Rayon

Data Parallelism for Fun and Profit

Nicholas Matsakis
(nmatsakis on IRC)

Want to make parallelization **easy**

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.iter()
       .map(|path| Image::load(path))
       .collect()
}
```

For each path…

…load an image…

…create and return a vector.

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.par_iter()
       .map(|path| Image::load(path))
       .collect()
}
```

Want to make parallelization **safe**

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
    let mut pngs = 0;
    paths.par_iter()
        .map(|path| {
            if path.ends_with(“png”) {
                pngs += 1;
            }
            Image::load(path)
        })
        .collect()
}
```

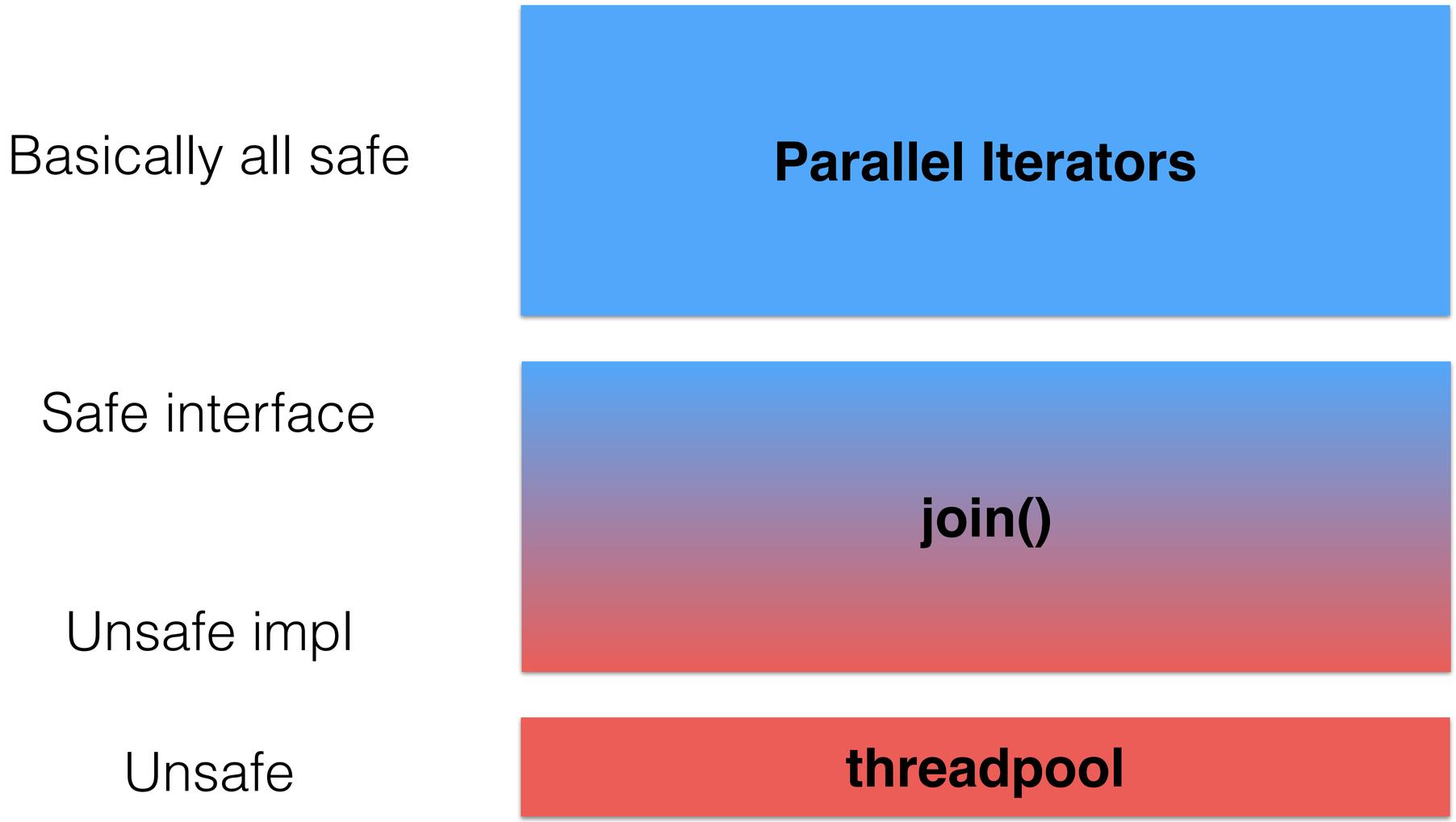
Data-race

Will not compile

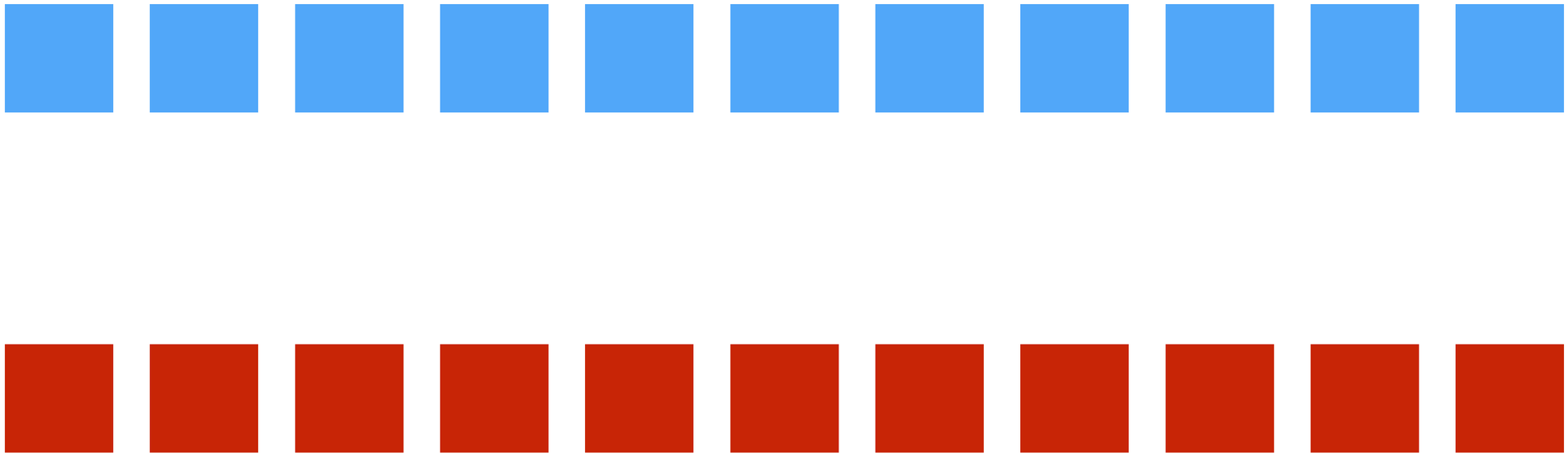
Saved by the compiler: Parallelizing a loop with Rust and rayon

Eric Kidd on Thursday 20 Oct 2016

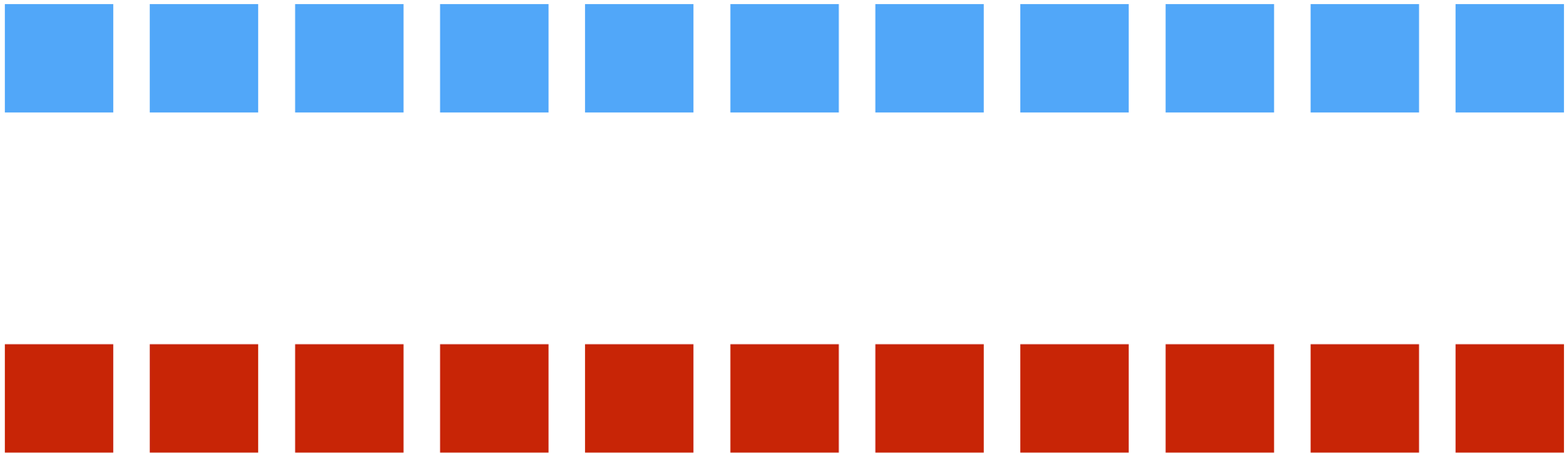
http://blog.faraday.io/saved-by-the-compiler-parallelizing-a-loop-with-rust-and-rayon/


Basically all safe
Parallel Iterators
Safe interface
join()
Unsafe impl
Unsafe
threadpool

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.iter()
       .map(|path| Image::load(path))
       .collect()
}
```



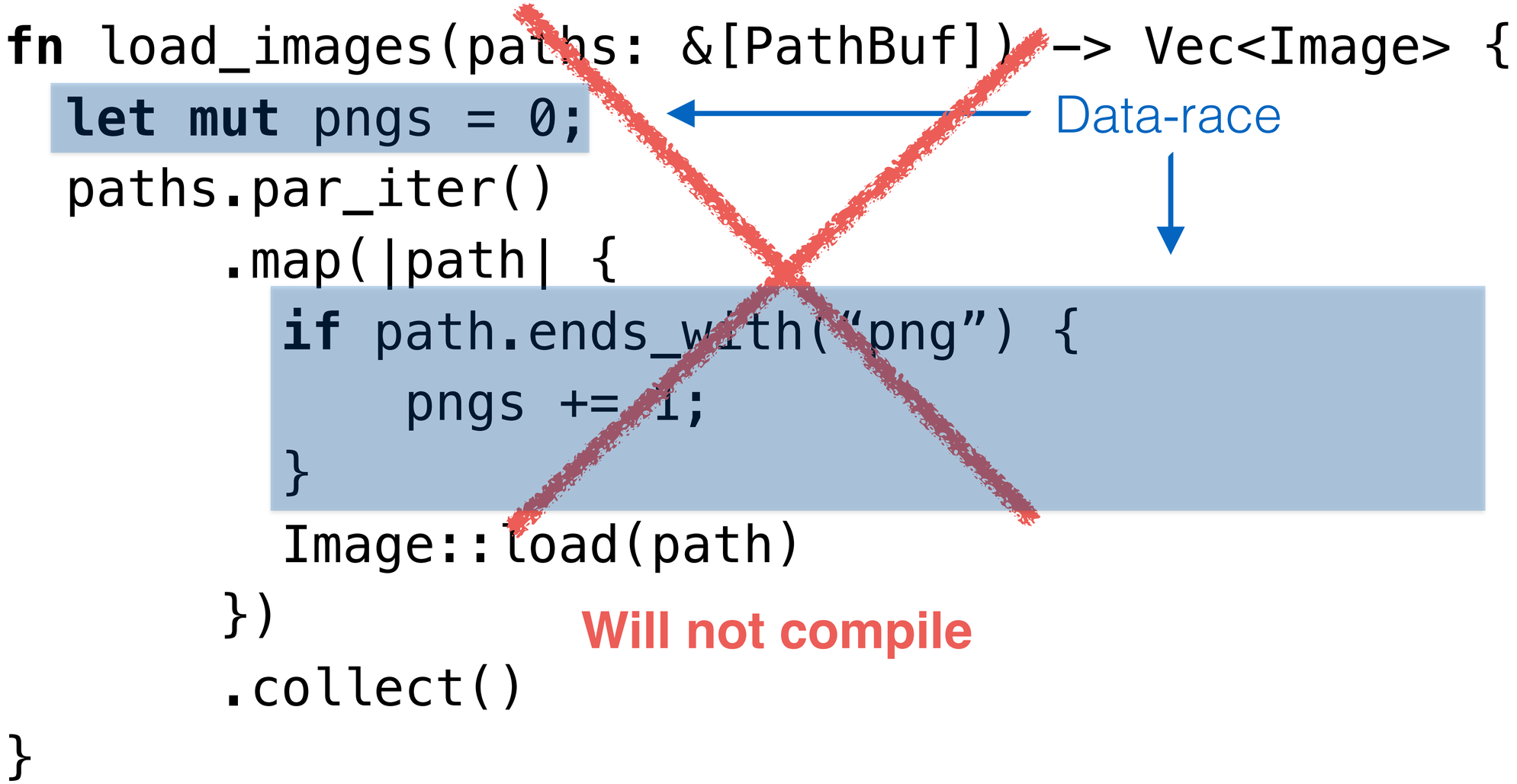
```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.par_iter()
       .map(|path| Image::load(path))
       .collect()
}
```



Not quite **that** simple…

(but **almost**!)

1. No mutating **shared state** (except for atomics, locks).
2. Some combinators are inherently sequential.
3. Some things aren’t implemented yet.


fn load_images(paths: &[PathBuf]) -> Vec<Image> {
let mut pngs = 0;
paths.par_iter()
.map(|path| {
if path.ends_with(“png”) {
pngs += 1;
}
Image::load(path)
})
.collect()
}
Data-race
Will not compile

```rust
fn increment_all(counts: &mut [u32]) {
  for c in counts.iter_mut() {
    *c += 1;
  }
}
```

```rust
fn increment_all(counts: &mut [u32]) {
  paths.par_iter_mut()
       .for_each(|c| *c += 1);
}
```

`c` not shared between iterations!

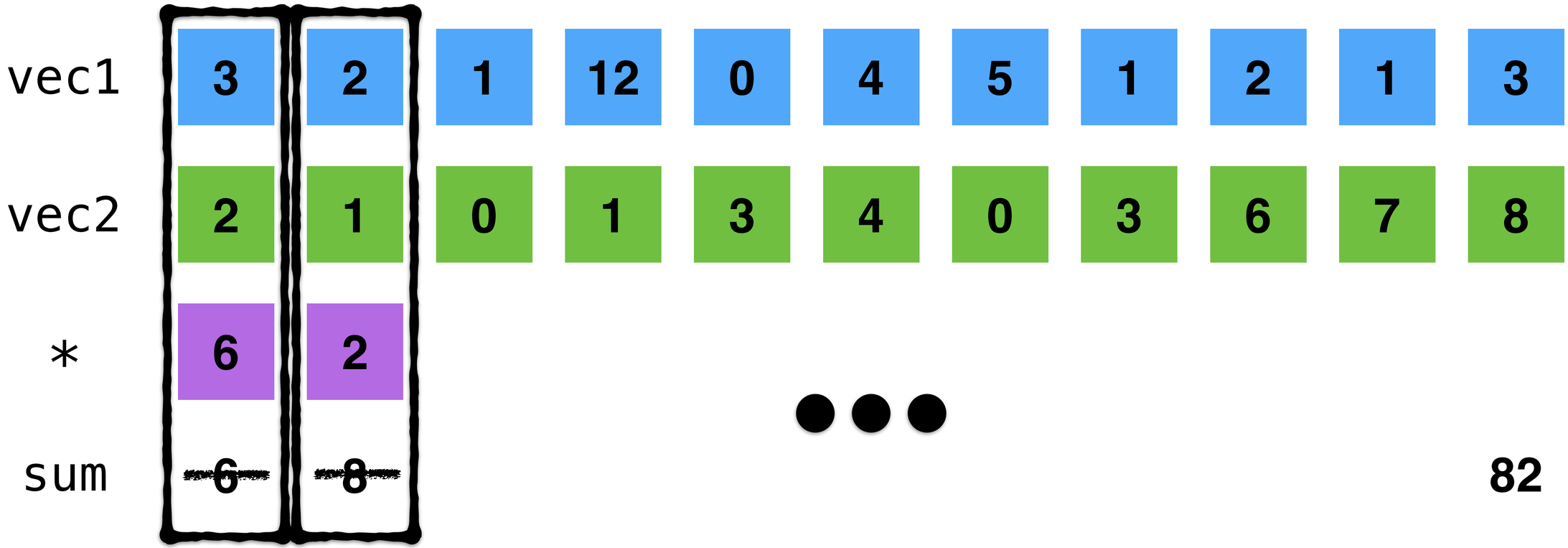
```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  let pngs =
    paths.par_iter()
         .filter(|p| p.ends_with(“png”))
         .map(|_| 1)
         .sum();

  paths.par_iter()
       .map(|p| Image::load(p))
       .collect()
}
```

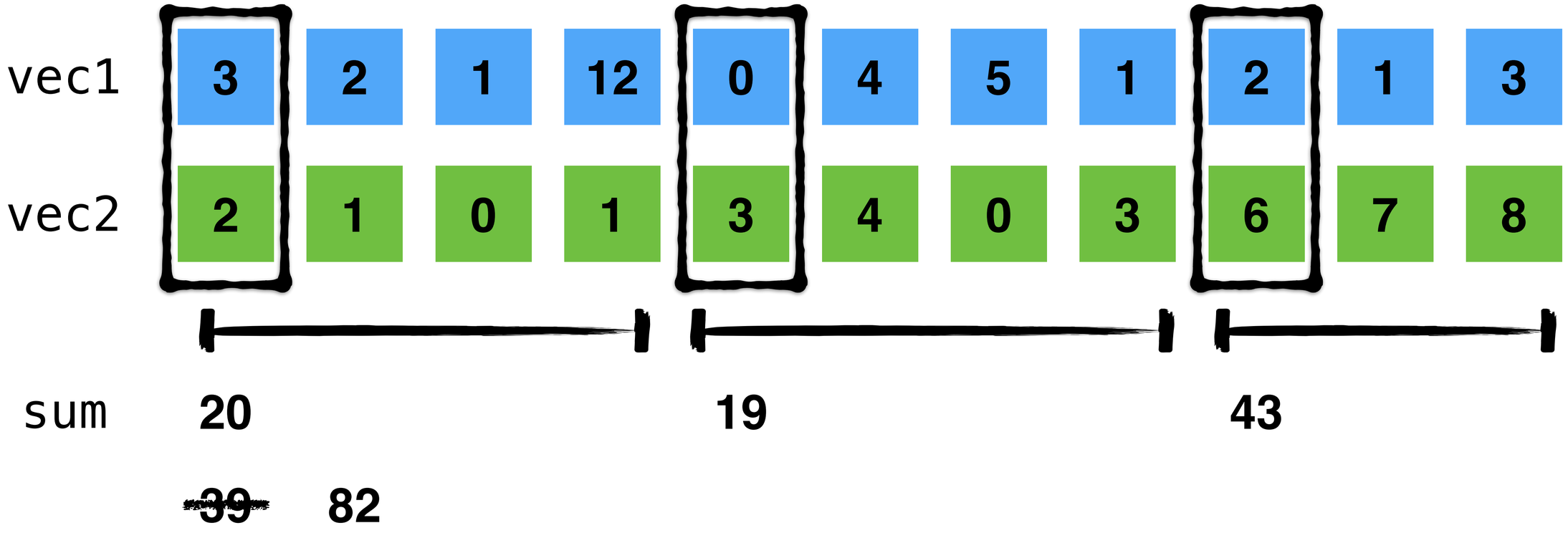
```
use std::sync::atomic::{AtomicUsize, Ordering};
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  let pngs = AtomicUsize::new(0);
  paths.par_iter()
       .map(|path| {
         if path.ends_with(“png”) {
            pngs.fetch_add(1, Ordering::SeqCst);
         }
         Image::load(path)
       })
       .collect()
}
```

But beware: atomics introduce nondeterminism!

```
fn dot_product(vec1: &[i32], vec2: &[i32]) -> i32 {
    vec1.iter()
        .zip(vec2)
        .map(|(e1, e2)| e1 * e2)
        .fold(0, |a, b| a + b)                    // aka .sum()
}
```



```
fn dot_product(vec1: &[i32], vec2: &[i32]) -> i32 {
  vec1.par_iter()
      .zip(vec2)
      .map(|(e1, e2)| e1 * e2)
      .reduce(|| 0, |a, b| a + b)      // aka .sum()
}
```



Parallel iterators:

Mostly like normal iterators, but:

- closures cannot mutate shared state
- some operations are different

For the most part, Rust protects you from surprises.

Parallel Iterators

join()

threadpool

The primitive: join()

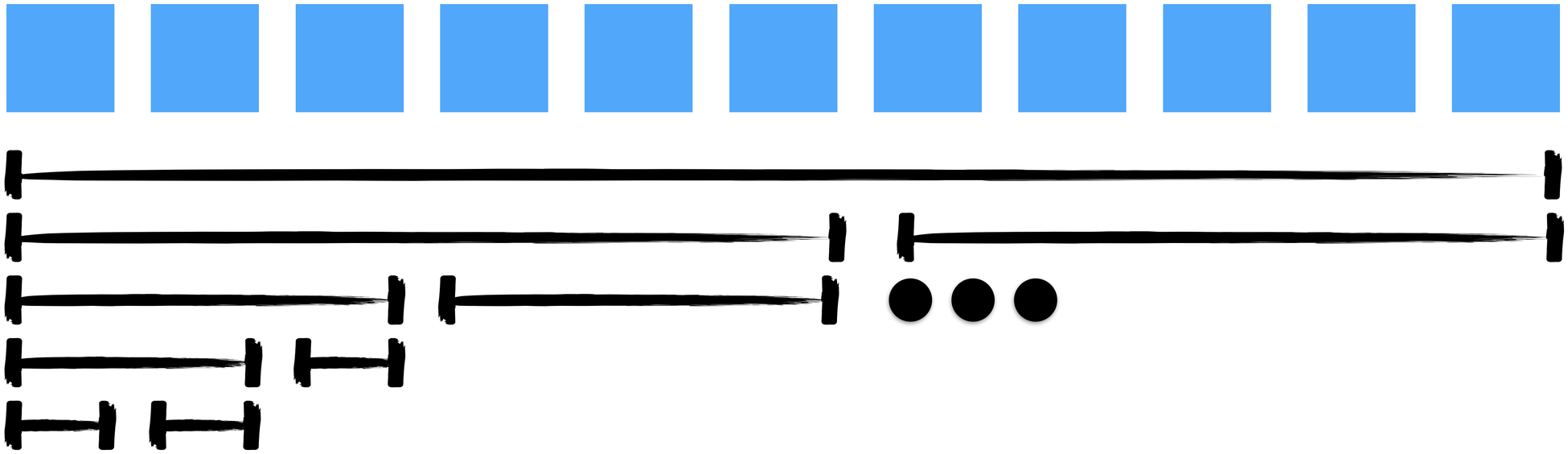
```
rayon::join(|| do_something(…),
            || do_something_else(…));
```

Meaning: **maybe** execute two closures in parallel.

Idea:

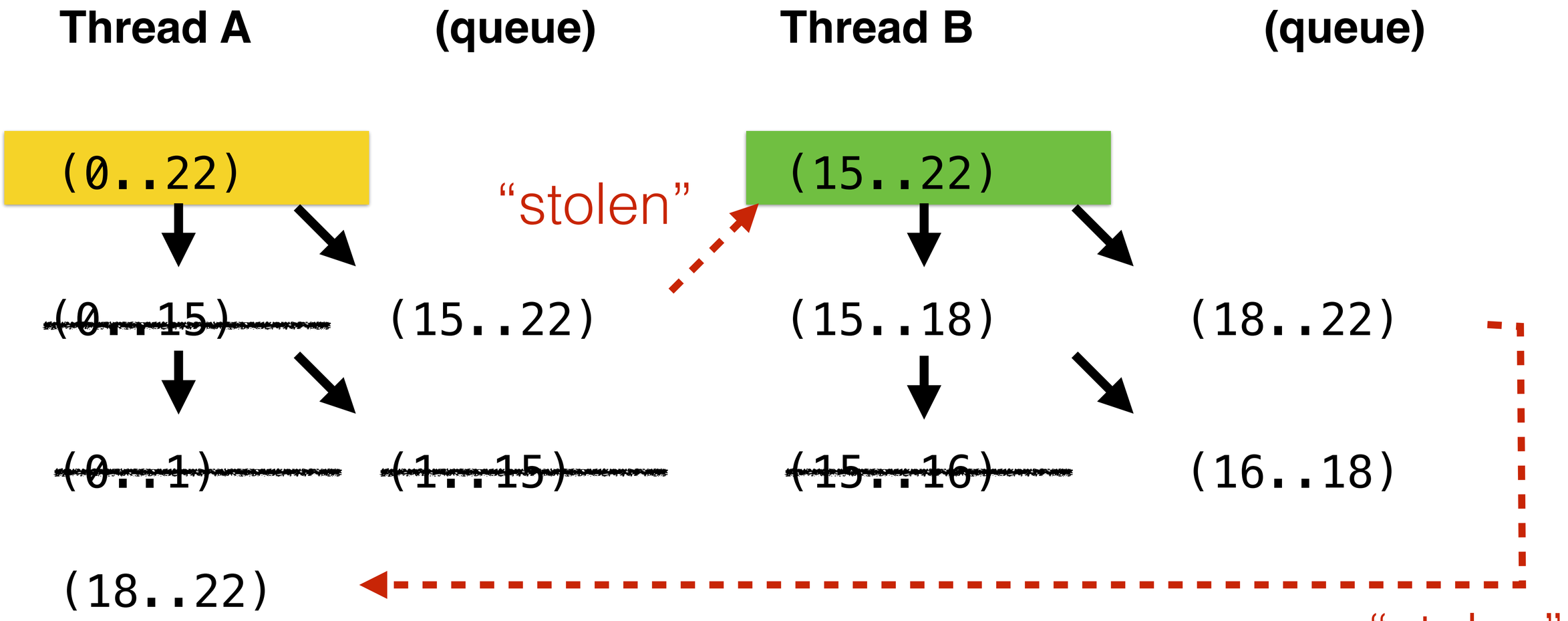
- add `join` wherever parallelism is **possible**
- let the library decide when it is **profitable**

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.par_iter()
       .map(|path| Image::load(path))
       .collect()
}
```



```
Image::load(paths[0])
     Image::load(paths[1])
```

Work stealing



Cilk: http://supertech.lcs.mit.edu/cilk/

RFC: adaptive "thief splitting" algorithm for par_iter #106

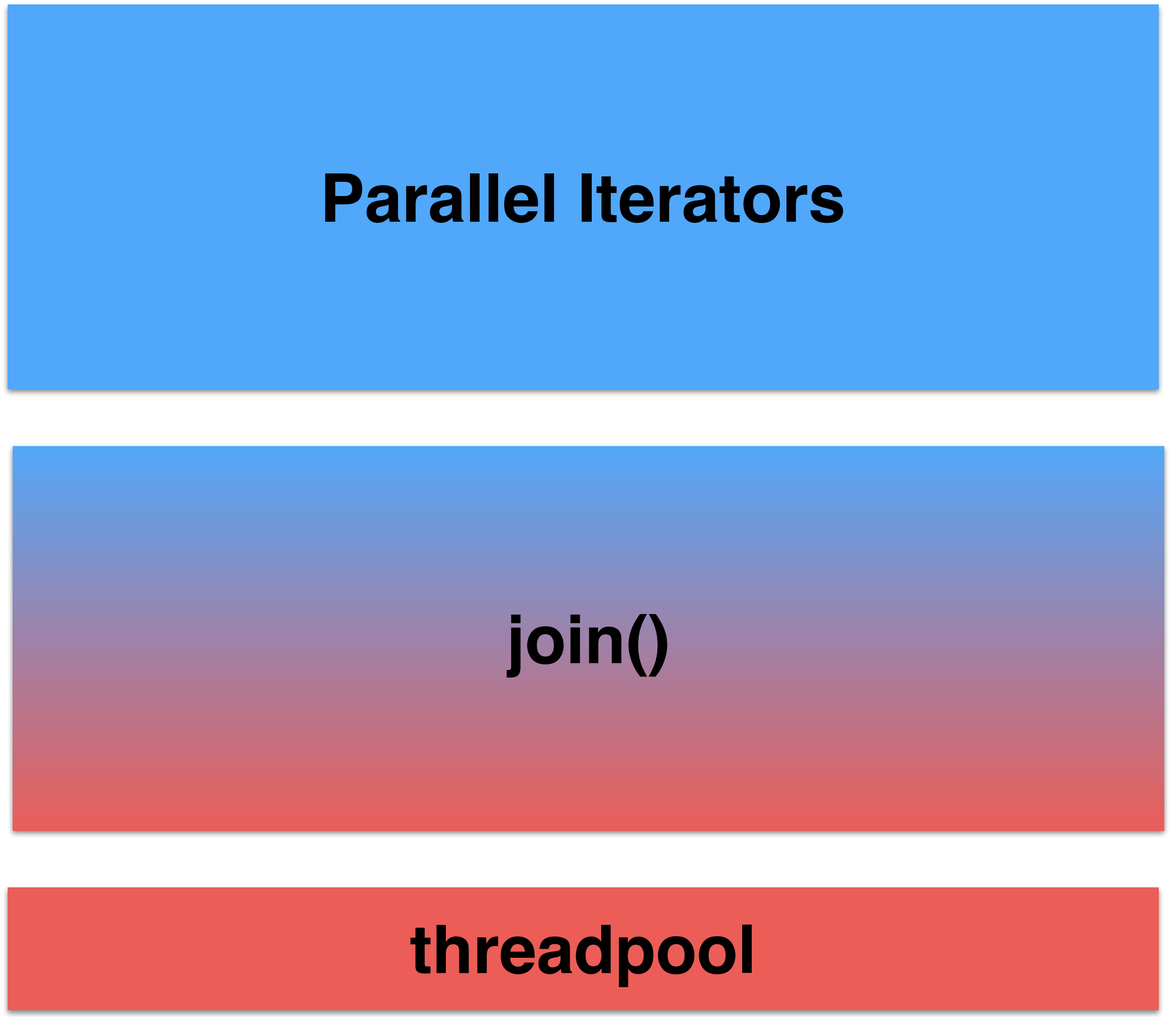
Merged **nikomatsakis** merged 4 commits into `nikomatsakis:master` from `cuviper:thief-splitter` 9 days ago

Conversation 33 | Commits 4 | Files changed 12

cuviper commented 14 days ago

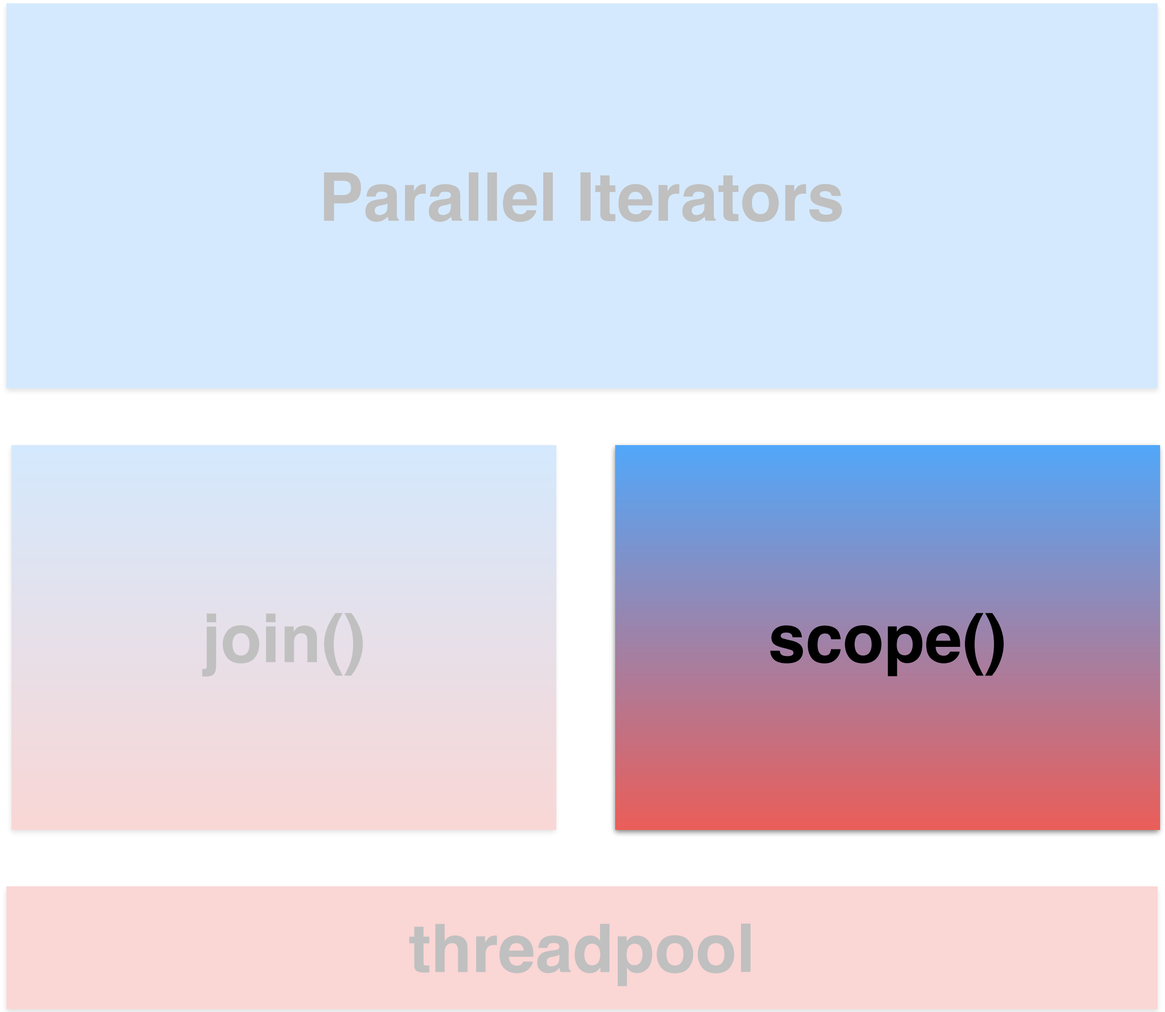
This introduces an adaptive algorithm for splitting par_iter jobs, used only as a default case when no weights are specified. Initially, it will split into enough jobs to fill every thread. Then whenever a job is stolen, that job will again be split enough for every thread.

This is roughly based on **@Amanieu**'s description in the users forum of the algorithm used by Async++ and TBB.

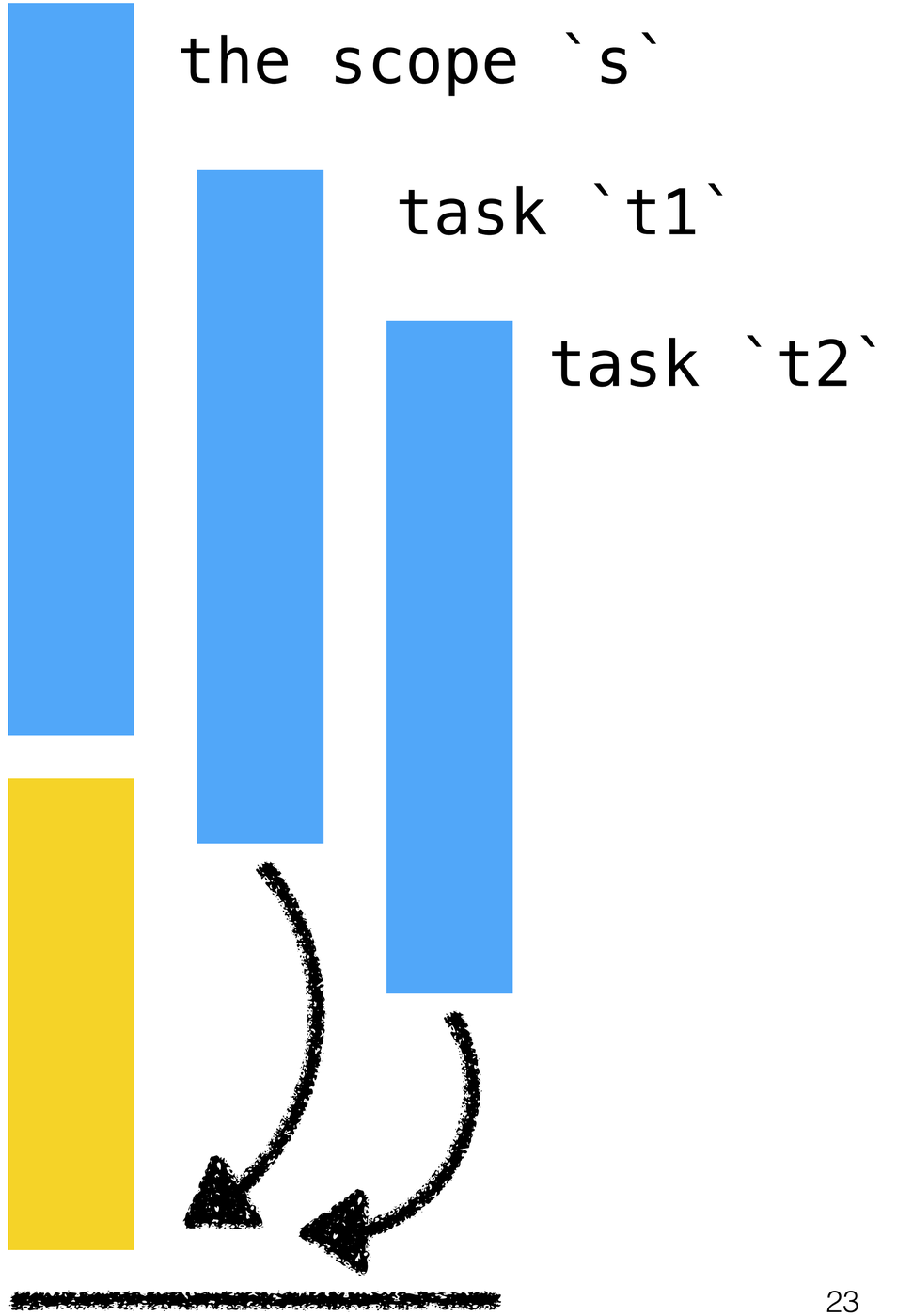


Rayon:

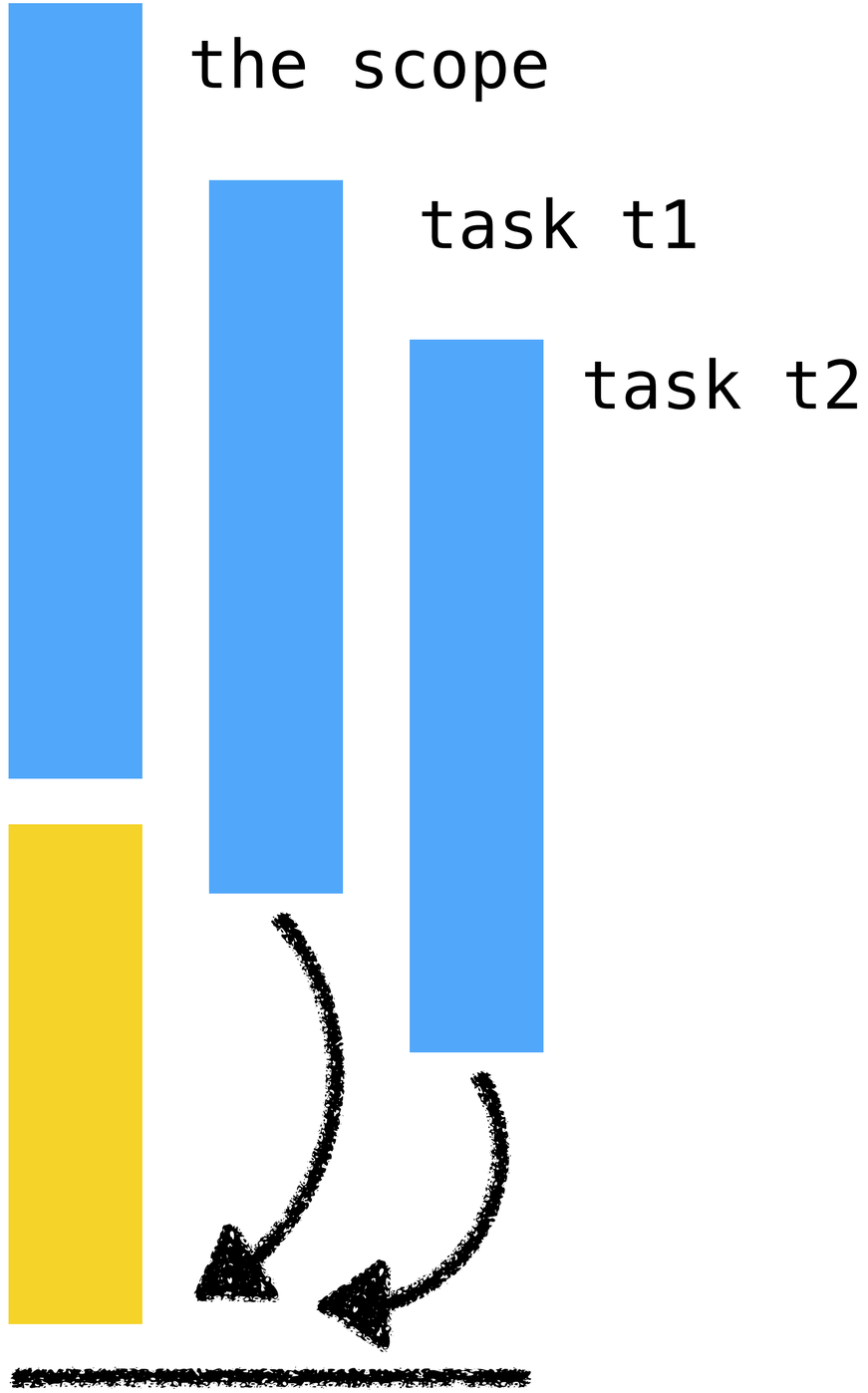
- Parallelize for fun and profit
- Variety of APIs available
- Future directions:
  - more iterators
  - integrate SIMD, array ops
  - integrate persistent trees
  - factor out threadpool


Parallel Iterators
join()
scope()
threadpool

```
rayon::scope(|s| {
    …
    s.spawn(move |s| {
        // task t1
    });
    s.spawn(move |s| {
        // task t2
    });
    …
});
```



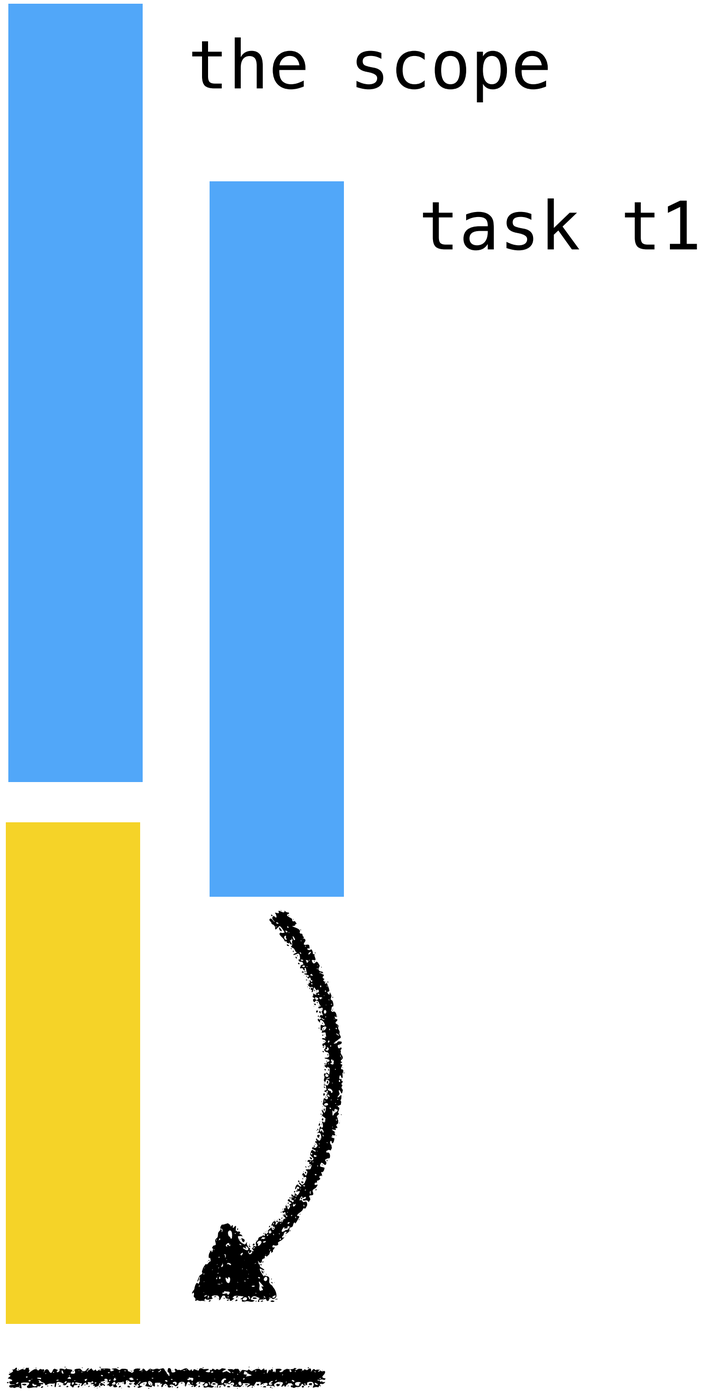
```
rayon::scope(|s| {
    …
    s.spawn(move |s| {
        // task t1
        s.spawn(move |s| {
            // task t2
            …
        });
        …
    });
    …
});
```



```
let ok: &[u32]s = &[…];

rayon::scope(|scope| {
    …
    let not_ok: &[u32] = &[…];
    …
    scope.spawn(move |scope| {
        // which variables can t1 use?
    });
});
```

`not_ok` is freed here



```
fn join<A,B>(a: A, b: B)
    where A: FnOnce() + Send,
          B: FnOnce() + Send,
{
    rayon::scope(|scope| {
        scope.spawn(move |_| a());
        scope.spawn(move |_| b());
    });
}
```

(Real join avoids heap allocation)

```
struct Tree<T> {
    value: T,
    children: Vec<Tree<T>>,
}

impl<T> Tree<T> {
  fn process_all(&mut self) {
    process_value(&mut self.value);
    for child in &mut self.children {
        child.process_all();
    }
  }
}
```

```
impl<T> Tree<T> {
  fn process_all(&mut self) where T: Send {
    rayon::scope(|scope| {
      for child in &mut self.children {
        scope.spawn(move |_| child.process_all());
      }
      process_value(&mut self.value);
    });
  }
}
```

```
impl<T> Tree<T> {
  fn process_all(&mut self) where T: Send {
    rayon::scope(|scope| {
      let children = &mut self.children;
      scope.spawn(move |scope| {
        for child in &mut children {
          scope.spawn(move |_| child.process_all());
        }
      });
      process_value(&mut self.value);
    });
  }
}
```

```
impl<T: Send> Tree<T> {
  fn process_all(&mut self) {
    rayon::scope(|s| self.process_in(s));
  }

  fn process_in<'s>(&'s mut self, scope: &Scope<'s>) {
    let children = &mut self.children;
    scope.spawn(move |scope| {
      for child in &mut children {
        scope.spawn(move |scope| child.process_in(scope));
      }
    });
    process_value(&mut self.value);
  }
}
```