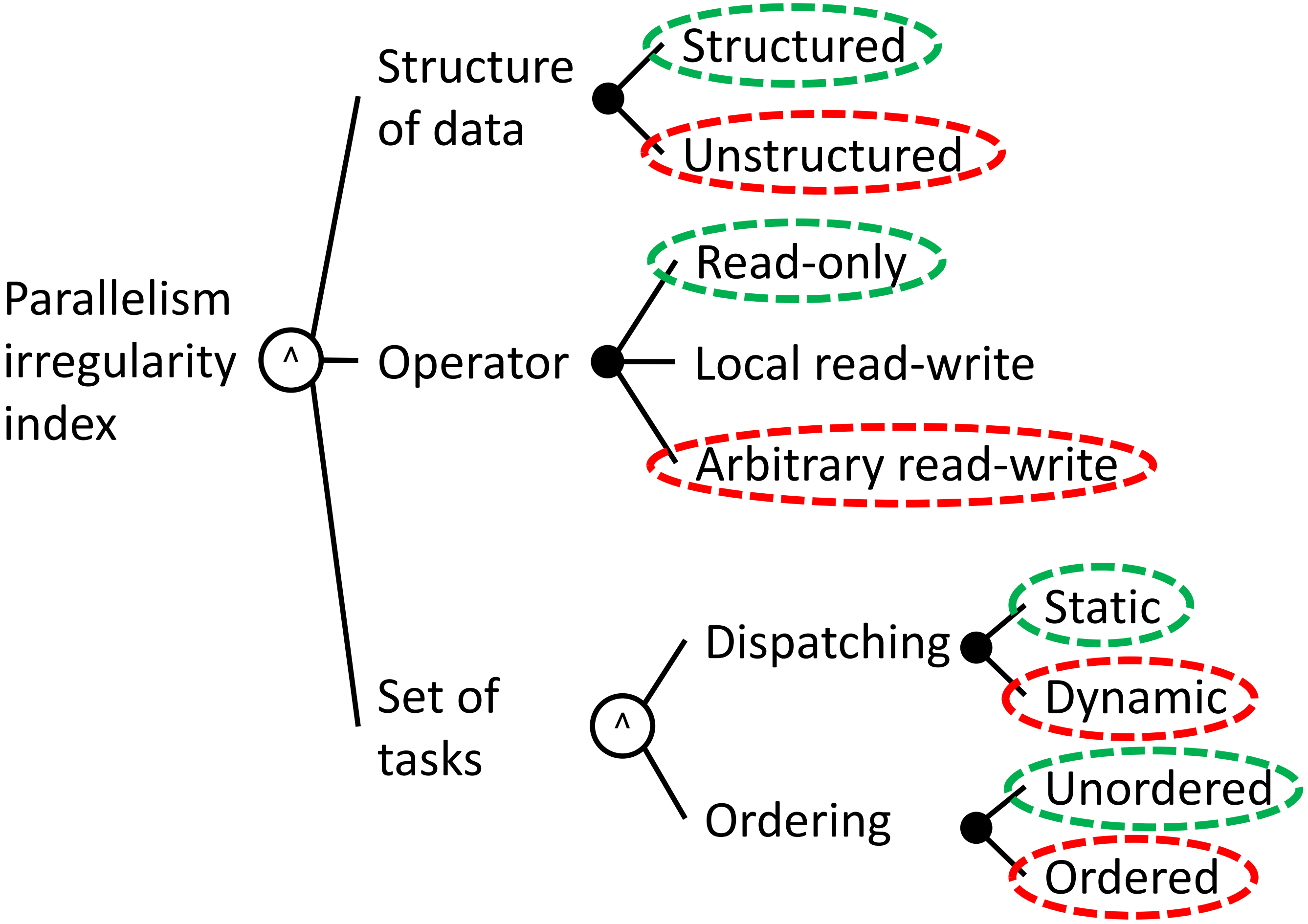
[Pingali et al., PLDI 2011]



Parallel array summation

The regularity of algorithms affects ease of parallelization

[Pingali et al., PLDI 2011]



Parallel array summation

Delta-stepping SSSP

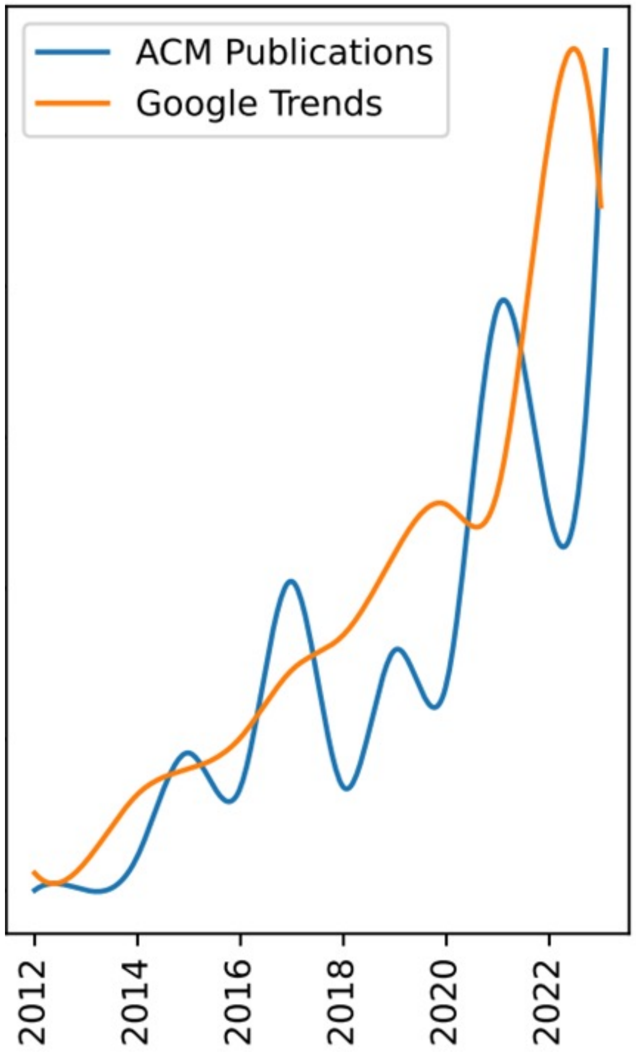
Background

Rust

Rust is gaining popularity

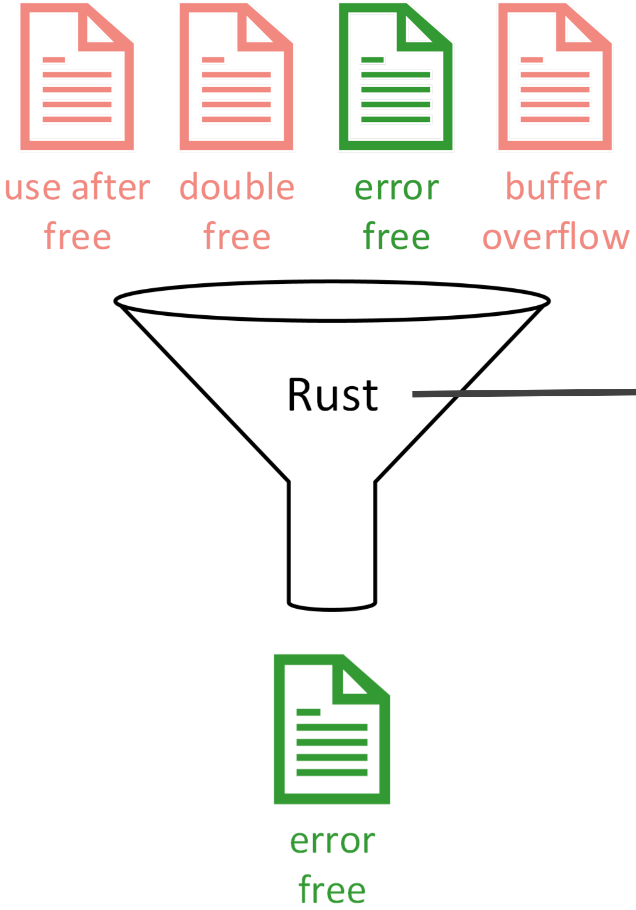
2022
Developer
Survey

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

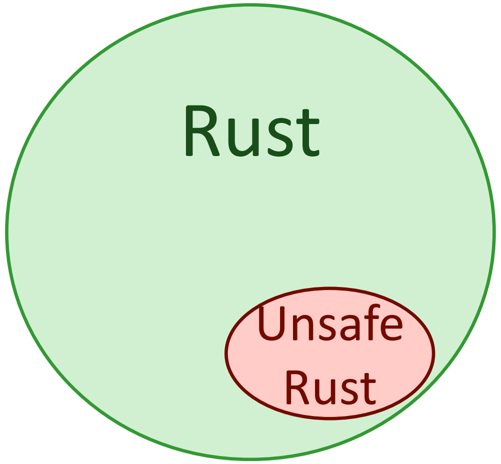


Rust is gaining popularity

Rust is gaining popularity because of its safety guarantees

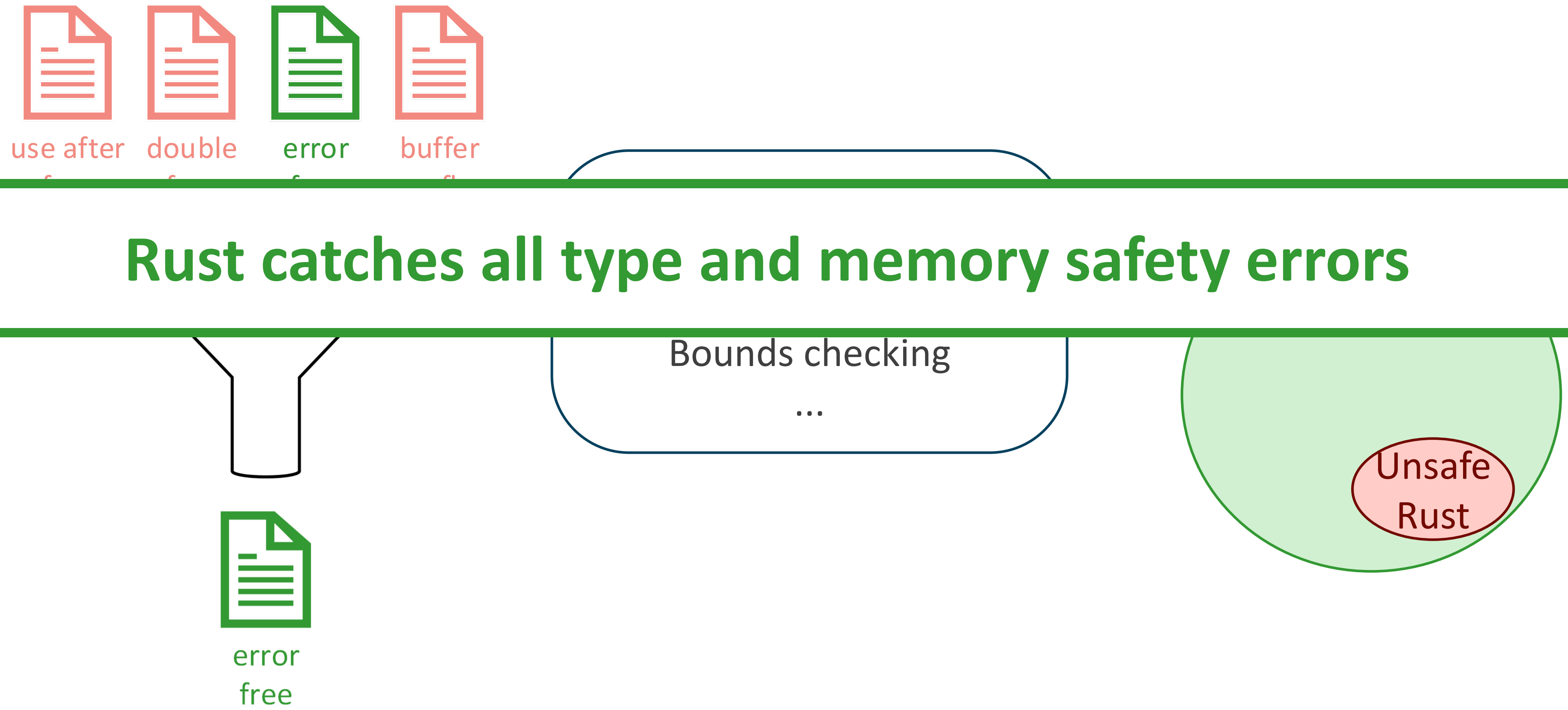


Ownership
Aliasing XOR Mutability[2]
Lifetime tracking
Bounds checking
...



[2] Yanovski et al., ICFP'21, GhostCell: separating permissions from data in Rust.

Rust is gaining popularity because of its safety guarantees



Rust catches all type and memory safety errors

[2] Yanovski et al., ICFP'21, GhostCell: separating permissions from data in Rust.

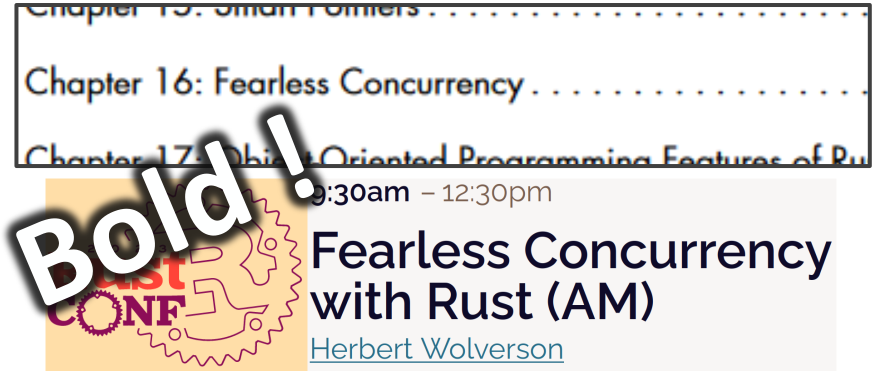
Rust claims to enable *fearless concurrency*!!!

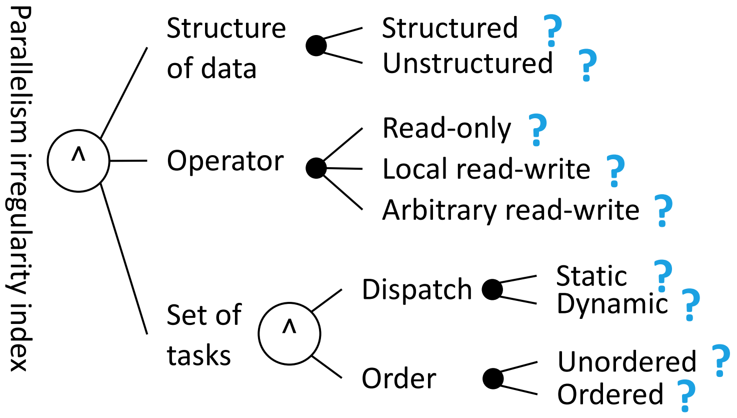
9:30am – 12:30pm

Fearless Concurrency with Rust (AM)

Herbert Wolverson

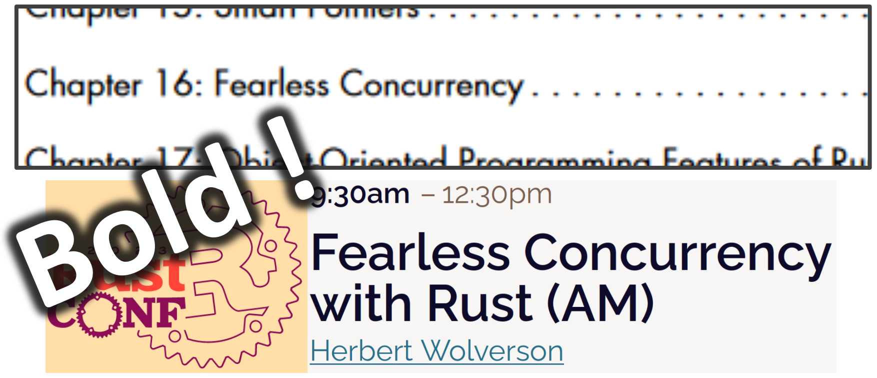
Rust claims to enable *fearless concurrency*!!!

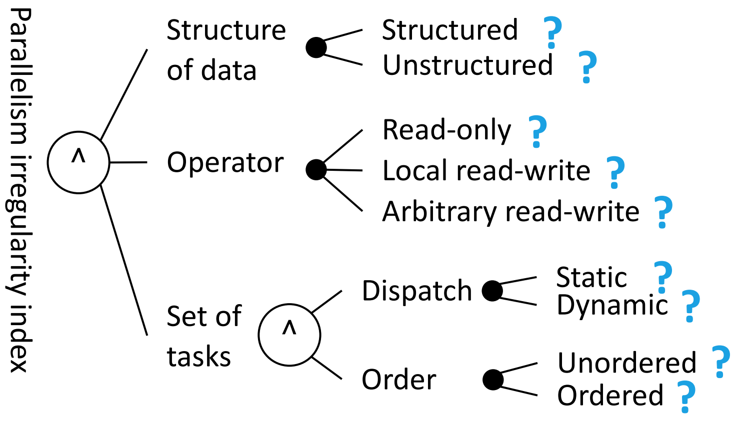


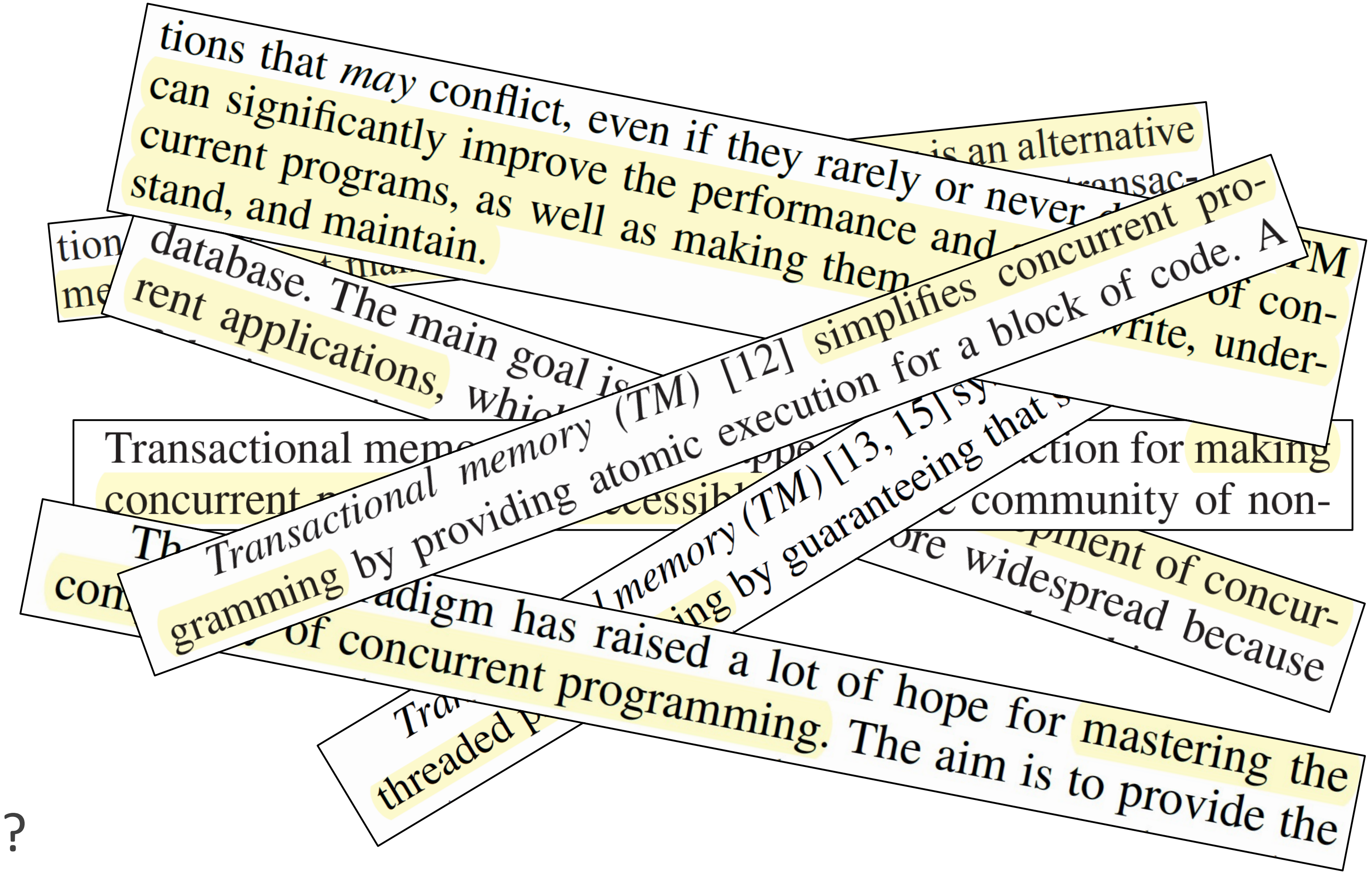


Does Rust cover all these branches?

Rust claims to enable *fearless concurrency*!!!

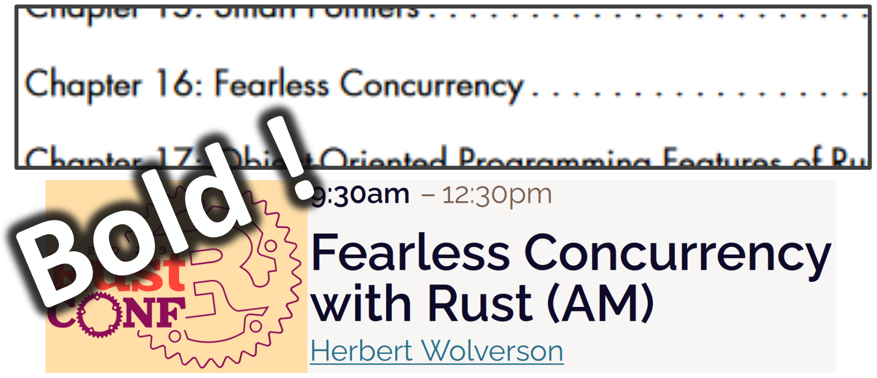


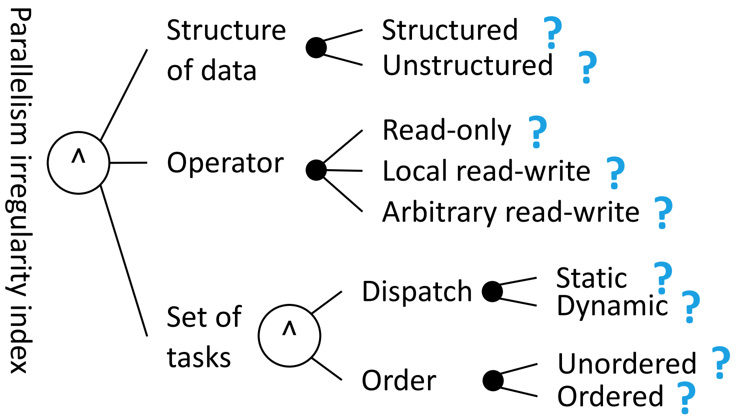




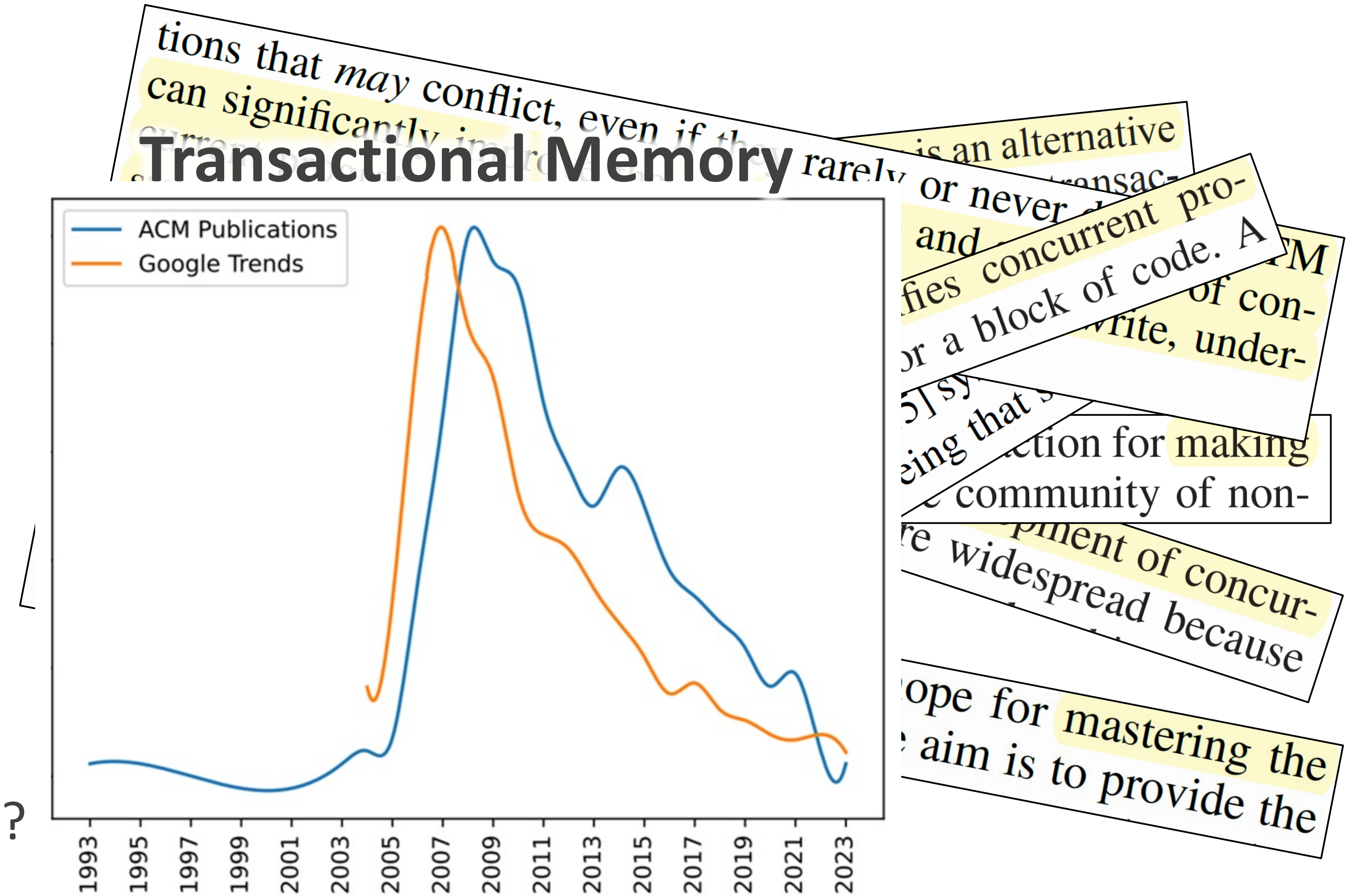
Does Rust cover all these branches?

Rust claims to enable *fearless concurrency*!!!

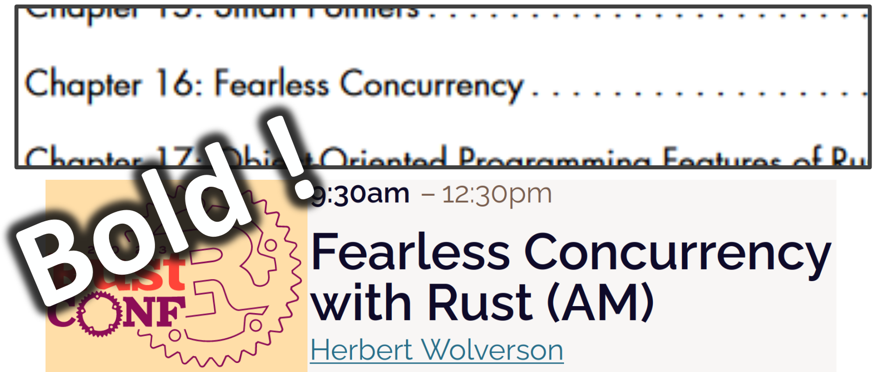


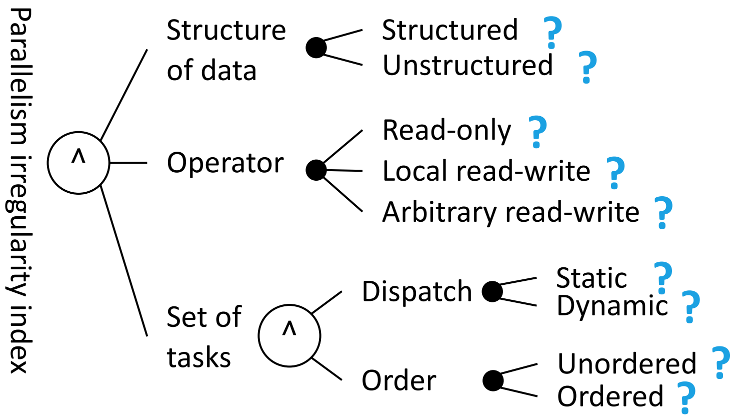


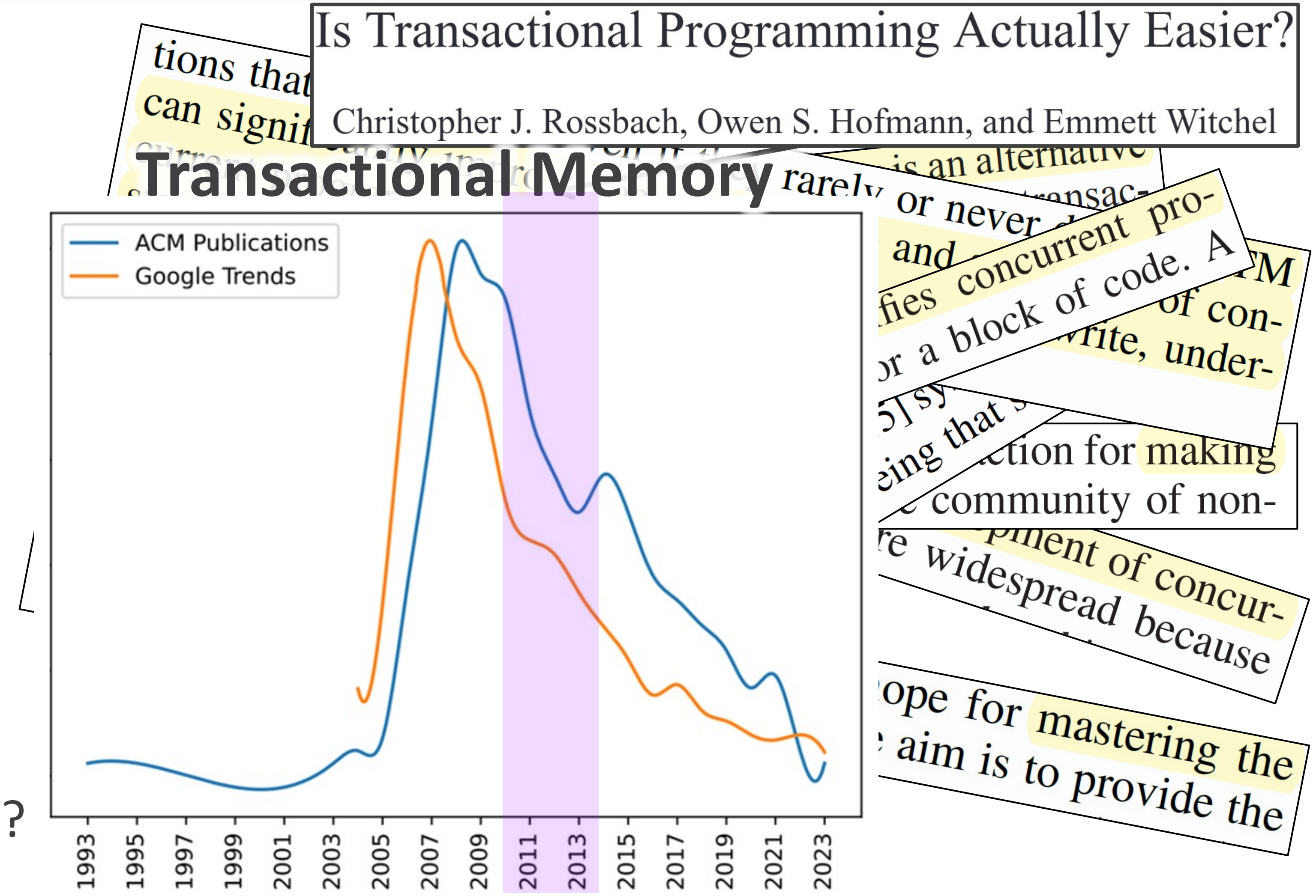
Does Rust cover all these branches?



Rust claims to enable *fearless concurrency*!!!

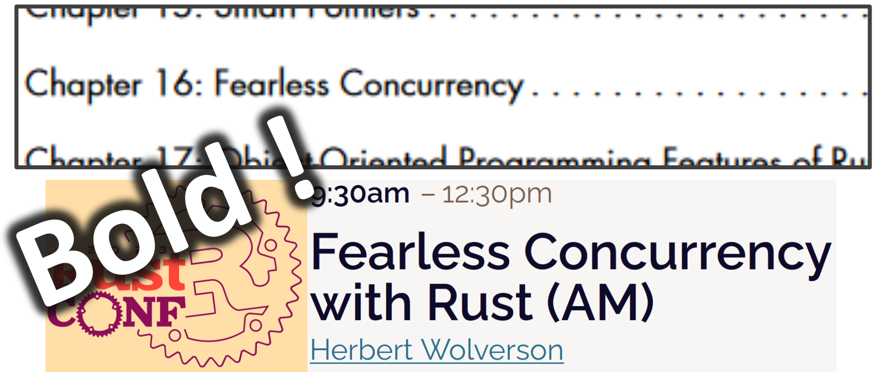


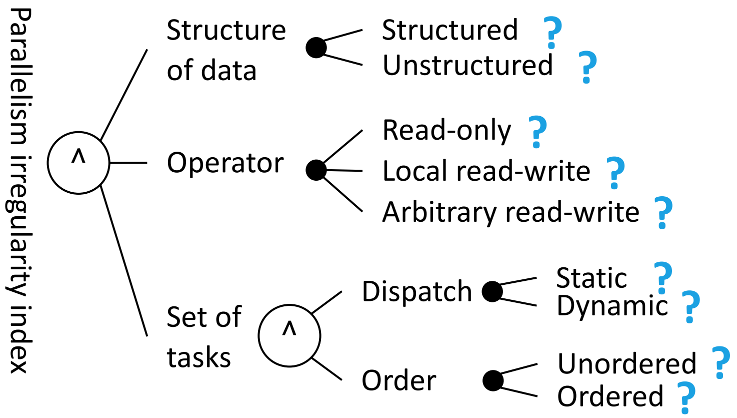




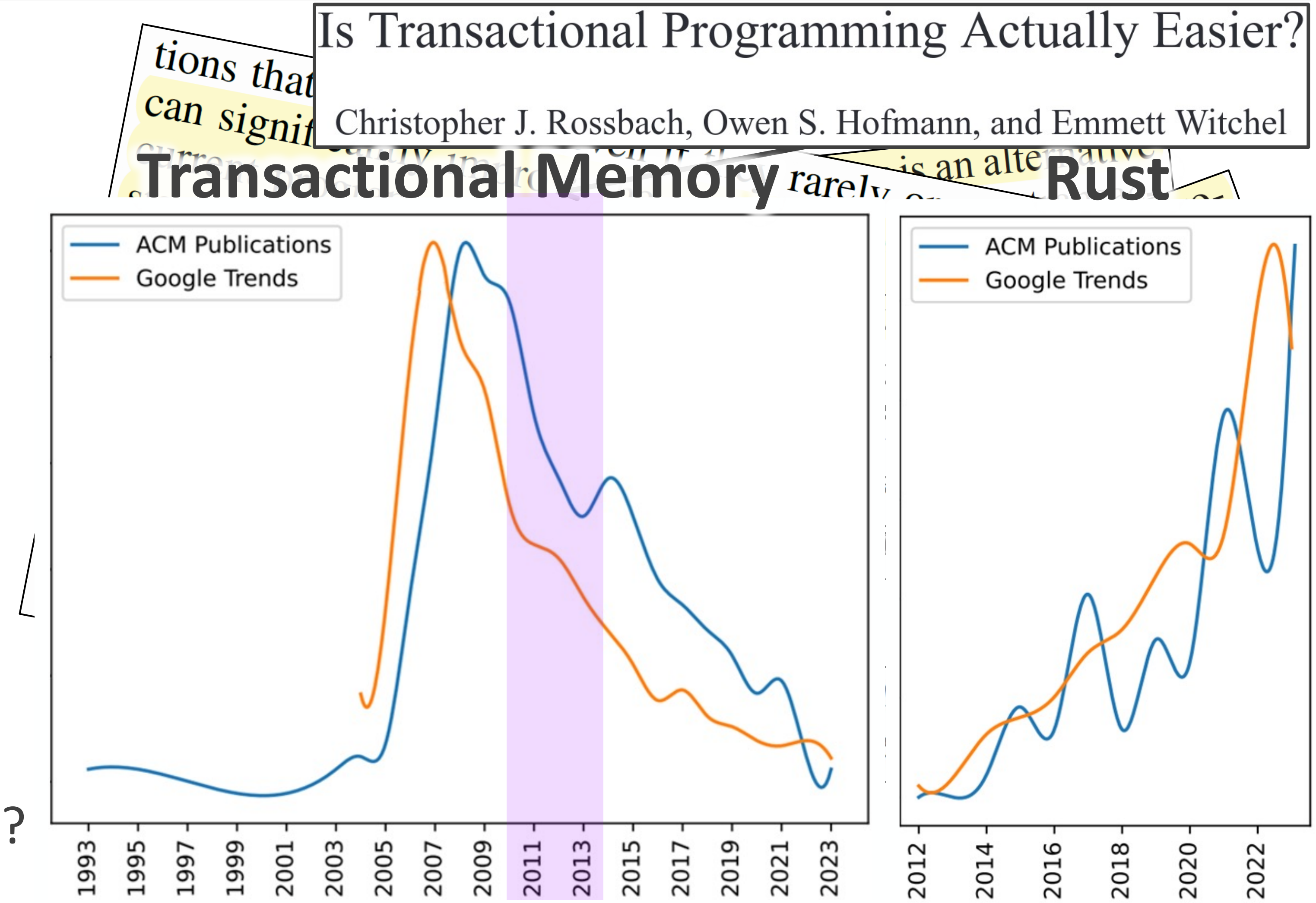
Does Rust cover all these branches?

Rust claims to enable *fearless concurrency*!!!

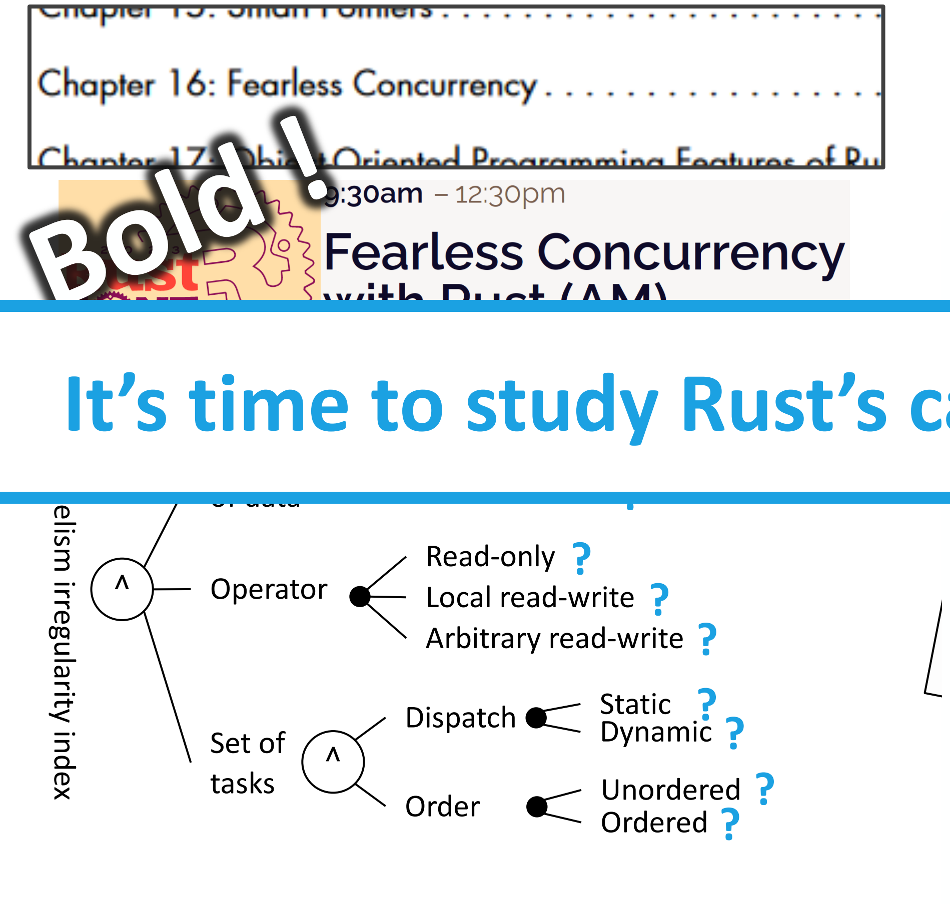


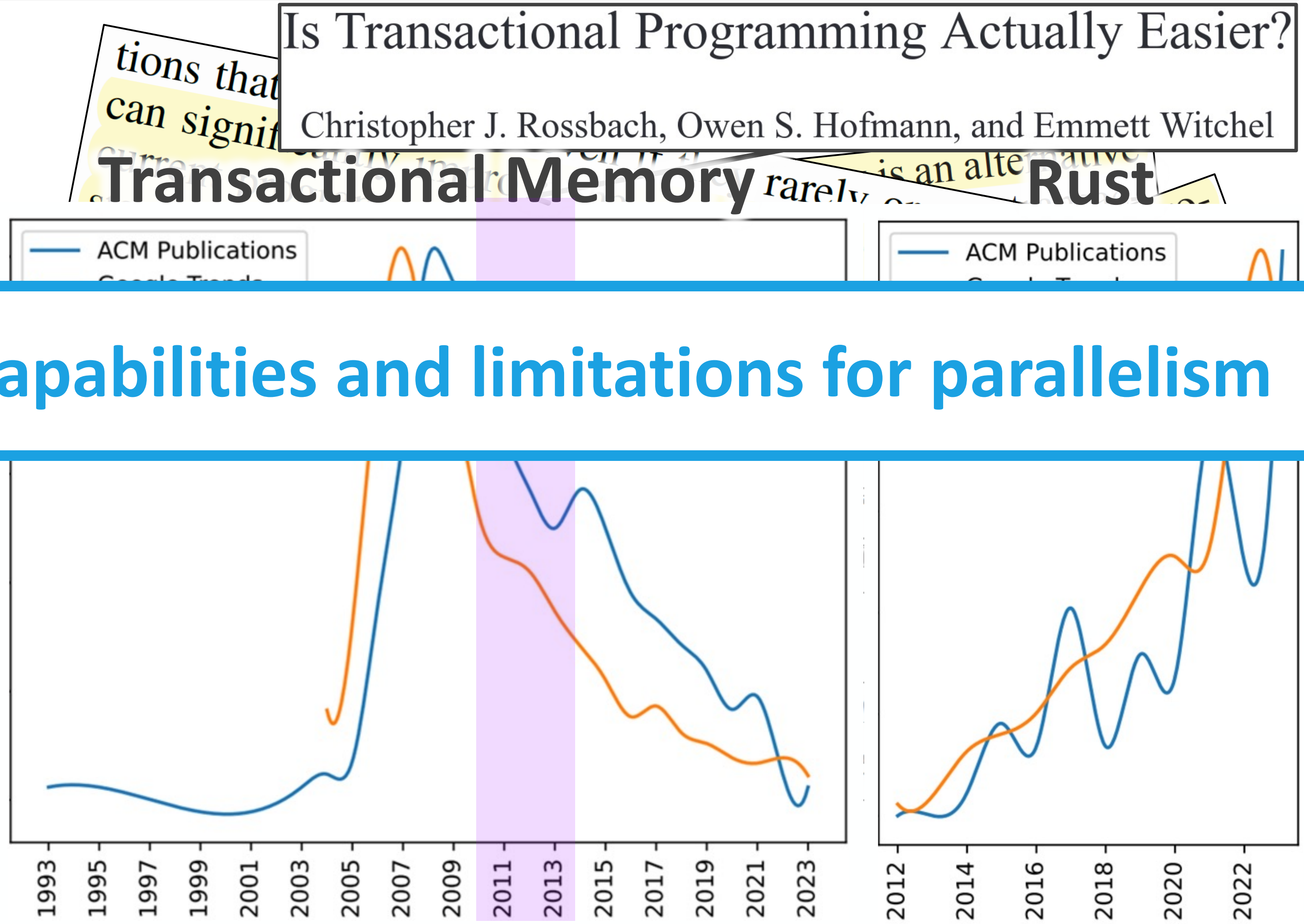


Does Rust cover all these branches?



Rust claims to enable *fearless concurrency*!!!





It's time to study Rust's capabilities and limitations for parallelism

Does Rust cover all these branches?

Case study

Clarifying “fearless concurrency”

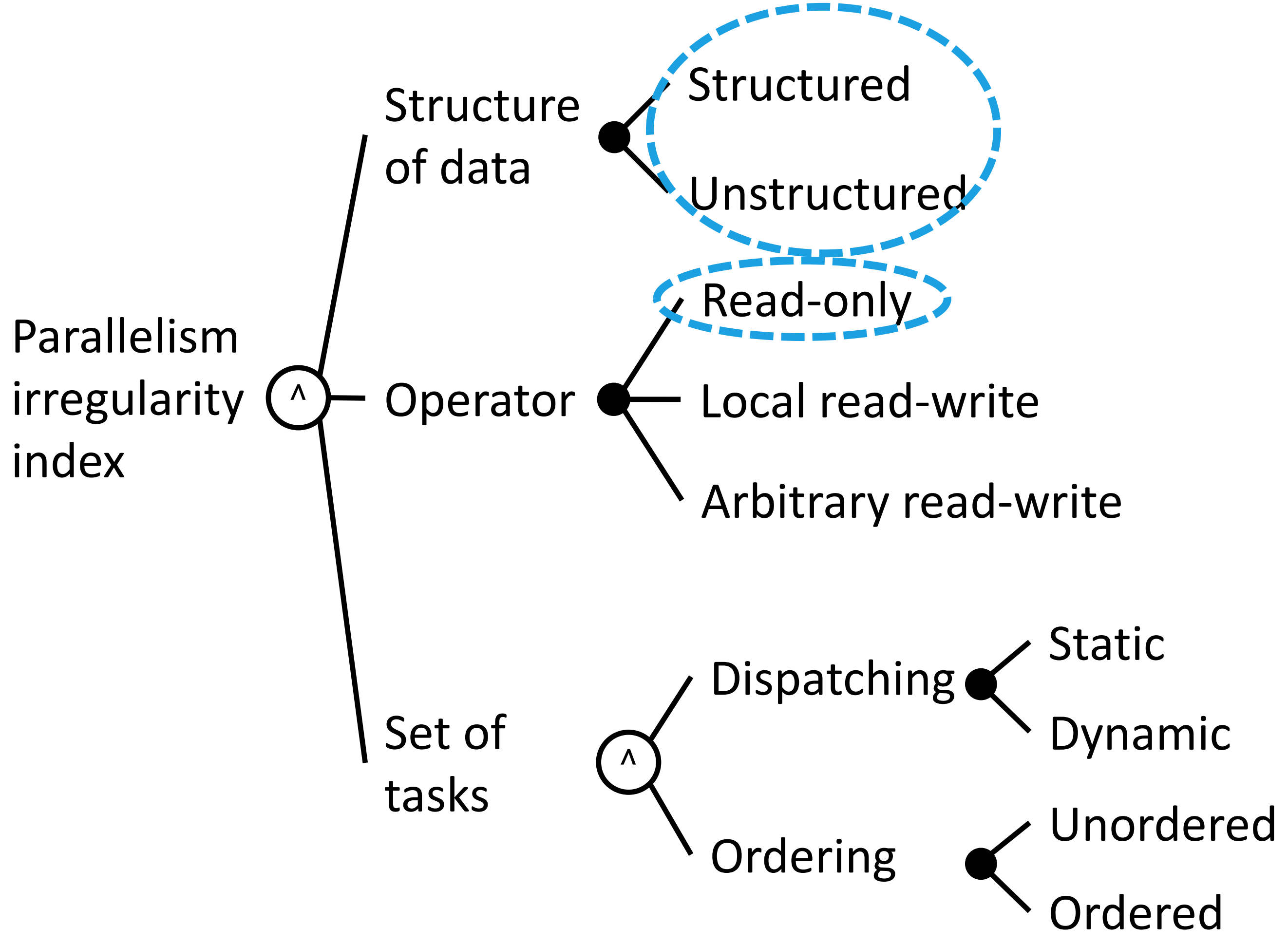
Fear : Anticipation of concurrency errors that manifest at run time.

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:

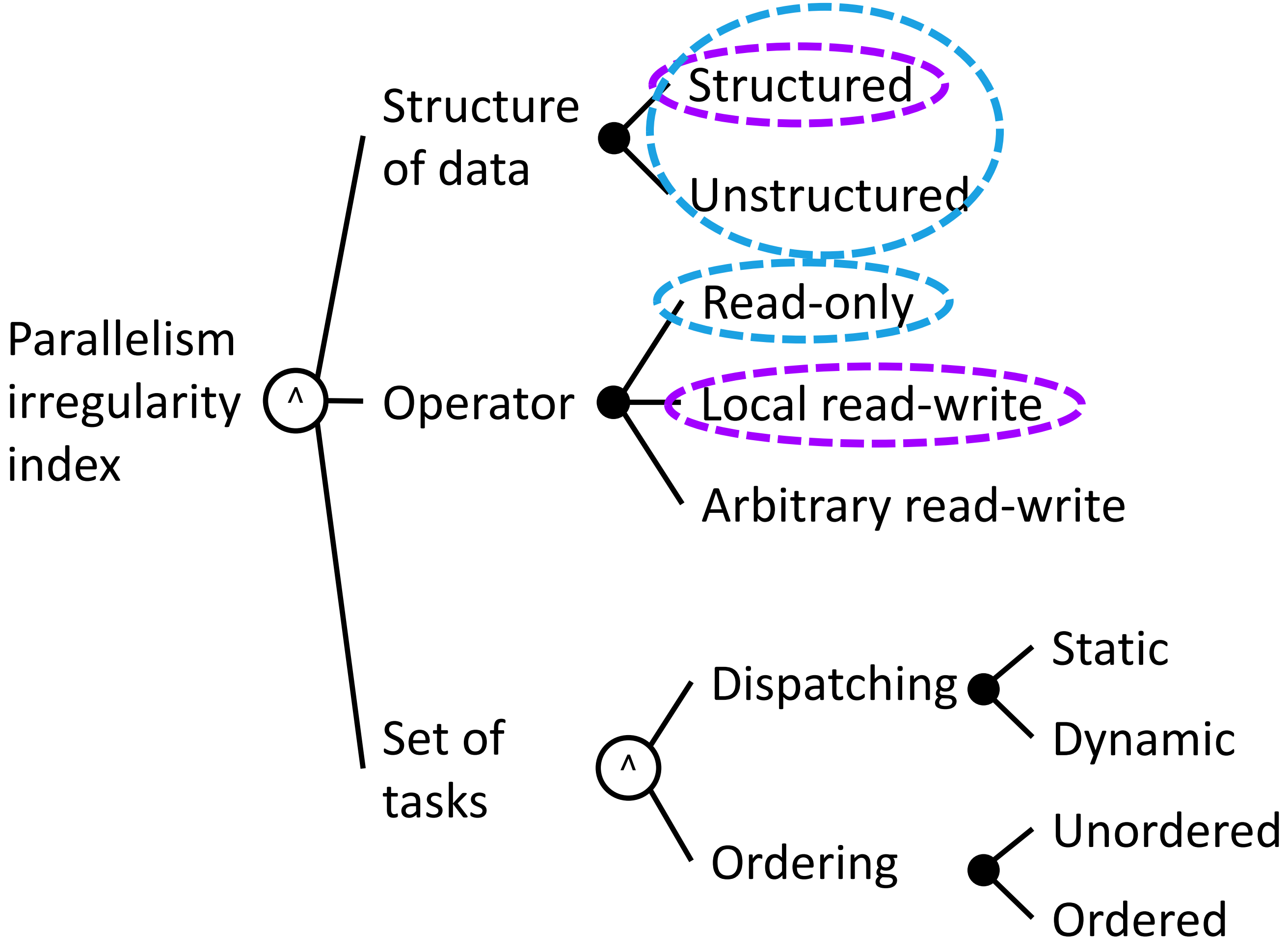
- Read-only accesses



Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:

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

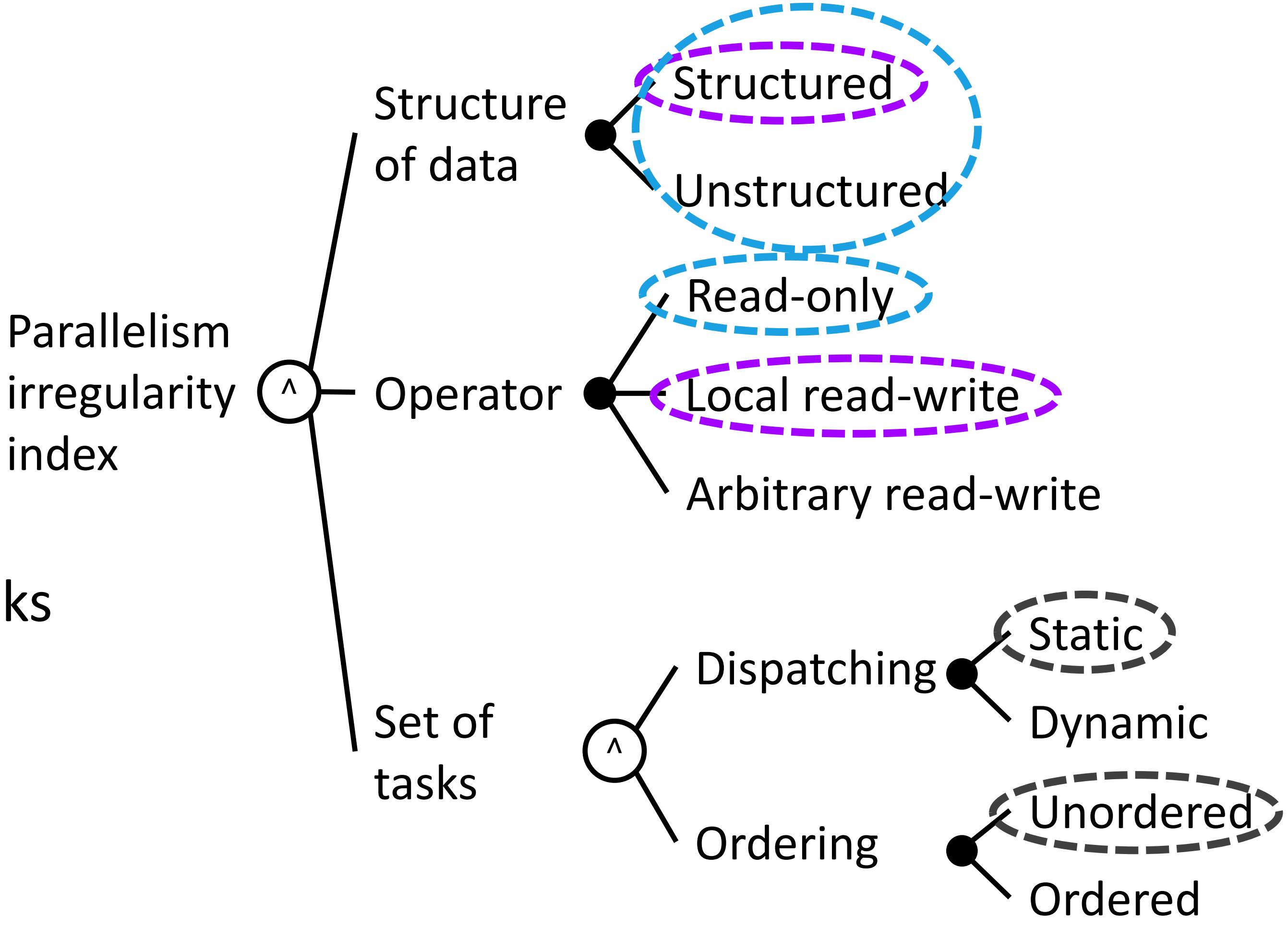


Expressing tasks with regular accesses in Rust (+Rayon)

Regular accesses:

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

Assume a static unordered set of tasks



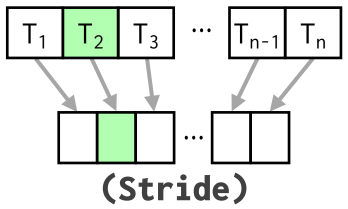
Expressing tasks with regular accesses in Rust (+Rayon)

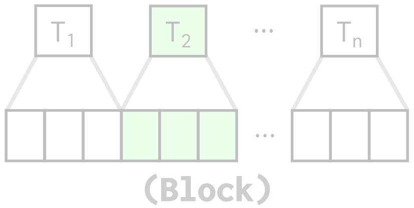
Regular accesses:

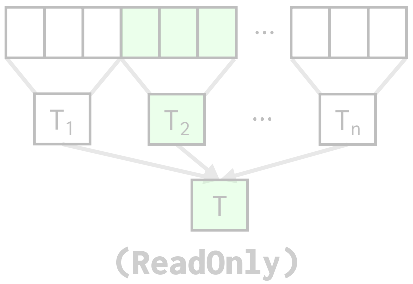
- Read-only accesses
- Local read-writes on structured data

```
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)

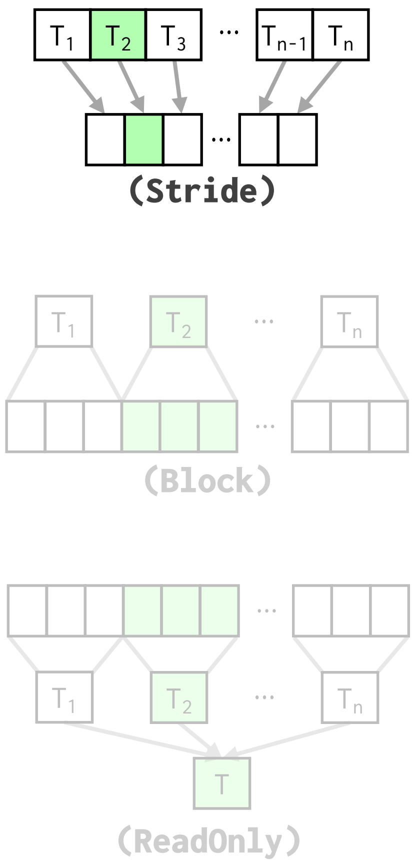
Regular accesses:

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

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

stride pattern on **v**

task



Expressing tasks with regular accesses in Rust (+Rayon)

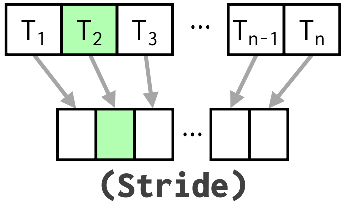
Regular accesses:

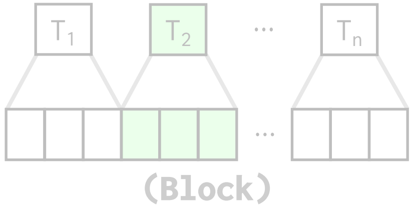
- Read-only accesses
- Local read-writes on structured data

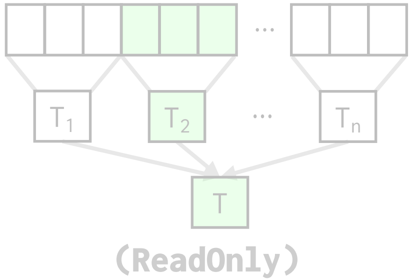
```
fn par_increment(v: &mut [u32])
{
  v.par_iter_mut()
   .for_each(|vi|  *vi+=1);
}
```

stride pattern on **v**

task : Can only mutate vi







Expressing tasks with regular accesses in Rust (+Rayon)

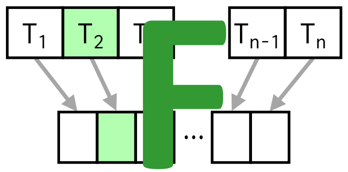
Regular accesses:

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

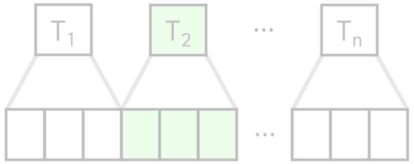
```
fn par_increment(v: &mut [u32])
{
  v.par_iter_mut()
   .for_each(|vi|  *vi+=1);
}
```

stride pattern on **v**

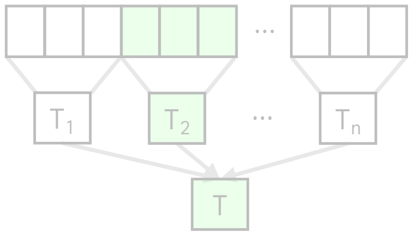
task : Can only mutate vi ⇒ **No errors**



(Stride)



(Block)



(ReadOnly)

Expressing tasks with regular accesses in Rust (+Rayon)

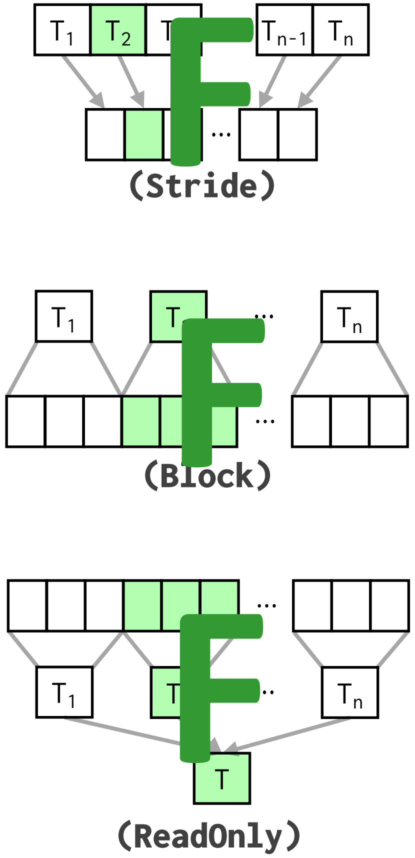
Regular accesses:

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

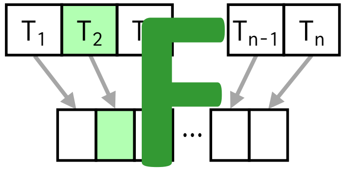
```
fn par_increment(v: &mut [u32])
{
  v.par_iter_mut()
   .for_each(|vi|  *vi+=1);
}
```

stride pattern on **v**

task : Can only mutate vi ⇒ **No errors**



Expressing tasks with regular accesses in Rust (+Rayon)



(Stride)

Regular accesses:

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

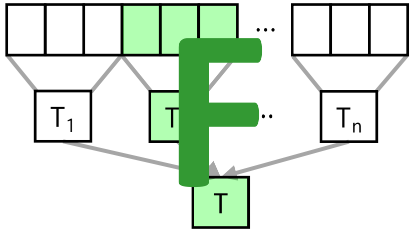
Rust provides fearlessness for regular accesses

(Block)

```
{
  v.par_iter_mut()
   .for_each(|vi|  *vi+=1);
}
```

stride pattern on **v**

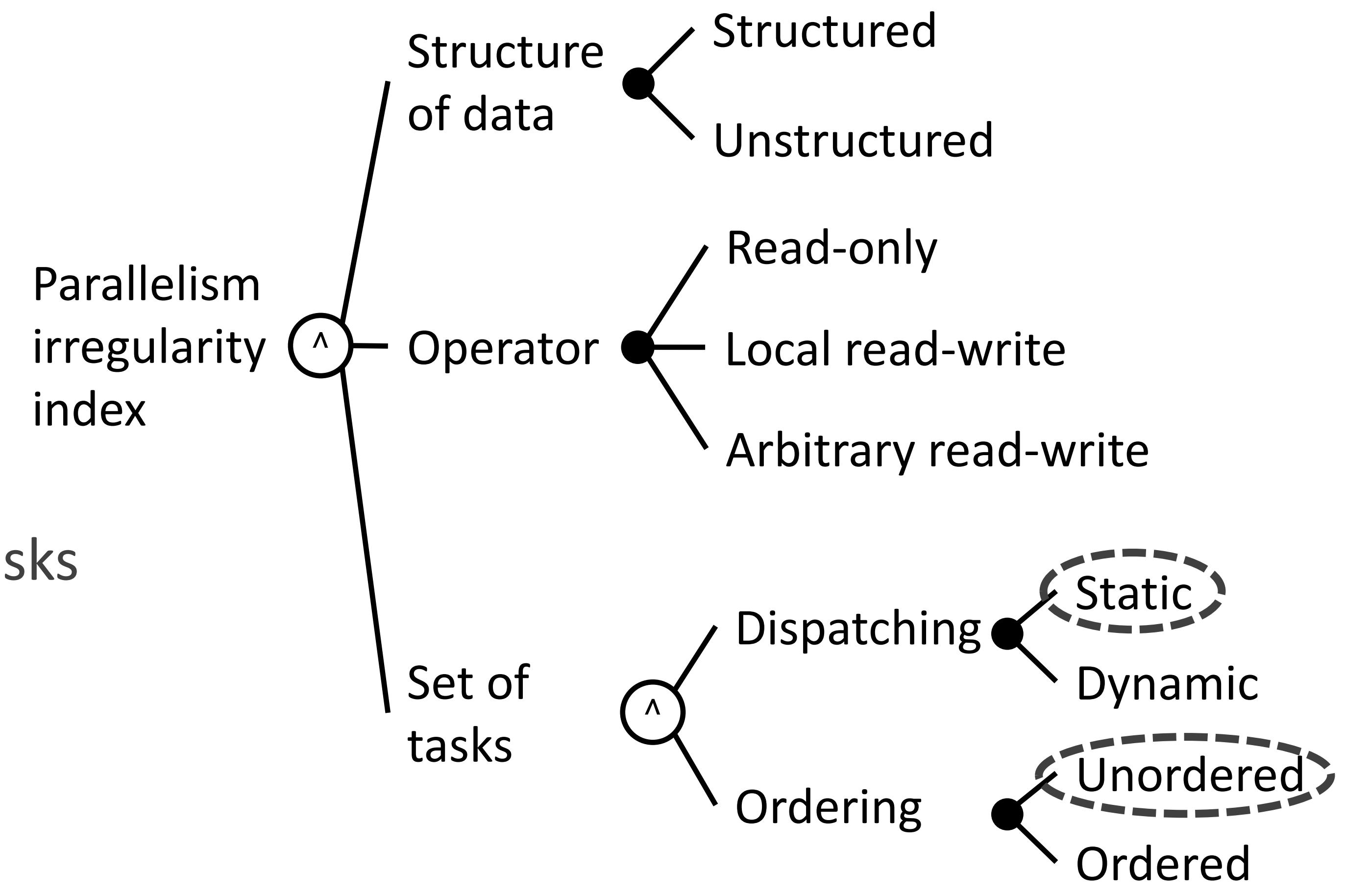
task : Can only mutate vi ⇒ **No errors**



(ReadOnly)

Rust rejects data-race-prone irregular accesses

Irregular accesses:



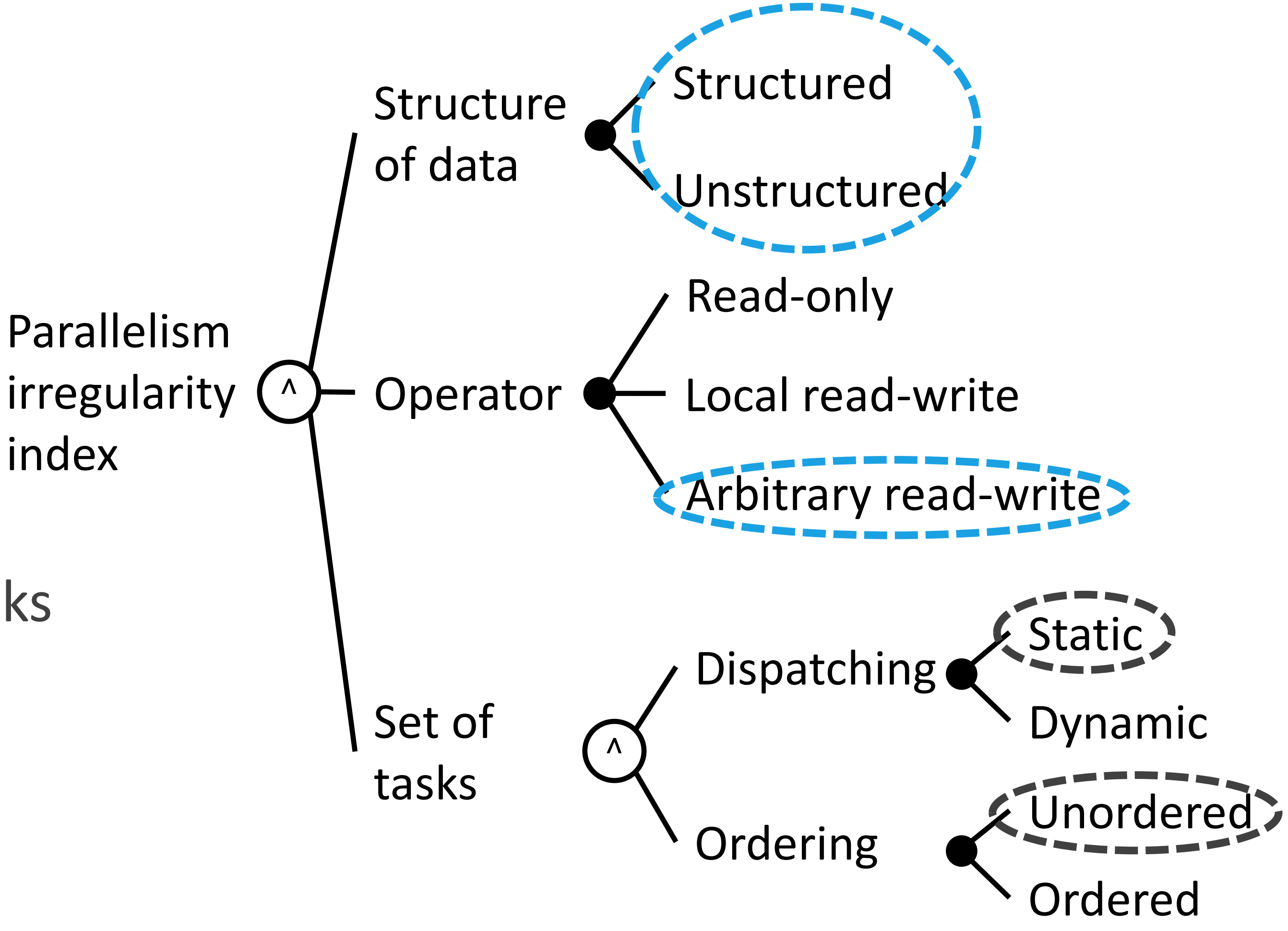
Assume a static unordered set of tasks

Rust rejects data-race-prone irregular accesses

Irregular accesses:

- Arbitrary read-writes

Assume a static unordered set of tasks

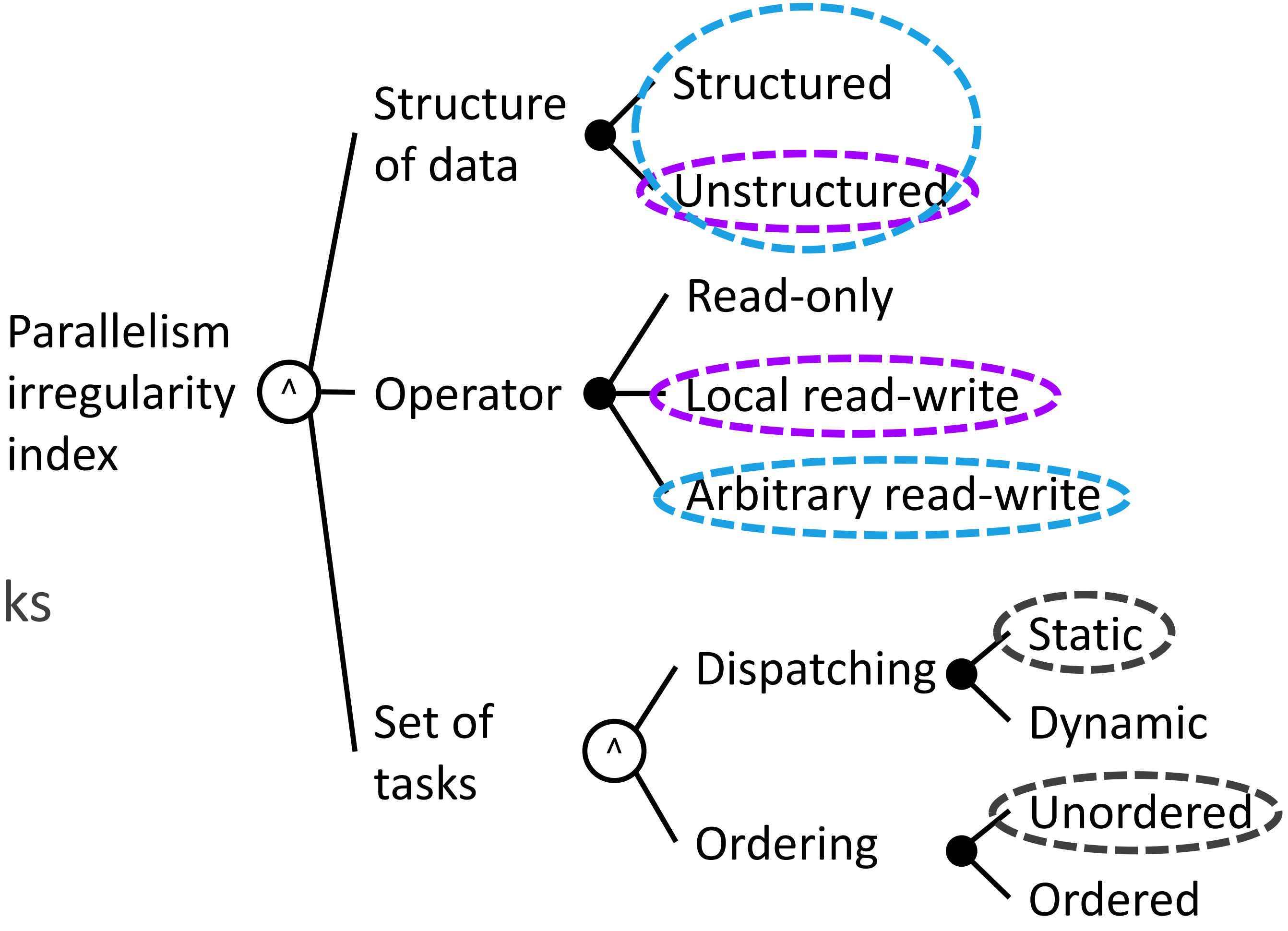


Rust rejects data-race-prone irregular accesses

Irregular accesses:

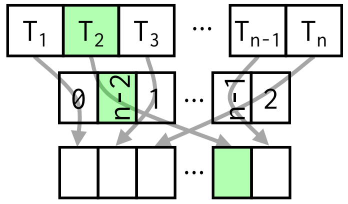
- Arbitrary read-writes
- Local read-writes on unstructured data

Assume a static unordered set of tasks

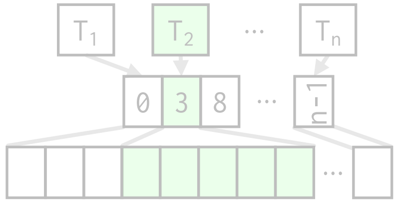


Rust rejects data-race-prone irregular accesses

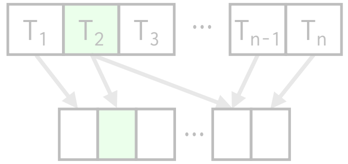
```
// serial indirect increment
(0..n)
  .into_iter()
  .for_each(|i| {
    v[offsets[i]] += 1;
  })
```



(SngInd)



(RngInd)

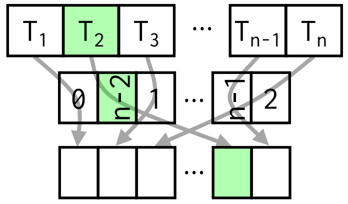


(AW)

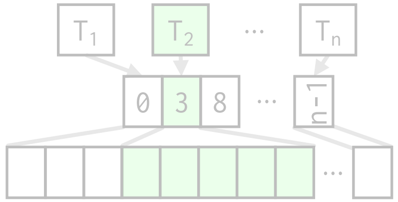
Rust rejects data-race-prone irregular accesses

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

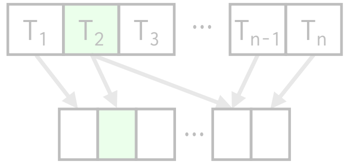
```rust
// parallel
(0..n)
  .into_par_iter()
  .for_each(|i| {
    v[offsets[i]] += 1;
  })
```



(SngInd)



(RngInd)



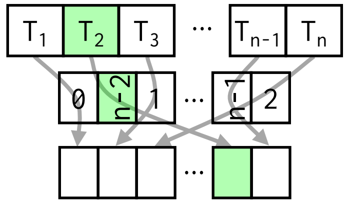
(AW)

Rust rejects data-race-prone irregular accesses

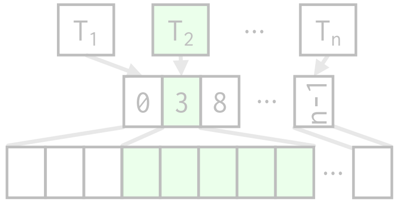
```
// serial indirect increment
(0..n)
  .into_iter()
  .for_each(|i| {
    v[offsets[i]] += 1;
  })
```

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

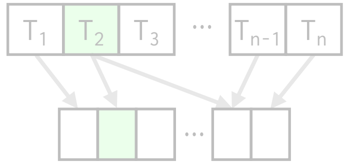
Risky



(SngInd)



(RngInd)

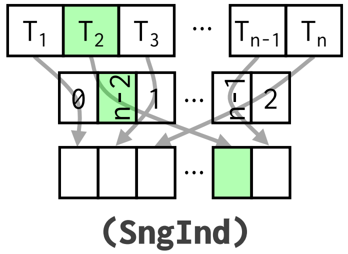


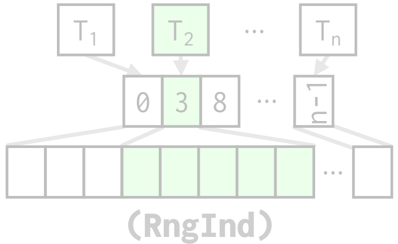
(AW)

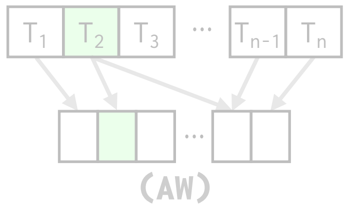
Rust rejects data-race-prone irregular accesses

```
// 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;
  })
```







~~Risky~~

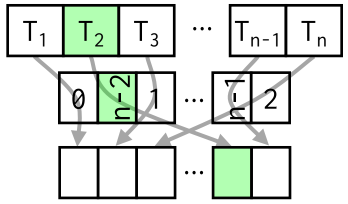
Compile error

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

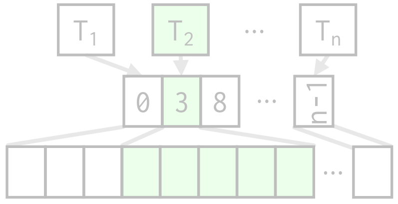
Rust rejects data-race-prone irregular accesses

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

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



(SngInd)



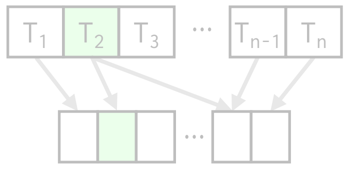
(RngInd)

~~Risky~~

Compile error

Offsets { Unique / Duplicates

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

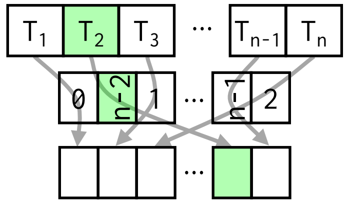


(AW)

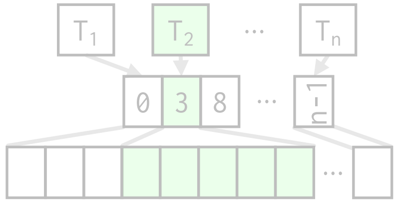
Programmers face a conundrum expressing irregular accesses

Synchronization

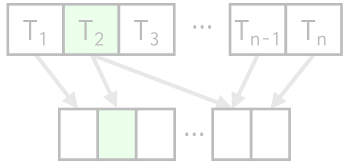
```
(0..n)
   .into_par_iter()
   .for_each(|i| {



   })
```



(SngInd)



(RngInd)



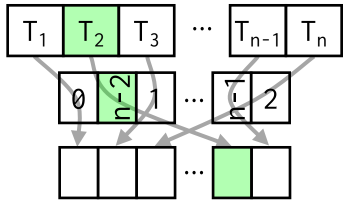
(AW)

Programmers face a conundrum expressing irregular accesses

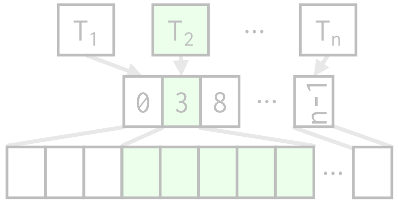
Synchronization

- Locks
  - Coarse 500x

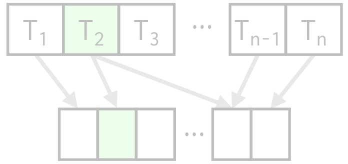
```
(0..n)
  .into_par_iter()
  .for_each(|i| {

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

  })
```



(SngInd)



(RngInd)

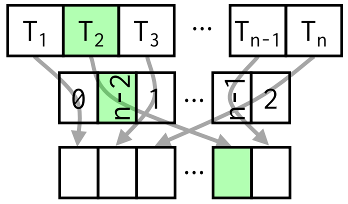


(AW)

Programmers face a conundrum expressing irregular accesses

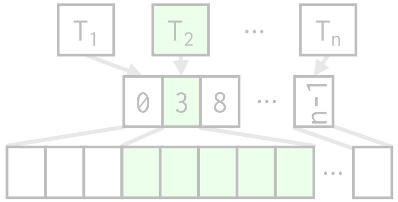
Synchronization

- Locks
  - Coarse 500x
  - Fine 5x

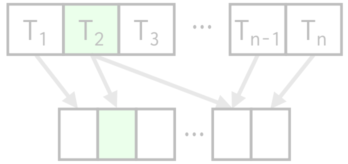


(SngInd)

```
(0..n)
  .into_par_iter()
  .for_each(|i| {

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

  })
```



(RngInd)

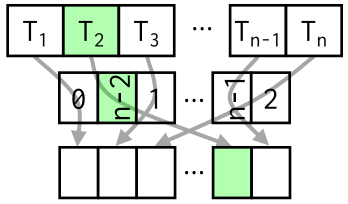


(AW)

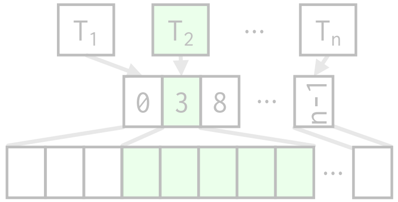
Programmers face a conundrum expressing irregular accesses

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

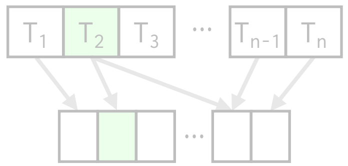
```
(0..n)
  .into_par_iter()
  .for_each(|i| {

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

  })
```



(SngInd)



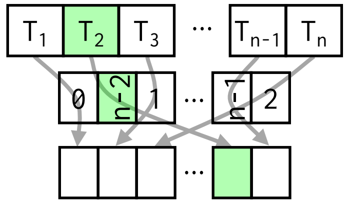
(RngInd)



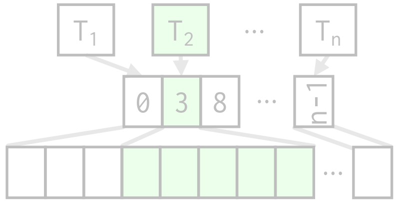
(AW)

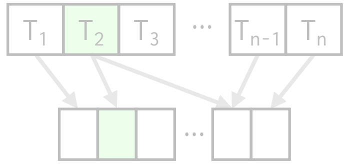
Programmers face a conundrum expressing irregular accesses

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




```
(0..n)
  .into_par_iter()
  .for_each(|i| {

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

  })
```



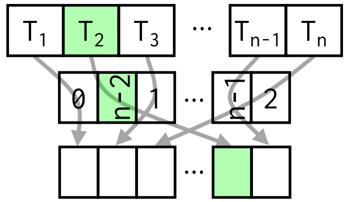


Programmers face a conundrum expressing irregular accesses

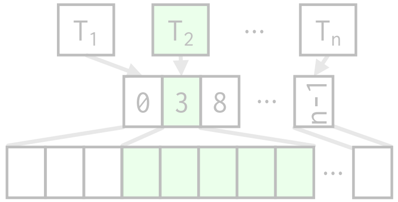
Synchronization

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

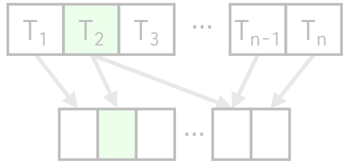
```
(0..n)
  .into_par_iter()
  .for_each(|i| {

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

  })
```



(SngInd)



(RngInd)



(AW)

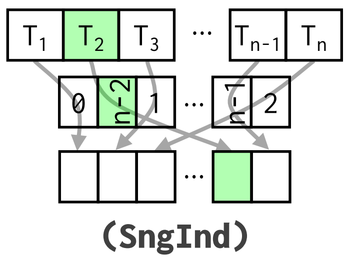
Programmers face a conundrum expressing irregular accesses

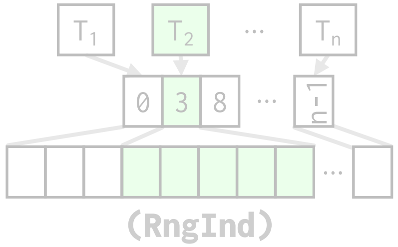
Synchronization **S**

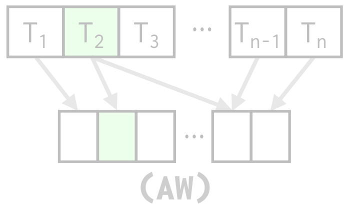
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

```
(0..n)
  .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

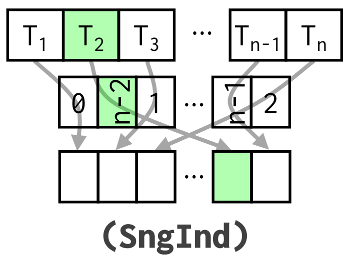
Synchronization **S**

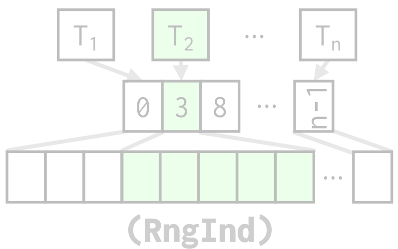
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

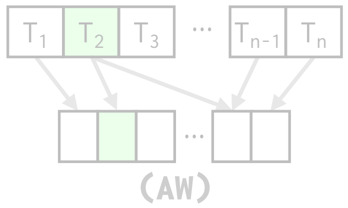
Unsafe

```
(0..n)
  .into_par_iter()
  .for_each(|i| {
    Old = v[offsets[i]].load(Order);
    v[offsets[i]].store(old+1, Order);
  })
```

Atomicity violation -> compiles!!







Programmers face a conundrum expressing irregular accesses

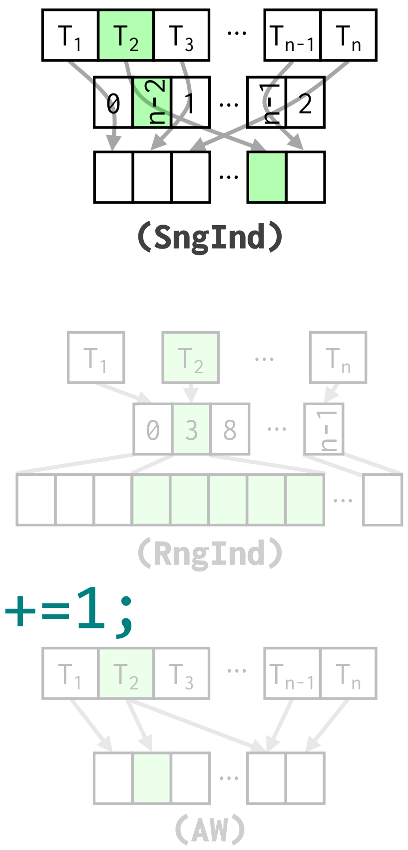
Synchronization **S**

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

Unsafe

- unchecked 1x **S**

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

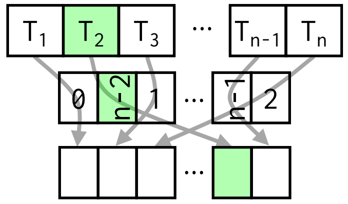


Programmers face a conundrum expressing irregular accesses

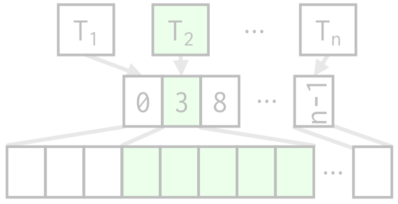
Synchronization **S**
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

Unsafe
- unchecked 1x **S**
- with check 3x **C**

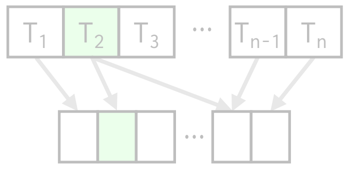
```
check_duplicates(offsets);
(0..n)
   .into_par_iter()
   .for_each(|i| {
     unsafe {
        *(v.as_ptr().add(i) as *mut T)+=1;
     }
   })
```



(SngInd)



(RngInd)



(AW)

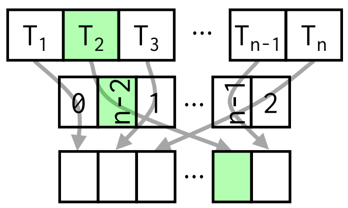
Programmers face a conundrum expressing irregular accesses

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

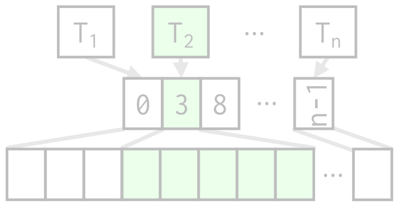
Unsafe
- unchecked 1x **S**
- with check 3x **C**

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

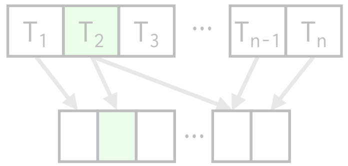
```
check_duplicates(offsets);
(0..n)
   .into_par_iter()
   .for_each(|i| {
     unsafe {
        *(v.as_ptr().add(i) as *mut T)+=1;
     }
   })
```



(SngInd)



(RngInd)



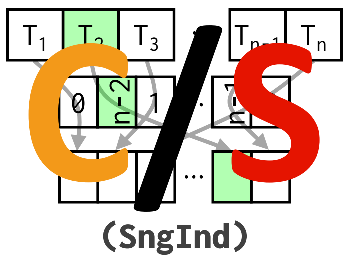
(AW)

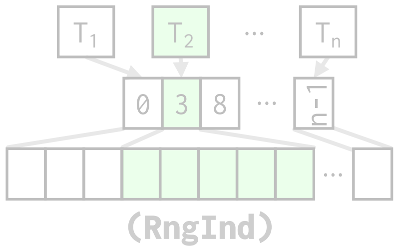
Programmers face a conundrum expressing irregular accesses

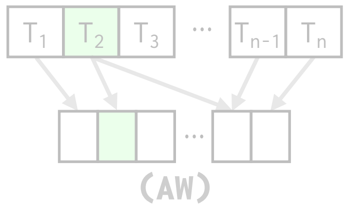
Synchronization **S**
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

Unsafe
- unchecked 1x **S**
- with check 3x **C**

Offsets { Unique **C/S**, Duplicates **S** }








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

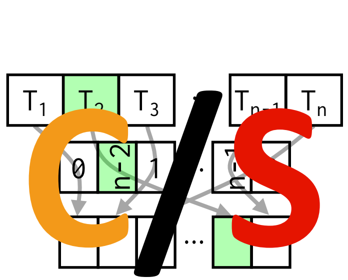
Programmers face a conundrum expressing irregular accesses

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

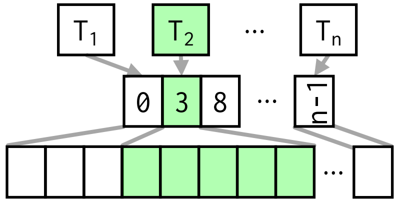
- Unsafe
  - unchecked 1x **S**
  - with check 3x **C**

Offsets { Unique **C/S** ; Duplicates **S** }

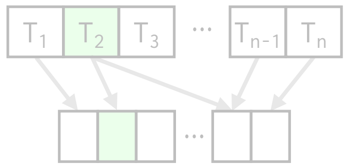
```
check_duplicates(offsets);
(0..n)
   .into_par_iter()
   .for_each(|i| {
     unsafe {
       *(v.as_ptr().add(i) as *mut T)+=1;
     }
   })
```



(SngInd)



(RngInd)


(AW)

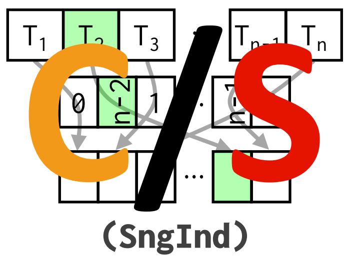
Programmers face a conundrum expressing irregular accesses

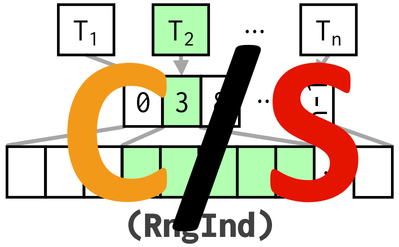
Synchronization **S**
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

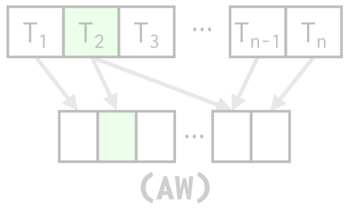
Unsafe
- unchecked 1x **S**
- with check 3x **C**

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

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







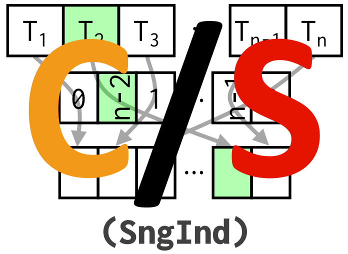
Programmers face a conundrum expressing irregular accesses

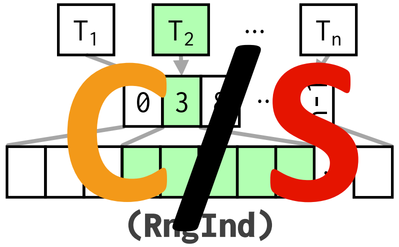
Synchronization **S**
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

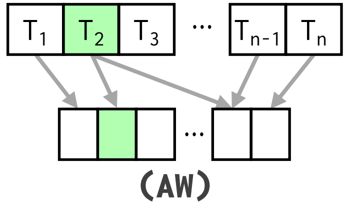
Unsafe
- unchecked 1x **S**
- with check 3x **C**

Offsets { Unique **C/S** ; Duplicates **S** }

```
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 **S**

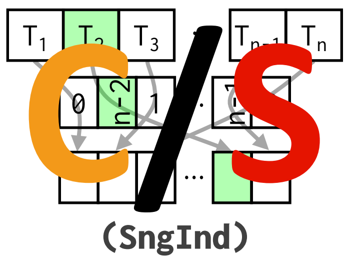
- Locks
  - Coarse 500x
  - Fine 5x
- Atomics
  - Relaxed 1x
  - SeqCst 2x

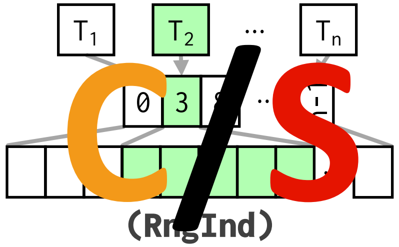
Unsafe

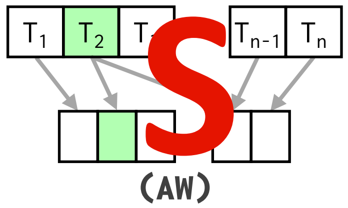
- unchecked 1x **S**
- with check 3x **C**

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

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







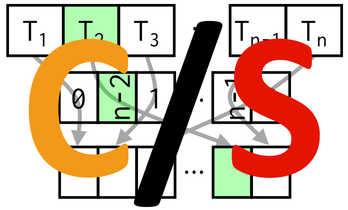
Programmers face a conundrum expressing irregular accesses

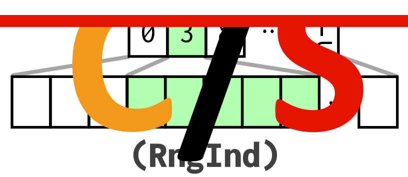
Synchronization **S**
- Locks
  - Coarse 500x
  - Relaxed 1x
  - SeqCst 2x

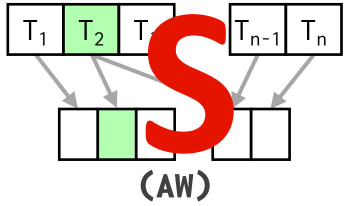
Unsafe
- unchecked 1x **S**
- with check 3x **C**

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

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





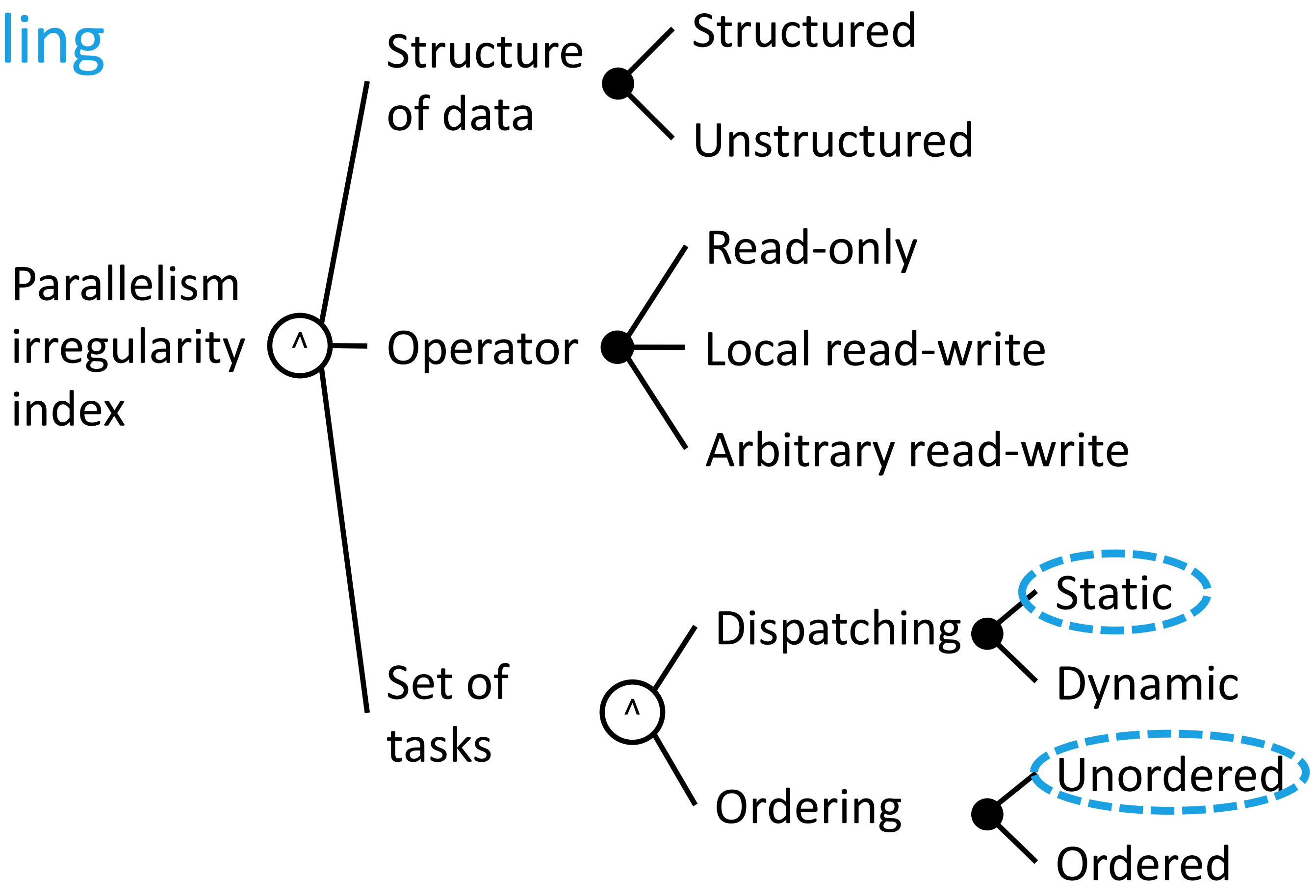


Rust solutions for irregular accesses are not fearless

Task scheduling

So far, static unordered scheduling

Let's consider other cases

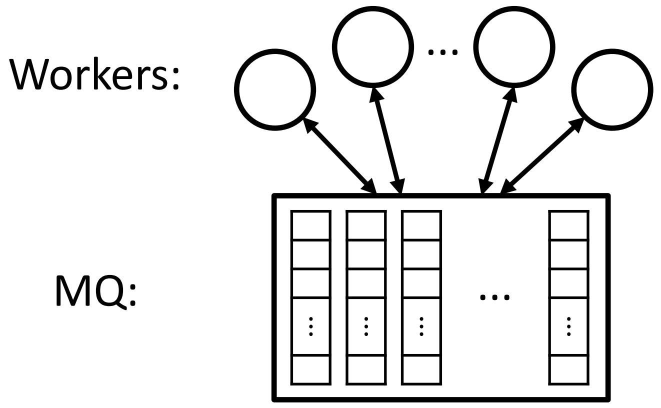


Task scheduling

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

Dynamic:

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



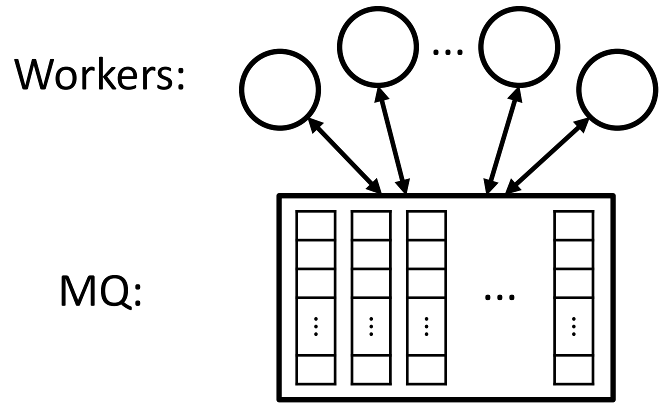
E.g., MultiQueue

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)



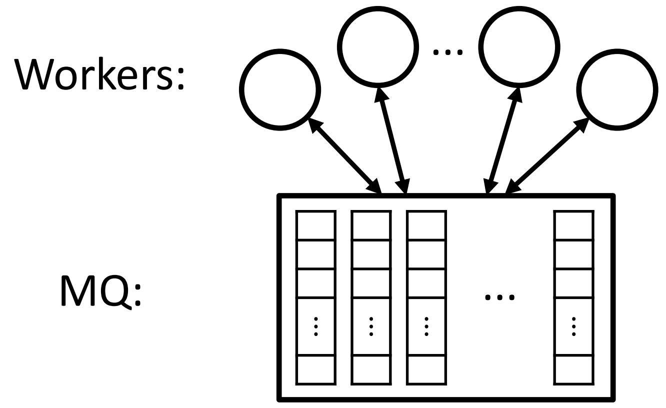
E.g., MultiQueue — User: depends on access types **F/C/S**

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**
- Implementer: Workers and Queue **S**

Task scheduling

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

Dynamic:

The dynamism of task scheduling does not affect fearlessness

E.g., MultiQueue
- User: depends on access types **F/C/S**
- Implementer: Workers and Queue **S**

Evaluation of RPB

Irregular accesses are common in RPB

	Tasks' Accesses							Task dispatch	
	Regular				Irregular				
Bench-mark	RO	Stride	Block	D&C	SngInd	RngInd	AW	static	dynamic
bw	✔	✔			✔	✔	✔	✔	
dedup	✔	✔					✔	✔	
dr	✔	✔			✔	✔	✔	✔	
hist	✔	✔				✔	✔	✔	
isort	✔					✔	✔	✔	
lrs	✔	✔	✔		✔	✔		✔	
mis	✔		✔		✔	✔		✔	
mm	✔		✔		✔	✔		✔	
msf	✔		✔	✔	✔	✔		✔	
sa	✔	✔		✔	✔	✔		✔	
sf	✔		✔		✔	✔		✔	
sort	✔		✔	✔		✔		✔	
bfs							✔		✔
sssp							✔		✔

Irregular accesses are common in RPB

Benchmark	Tasks' Accesses: Regular				Tasks' Accesses: Irregular			Task dispatch	
	RO	Stride	Block	D&C	SngInd	RngInd	AW	static	dynamic
bw	✔	✔			✔	✔	✔	✔	
dedup	✔	✔					✔	✔	
dr	✔	✔			✔	✔	✔	✔	
hist	✔	✔				✔	✔	✔	
isort	✔					✔	✔	✔	
lrs	✔	✔	✔		✔	✔		✔	
mis	✔		✔		✔	✔		✔	
mm	✔		✔		✔	✔		✔	
msf	✔		✔	✔	✔	✔		✔	
sa	✔	✔		✔	✔	✔		✔	
sf	✔		✔		✔	✔		✔	
sort	✔		✔	✔		✔		✔	
bfs							✔		✔
sssp							✔		✔

Regular accesses **F**

Irregular accesses **C/S**

Irregular accesses are common in RPB

Benchmark	Tasks' Accesses: Regular: RO	Stride	Block	D&C	Irregular: SngInd	RngInd	AW	Task dispatch: static	dynamic
bw	✔	✔			✔	✔	✔	✔	
dedup	✔	✔					✔	✔	
dr	✔	✔			✔	✔	✔	✔	
hist	✔	✔				✔	✔	✔	
isort	✔					✔	✔	✔	
sa	✔	✔		✔	✔	✔		✔	
sf	✔		✔		✔	✔		✔	
sort	✔		✔	✔		✔		✔	
bfs							✔		✔
sssp							✔		✔

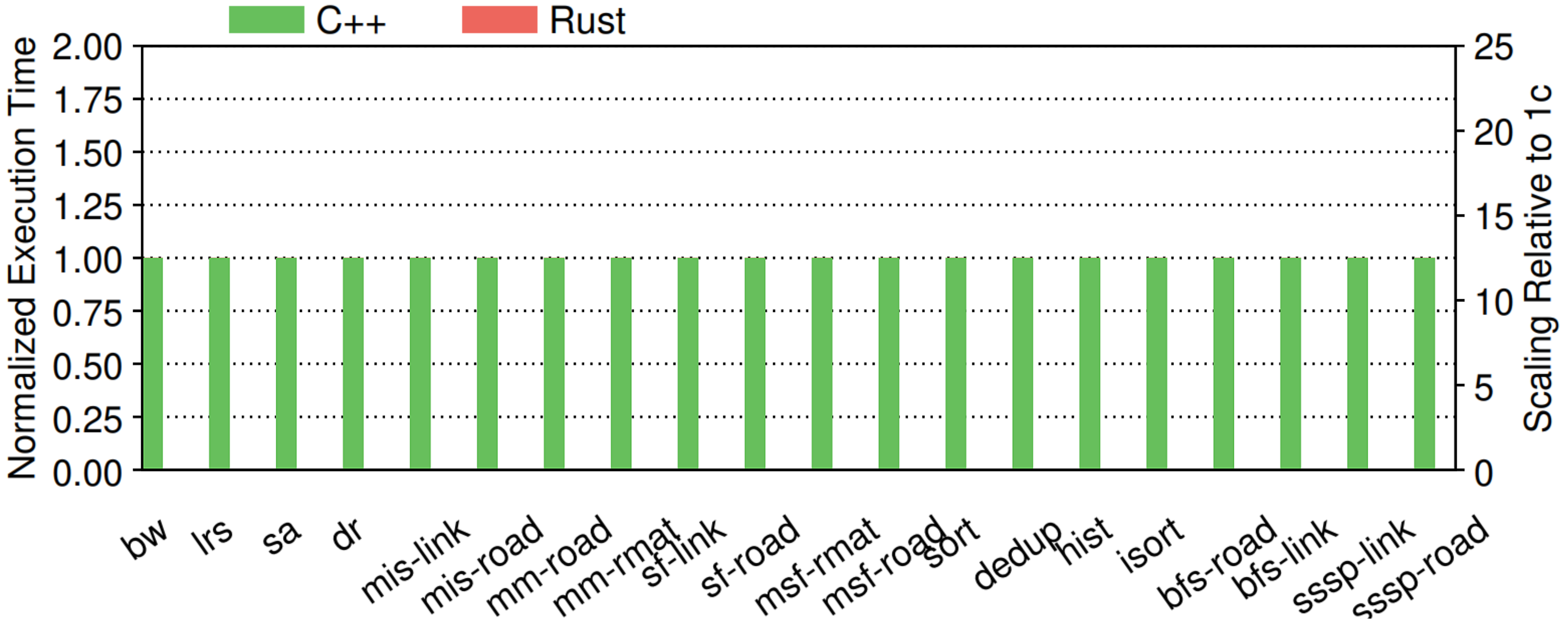
Regular accesses F

Irregular accesses C/S

Expressing RPB in Rust is not fearless

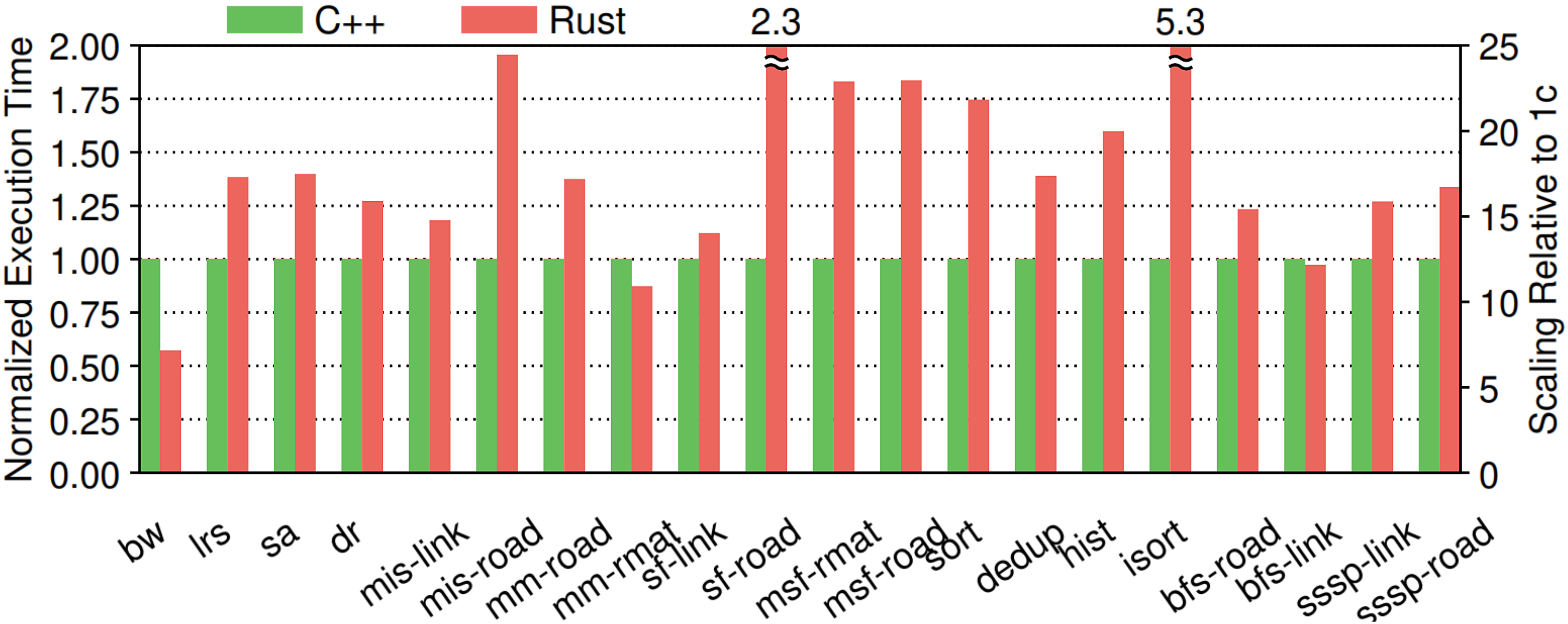
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses (24 cores)



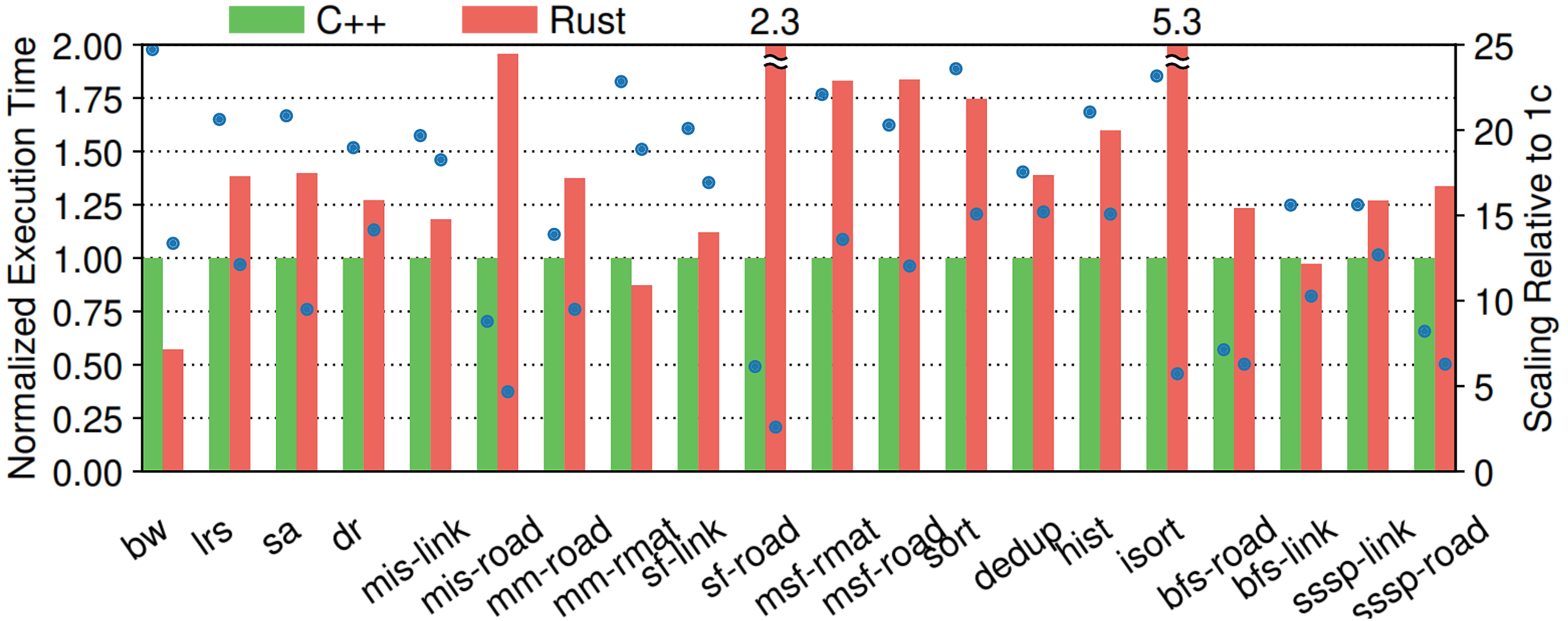
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses (24 cores)



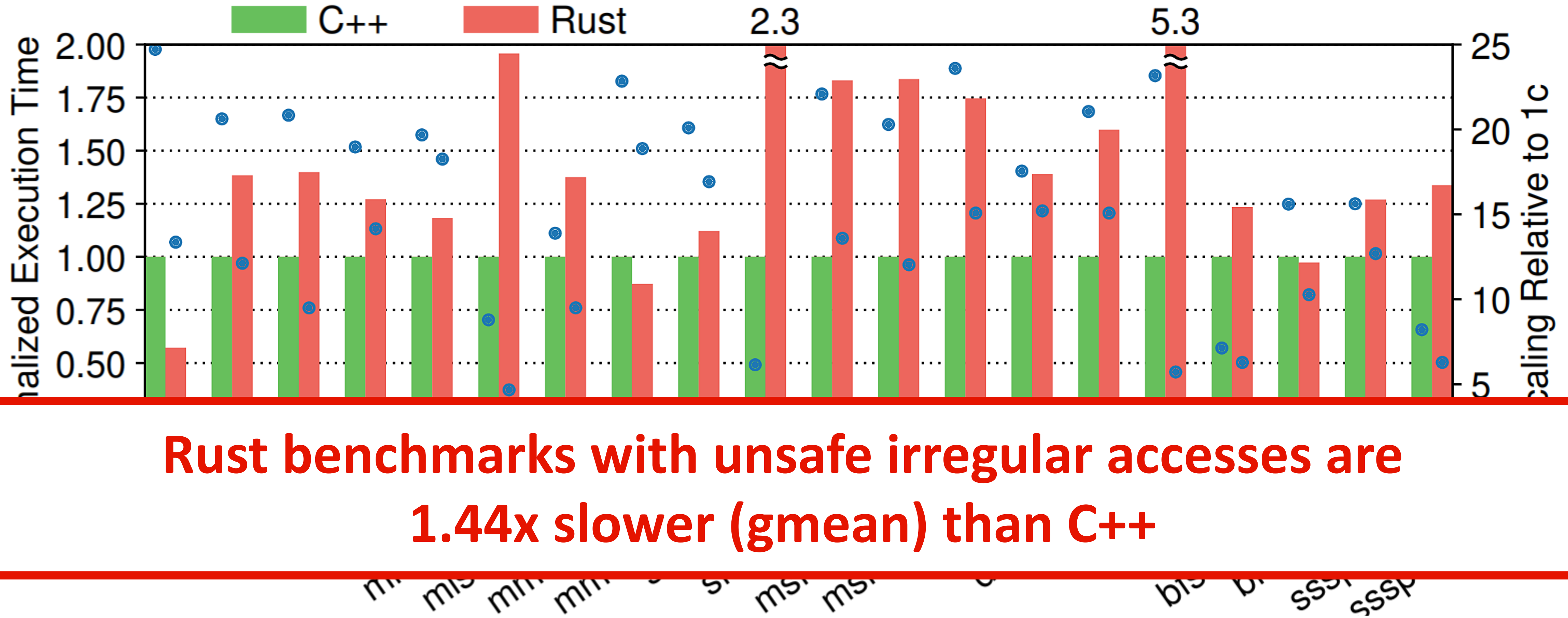
Performance of RPB vs C++ across benchmarks

* using unsafe for irregular accesses (24 cores)



Performance of RPB vs C++ across benchmarks

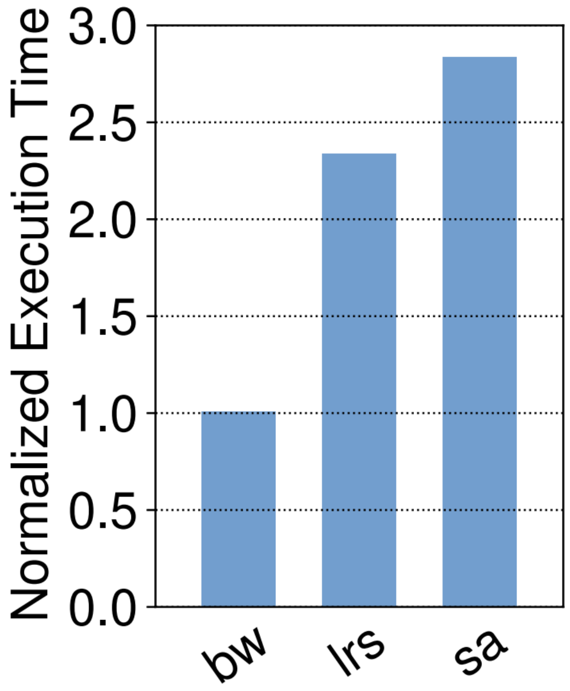
* using unsafe for irregular accesses (24 cores)



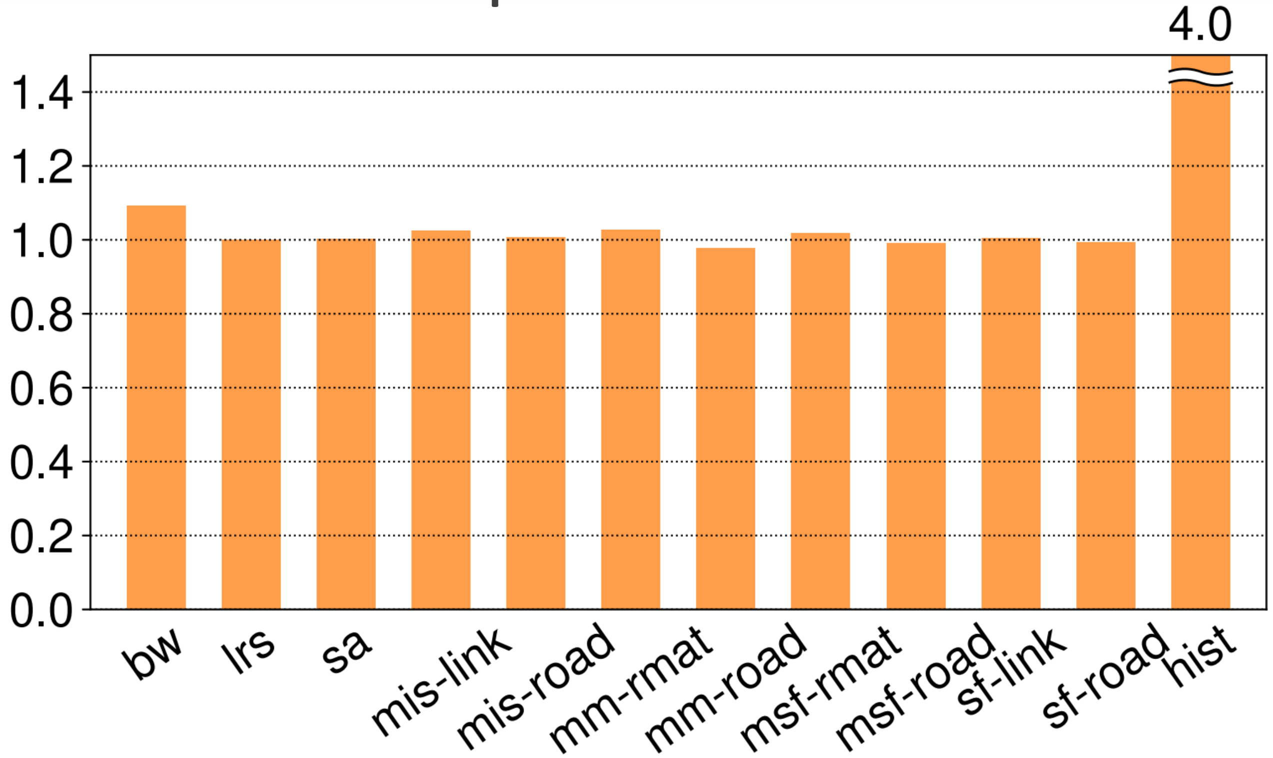
Rust benchmarks with unsafe irregular accesses are 1.44x slower (gmean) than C++

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**

UNIVERSITY OF TORONTO

Find RPB on github!

Conclusions

To replace C++, Rust’s claims should be verified, including fearle 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**

UNIVERSITY OF TORONTO