Rayon

Frédéric Wagner

2024

Ensimag

Plan

Rayon

Itérateurs parallèles

À l'intérieur d'un itérateur parallèle

Join

- primitive de base de rayon
  - **note : en vrai c'est plutôt `join_context`**
- exécute deux fonctions (sans arguments) **potentiellement** en simultané

Join

- primitive de base de rayon
  - **note : en vrai c'est plutôt `join_context`**
- exécute deux fonctions (sans arguments) **potentiellement** en simultané
- une programmation sans threads !

Join

- primitive de base de rayon
  - **note : en vrai c'est plutôt `join_context`**
- exécute deux fonctions (sans arguments) **potentiellement** en simultané
- une programmation sans threads !
  - les threads c'est l'enfer

Join

- primitive de base de rayon
  - **note : en vrai c'est plutôt `join_context`**
- exécute deux fonctions (sans arguments) **potentiellement** en simultané
- une programmation sans threads !
  - les threads c'est l'enfer
- comme il n'y a pas d'arguments les fermetures sont utiles ici

Retour sur la Cell

```rust
use std::cell::Cell;
let c = Cell::new(3);
rayon::join(|| c.set(3), || c.set(4));
```

```
|   rayon::join(|| c.set(3), || c.set(4));
|   ----------- ^^^^^^^^^^^ `Cell<i32>` cannot be shared between threads safely
|   |
|   required by a bound introduced by this call
|
= help: the trait `Sync` is not implemented for `Cell<i32>`
```

Send et Sync

- deux auto-traits pour la concurrence
  - implémentés automatiquement par le compilateur
  - implémentable manuellement par l'utilisateur
    - **unsafe**

Send et Sync

- deux auto-traits pour la concurrence
  - implémentés automatiquement par le compilateur
  - implémentable manuellement par l'utilisateur
    - **unsafe**
- **Send**

Send et Sync

- deux auto-traits pour la concurrence
  - implémentés automatiquement par le compilateur
  - implémentable manuellement par l'utilisateur
    - **unsafe**
- **Send**
  - types qui peuvent être déplacés d'un thread à un autre
- Sync

Send et Sync

- deux auto-traits pour la concurrence
  - implémentés automatiquement par le compilateur
  - implémentable manuellement par l'utilisateur
    - **unsafe**
- **Send**
  - types qui peuvent être déplacés d'un thread à un autre
- Sync
  - types qui peuvent être pris par référence dans un autre thread
  - T est **Sync** ssi `&T` est **Send**

Send et Sync

- deux auto-traits pour la concurrence
  - implémentés automatiquement par le compilateur
  - implémentable manuellement par l'utilisateur
    - **unsafe**
- **Send**
  - types qui peuvent être déplacés d'un thread à un autre
- Sync
  - types qui peuvent être pris par référence dans un autre thread
  - T est **Sync** ssi `&T` est **Send**
- Cell ?

Send et Sync

- deux auto-traits pour la concurrence
  - implémentés automatiquement par le compilateur
  - implémentable manuellement par l'utilisateur
    - **unsafe**
- **Send**
  - types qui peuvent être déplacés d'un thread à un autre
- Sync
  - types qui peuvent être pris par référence dans un autre thread
  - T est **Sync** ssi `&T` est **Send**
- Cell ?
  - **Send** mais pas **Sync**
  - car **UnsafeCell** est **Send** mais pas **Sync**

Typage

```
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
  A: FnOnce() -> RA + Send,
  B: FnOnce() -> RB + Send,
  RA: Send,
  RB: Send,
```

Typage

```
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
  A: FnOnce() -> RA + Send,
  B: FnOnce() -> RB + Send,
  RA: Send,
  RB: Send,
    let c = Cell::new(0);
    join(|| c.set(3), || c.set(4));
```

- **Cell** est **Send**, pourquoi ça ne compile pas ?

Typage

```
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
  A: FnOnce() -> RA + Send,
  B: FnOnce() -> RB + Send,
  RA: Send,
  RB: Send,
```

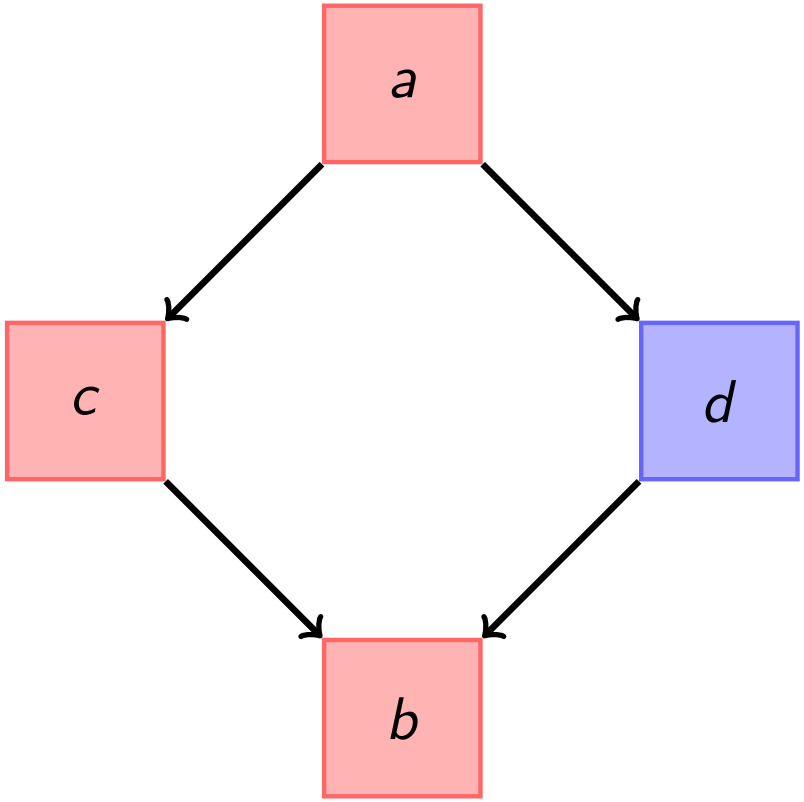
```
let c = Cell::new(0);
join(|| c.set(3), || c.set(4));
```

- **Cell** est **Send**, pourquoi ça ne compile pas ?
- les fermetures A et B empruntent la cellule qui n'est pas **Sync** et donc ne sont pas **Send**

Taches

```
fn foo() {
  a();
  join(c, d);
  b();
}
```

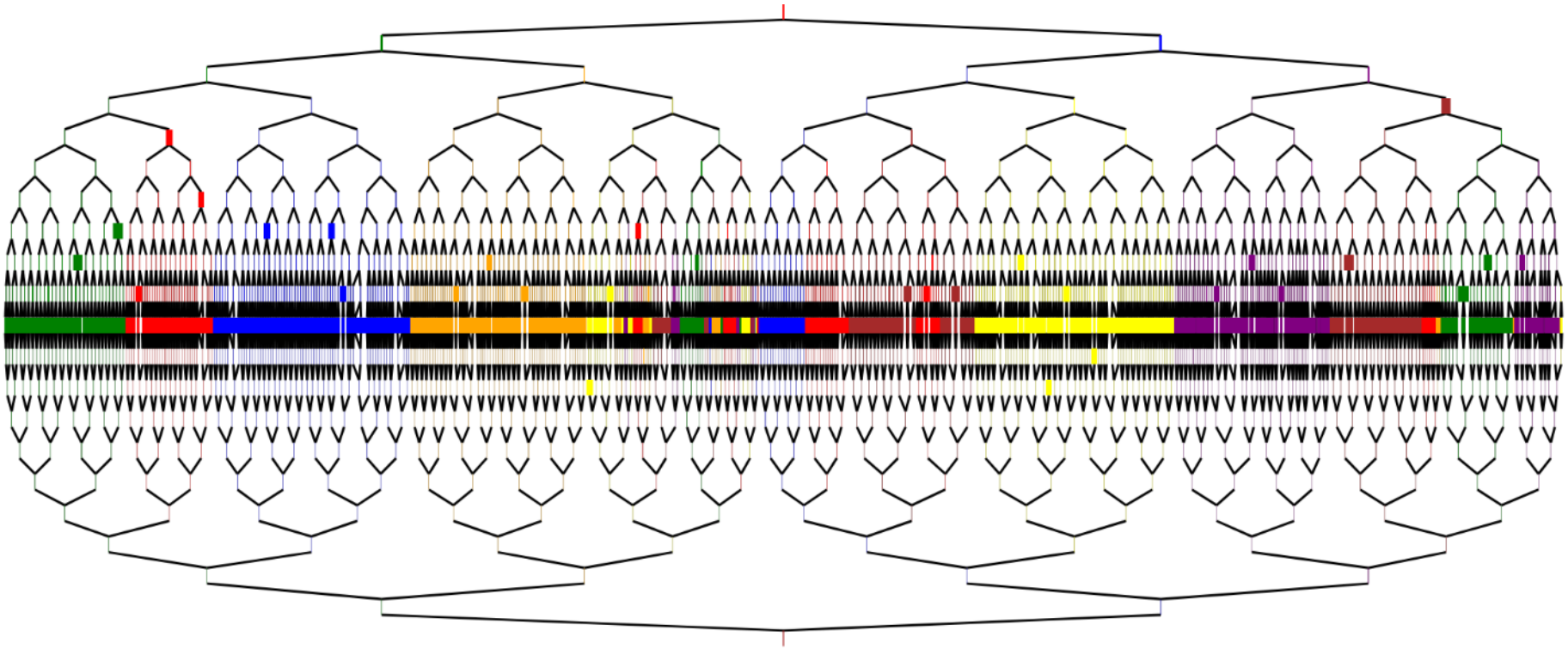
- crée une nouvelle tâche pour d
- exécute c immédiatement
- exécute d si encore là
- synchro avant b



Somme des $f(x)$

```
fn somme_f_rec<F>(r: Range<u32>, f: &F) -> u32
where F: Fn(u32) -> u32 + Sync,
{
  if r.len() < 1000 {
    r.map(f).sum()
  } else {
    let milieu = (r.start + r.end) / 2;
    let gauche = r.start..milieu;
    let droite = milieu..r.end;
    let (somme_gauche, somme_droite) = join(
      || somme_f_rec(gauche, f),
      || somme_f_rec(droite, f),
    );
    somme_gauche + somme_droite
  }
}
```

Log d'exécution (10^6 entiers)



- le graphe de tâches est donné par le code
- la coloration (les threads) par le moteur d'exécution

Pool et vols

- pool de threads créé lors du premier appel à **rayon**

Pool et vols

- pool de threads créé lors du premier appel à **rayon**
- chaque thread possède un **Deque** de tâches

Pool et vols

- pool de threads créé lors du premier appel à **rayon**
- chaque thread possède un **Deque** de tâches
  - le graphe est distribué
  - le graphe apparait au cours du temps
- **join**
  - on empile la nouvelle tâche en fin de deque

Pool et vols

- pool de threads créé lors du premier appel à **rayon**
- chaque thread possède un **Deque** de tâches
  - le graphe est distribué
  - le graphe apparait au cours du temps
- **join**
  - on empile la nouvelle tâche en fin de deque
- fin de tâche
  - on dépile une tâche en fin de deque
  - ou on **vole** une tâche en début de deque d'une victime aléatoire

Pool et vols

- pool de threads créé lors du premier appel à **rayon**
- chaque thread possède un **Deque** de tâches
  - le graphe est distribué
  - le graphe apparait au cours du temps
- **join**
  - on empile la nouvelle tâche en fin de deque
- fin de tâche
  - on dépile une tâche en fin de deque
  - ou on **vole** une tâche en début de deque d'une victime aléatoire
- excellents temps d'exécutions sur de nombreux graphes (au programme d'algo avancée en 2A)

Plan

Rayon

Itérateurs parallèles

À l'intérieur d'un itérateur parallèle

Itérateurs parallèles

```
use rayon::prelude::*;
(0..1_000_000).into_par_iter().map(f).sum()
```

Itérateurs parallèles

```
use rayon::prelude::*;
(0..1_000_000).into_par_iter().map(f).sum()
```

- rayon fournit des itérateurs parallèles
  - beaucoup de fonctionnalités
  - peu de surcoûts
  - faciles à utiliser
  - une merveille d'informatique

Des traits, encore des traits

- traits du prélude
  - ParallelIterator
  - IndexedParallelIterator
  - IntoParallelIterator
  - FromParallelIterator
  - ParallelSlice
  - ParallelString
  - ...
- différentes fonctionnalités
  - correspondent aux traits classiques
  - étendent les types de base

Produit scalaire

```
fn produit_scalaire(a: &[u32], b: &[u32]) -> u32 {
  a.par_iter().zip(b).map(|(ea, eb)|*ea * *eb).sum()
}
```

Produit scalaire

```
fn produit_scalaire(a: &[u32], b: &[u32]) -> u32 {
  a.par_iter().zip(b).map(|(ea, eb)|*ea * *eb).sum()
}
```

- algorithme parallèle sophistiqué
  - une ligne de rust
  - aucun indice en vue
- compile vers de l'assembleur vectorisé
  - des performances excellentes

ParallelIterator

- version "faible" de l'itérateur

ParallelIterator

- version "faible" de l'itérateur
  - le point de découpe n'est pas controlé
    - par exemple `tranche.par_iter().filter(f)`
    - découpe au milieu de la tranche mais pas des éléments

ParallelIterator

- version "faible" de l'itérateur
  - le point de découpe n'est pas controlé
    - par exemple `tranche.par_iter().filter(f)`
    - découpe au milieu de la tranche mais pas des éléments
- pleins de méthodes néanmoins
  - `for_each, count, map, cloned, copied, inspect,`
  - `filter, filter_map, flat_map, flatten, reduce, fold,`
  - `try_fold, sum, product, min, max, chain, any,`
  - `all, collect, find_any` ...

Reduce

```
    fn reduce<F>(self, f: F) -> Option<Self::Item>
where
    Self: Sized,
    F: FnMut(Self::Item, Self::Item) -> Self::Item,
```

- un **fold** plus simple : l'état est de même type que les éléments de l'itérateur

Reduce

```
    fn reduce<F>(self, f: F) -> Option<Self::Item>
where
    Self: Sized,
    F: FnMut(Self::Item, Self::Item) -> Self::Item,
```

- un **fold** plus simple : l'état est de même type que les éléments de l'itérateur
- version parallèle

Reduce

```
    fn reduce<F>(self, f: F) -> Option<Self::Item>
where
    Self: Sized,
    F: FnMut(Self::Item, Self::Item) -> Self::Item,
```

- un **fold** plus simple : l'état est de même type que les éléments de l'itérateur
- version parallèle
  - l'itérateur parallèle est "coupé" en petits morceaux

Reduce

```
    fn reduce<F>(self, f: F) -> Option<Self::Item>
where
    Self: Sized,
    F: FnMut(Self::Item, Self::Item) -> Self::Item,
```

- un **fold** plus simple : l'état est de même type que les éléments de l'itérateur
- version parallèle
  - l'itérateur parallèle est "coupé" en petits morceaux
  - chaque morceau est "transformé" en itérateur séquentiel

Reduce

```
    fn reduce<F>(self, f: F) -> Option<Self::Item>
where
    Self: Sized,
    F: FnMut(Self::Item, Self::Item) -> Self::Item,
```

- un **fold** plus simple : l'état est de même type que les éléments de l'itérateur
- version parallèle
  - l'itérateur parallèle est "coupé" en petits morceaux
  - chaque morceau est "transformé" en itérateur séquentiel
  - les résultats de chaque morceau sont réduits deux à deux

IndexedParallelIterator

- version "forte" de l'itérateur
- le trait étend **ParallelIterator**

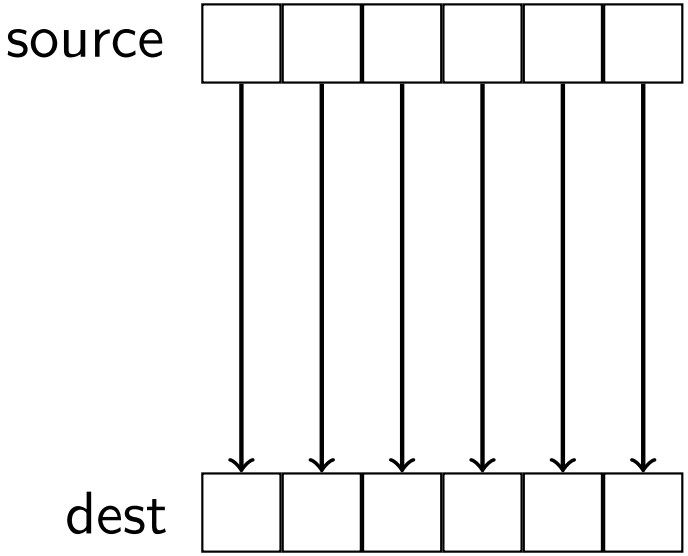
IndexedParallelIterator

- version "forte" de l'itérateur
- le trait étend **ParallelIterator**
- de nouvelles méthodes
  - `zip, skip, take, enumerate, position_any,`
  - `interleave, step_by, rev, eq, chunks, ...`

IndexedParallelIterator

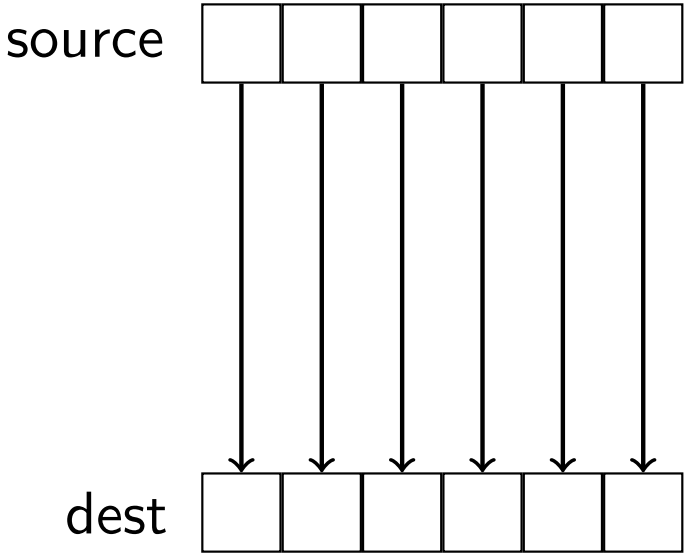
- version "forte" de l'itérateur
- le trait étend **ParallelIterator**
- de nouvelles méthodes
  - `zip, skip, take, enumerate, position_any,`
  - `interleave, step_by, rev, eq, chunks, ...`
- deux méthodes à moi dedans 🙂

Collect (facile)



- facile : à partir d'un itérateur indicé

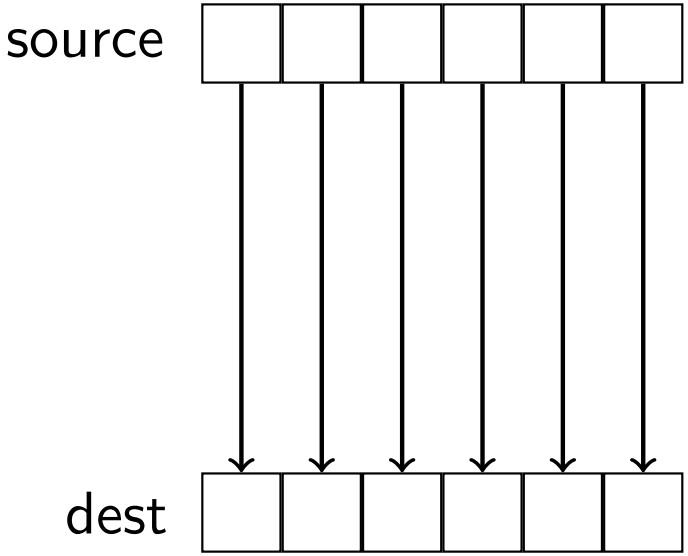
Collect (facile)



- facile : à partir d'un itérateur indicé

```
source.zip(destination.iter_mut())
      .for_each(|(s, d)| *d = s)
```

Collect (facile)

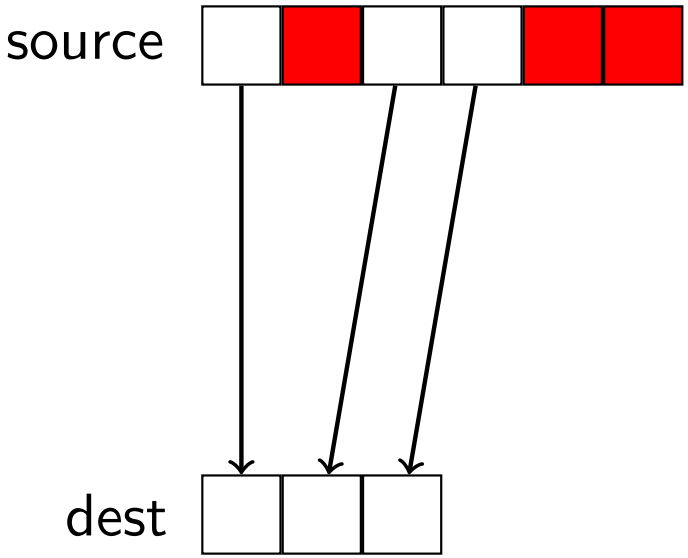


- facile : à partir d'un itérateur indicé

```
source.zip(destination.iter_mut())
      .for_each(|(s, d)| *d = s)
```

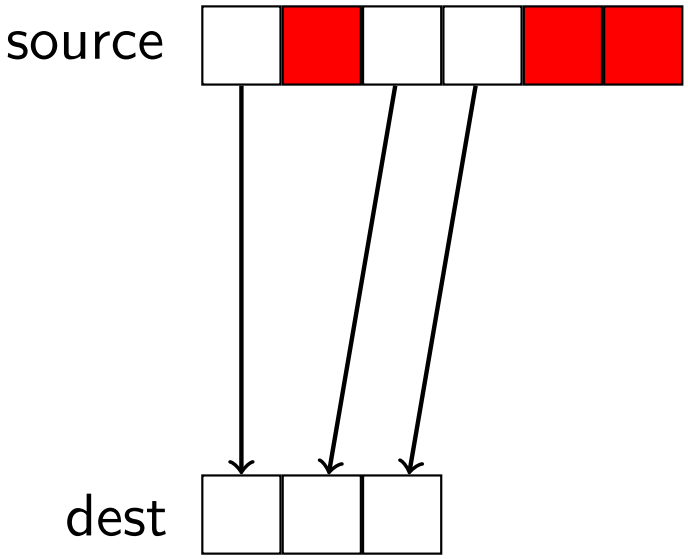
- à ajouter :
  - création de la destination
  - gestion de la mémoire non initialisée

Collect (pas facile)



- pas facile : à partir d'un itérateur pas indicé

Collect (pas facile)



- pas facile : à partir d'un itérateur pas indicé
- par où on commence ?

Collect (pas facile) : for

```
let mut v = Vec::new();
for e in iter {
  if p(&e) {
    v.push(e)
  }
}
v
```

Collect (pas facile) : for

```
let mut v = Vec::new();
for e in iter {
  if p(&e) {
    v.push(e)
  }
}
v
```

```
let mut v = Vec::new();
par_iter.filter(p)
  .for_each(|e| v.push(e));
v
```

Collect (pas facile) : for

```rust
let mut v = Vec::new();
for e in iter {
  if p(&e) {
    v.push(e)
  }
}
v
```

```rust
let mut v = Vec::new();
par_iter.filter(p)
  .for_each(|e| v.push(e));
v
```

```
error[E0596]: cannot borrow `v` as mutable, as it is a captured variable in a `Fn` closure
 --> src/main.rs:6:37
  |
6 |     par_iter.filter(p).for_each(|e| v.push(e));
  |                                     ^ cannot borrow as mutable
```

Évitons la variable mutable

Évitons la variable mutable

```
iter.filter(p).fold(Vec::new(), |mut v, e| {
  v.push(e);
  v
}
```

Évitons la variable mutable

```
iter.filter(p).fold(Vec::new(), |mut v, e| {
  v.push(e);
  v
}
```

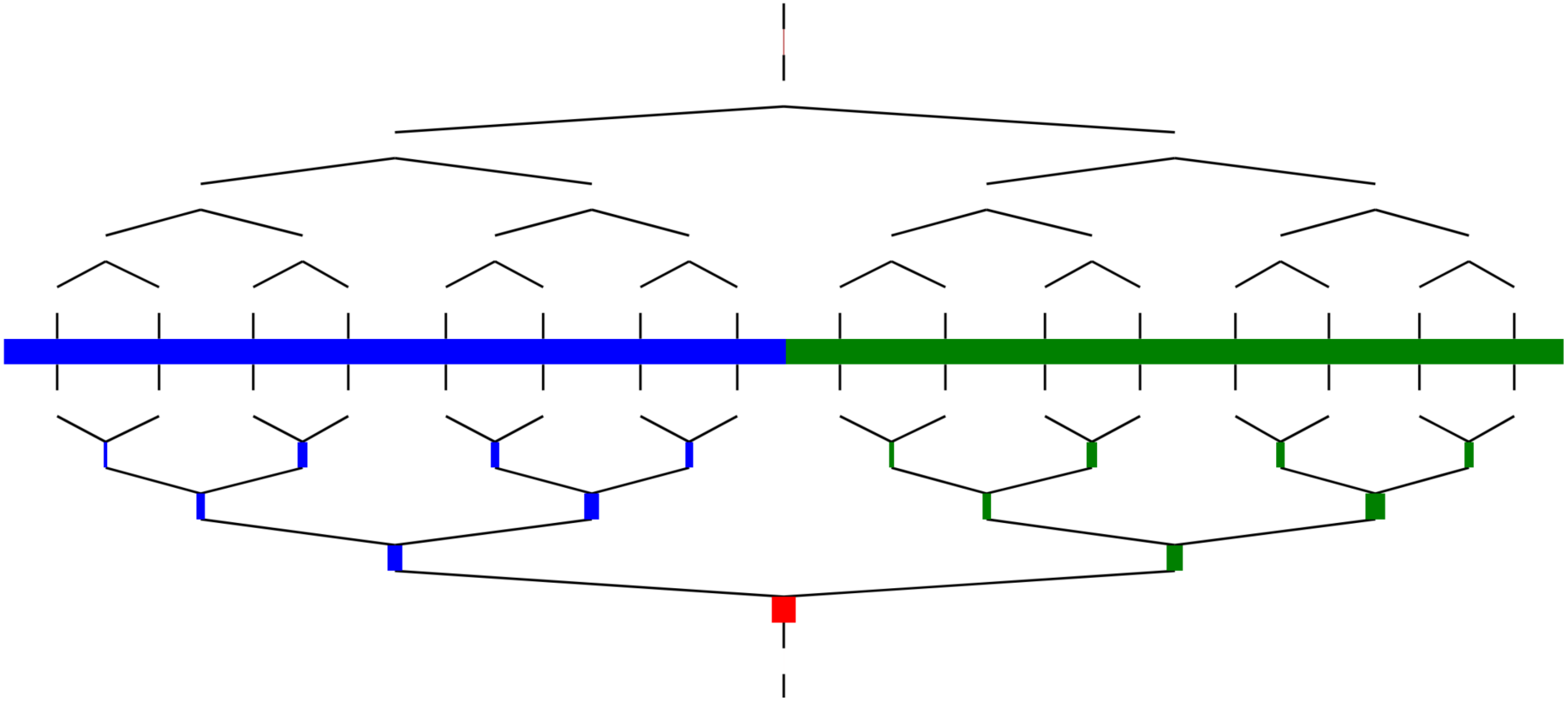
```
fn fold<T, ID, F>(self, identity: ID, fold_op: F)
    -> Fold<Self, ID, F>
where
    F: Fn(T, Self::Item) -> T + Sync + Send,
    ID: Fn() -> T + Sync + Send,
    T: Send,
```

- fold parallèle
  - un état par tâche
  - produit un itérateur parallèle

Collect v1

```
par_iter
  .filter(|e| e % 2 == 0)
  .fold(Vec::new, |mut v, e| {
    v.push(e);
    v
  })
  .reduce(Vec::new, |mut vg, mut vd| {
    vg.append(&mut vd);
    vg
  })
```

Analyse

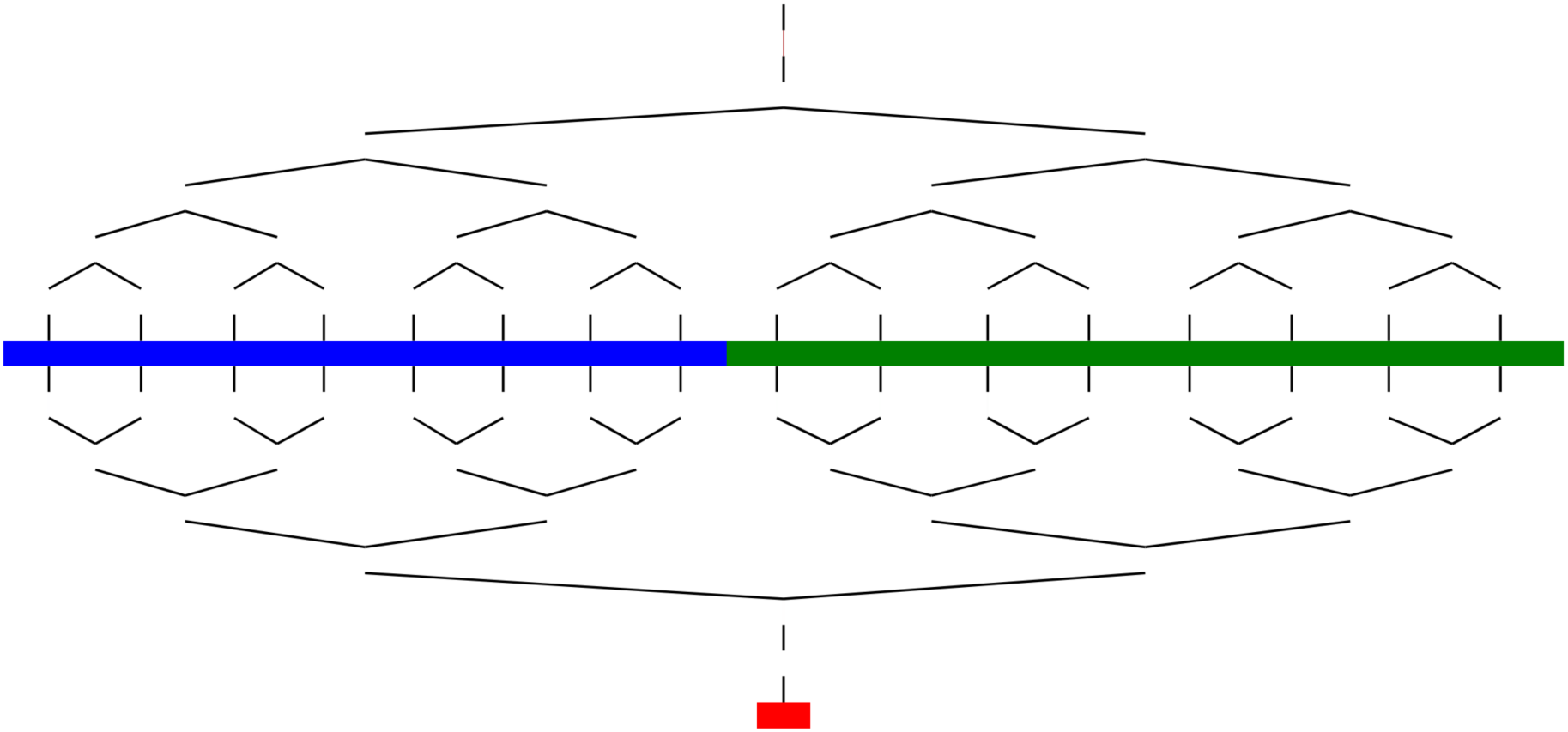


$$W = O(n \log(n))$$

Collect de rayon

```
par_iter
 .filter(|e| e % 2 == 0)
 .fold(Vec::new, |mut v, e| {
   v.push(e);
   v
 })
 .map(|v| std::iter::once(v).collect::<LinkedList<Vec<_>>>())
 .reduce(LinkedList::new, |mut lg, mut ld| {
   lg.append(&mut ld); lg
 })
 .into_iter()
 .reduce(|mut vg, mut vd| {
   vg.append(&mut vd); vg
 })
 .unwrap_or_default()
```

Analyse



$$W = O(n)$$

Plan

Rayon

Itérateurs parallèles

À l'intérieur d'un itérateur parallèle

ParallelIterator

- Producer et Consumer
- bridge
- drive

Pour aller plus loin

- la doc de rayon
- des énigmes parallèles