Rayon

Beautiful Parallelism in Rust

Josh Stone (cuviper)
PDXRust, July 3, 2019

Intro

Rayon is a Rust crate for data parallelism.

Your computer has many cores, and Rust promises “fearless concurrency” -- Rayon provides an easy means to take advantage of this.

Parallel iterators are built on a work-stealing thread pool in rayon-core, with just a few primitive operations.

https://github.com/rayon-rs/rayon
https://crates.io/crates/rayon
https://docs.rs/rayon

"I added 4 characters, and now my code is parallel!"

- Typical Rayon user

Usage

Add a dependency to `Cargo.toml`:

```
[dependencies]
rayon = "1"
```

Import the traits in your source:

```
use rayon::prelude::*;
```

IntoParallelIterator

Item =	std	rayon
T	into_iter()	into_par_iter()
&T	iter()	par_iter()
&mut T	iter_mut()	par_iter_mut()

ParallelSlice

std	rayon
chunks(n)	par_chunks(n)
windows(n)	par_windows(n)
split(f)	par_split(f)

ParallelSliceMut

std	rayon
chunks_mut(n)	par_chunks_mut(n)
split_mut(f)	par_split_mut(f)
sort()	par_sort()
sort_unstable()	par_sort_unstable()

ParallelString

std	rayon
chars()	par_chars()
bytes()	par_bytes()
lines()	par_lines()
split(f)	par_split(f)
split_whitespace()	par_split_whitespace()

Demo: Wasm Parallel Raytracing

```
rgb_data
    .par_chunks_mut(4)
    .enumerate()
    .for_each(|(i, chunk)| {
        let i = i as u32;
        let x = i % width;
        let y = i / width;
        let ray = raytracer::Ray::create_prime(x, y, &scene);
        let result = raytracer::cast_ray(&scene, &ray, 0).to_rgba();
        chunk[0] = result.data[0];
        chunk[1] = result.data[1];
        chunk[2] = result.data[2];
        chunk[3] = result.data[3];
    });
```

https://github.com/rustwasm/wasm-bindgen
examples/raytrace-parallel/

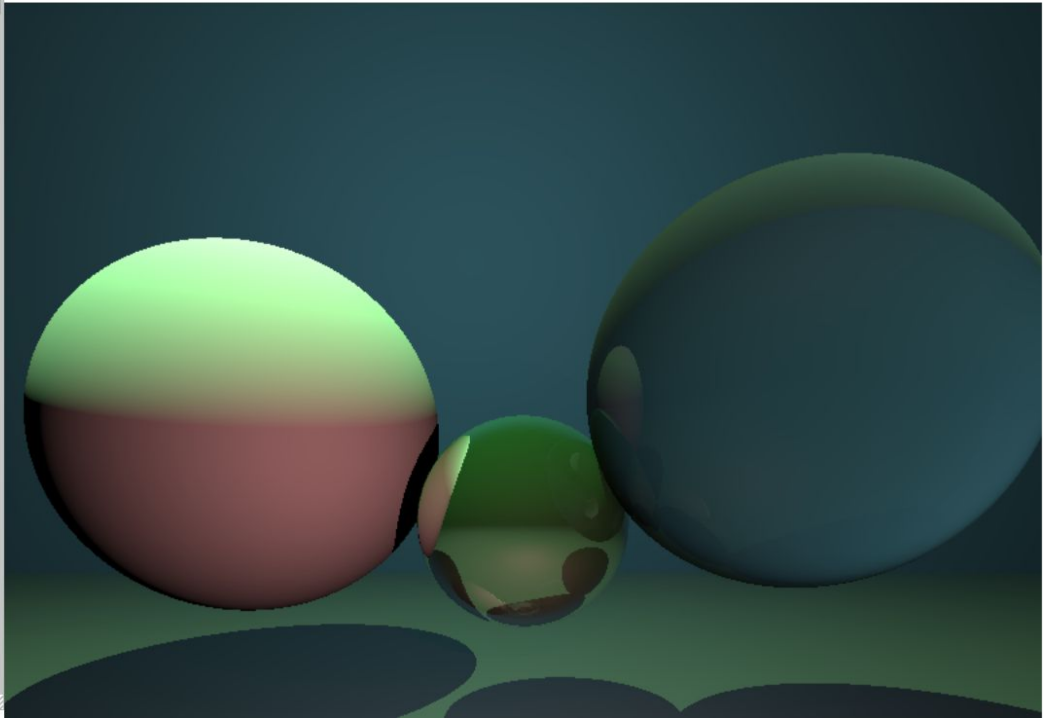
```
{
  "width": 800,
  "height": 800,
  "fov": 90.0,
  "shadow_bias": 1e-13,
  "max_recursion_depth": 20,
  "elements": [
    {
      "Sphere" : {
        "center": {
          "x": 0.0,
          "y": 0.0,
          "z": -5.0
        },
        "radius": 1.0,
        "material": {
          "coloration" : {
            "Color": {
              "red": 0.2,
              "green": 1.0,
              "blue": 0.2
            }
          },
          "albedo": 0.18,
          "surface": {
            "Reflective": {
              "reflectivity": 0.7
            }
          }
        }
      }
    },
    {
      "Sphere" : {
        "center": {
          "x": -3.0,
          "y": 1.0,
          "z": -6.0
        },
        "radius": 2.0,
        "material": {
          "coloration": {
            "Color": {
              "red": 1.0,
              "green": 1.0,
              "blue": 1.0
            }
          },
          "albedo": 0.58,
          "surface": "Diffuse"
```


Render!

Concurrency: 4

Render duration:

785ms



FIN

Enjoy your parallel Rust!

OK

There's more to it…

Compiler magic: `Send + Sync`

Rayon is a regular library crate. The compiler doesn't know anything about it, but can still enforce threading safety with two builtin traits:

- `T: Send` means `T` can move control to another thread (`T` or `&mut T`)
- `T: Sync` means `T` can be shared with another thread (`&T`)

Rayon requires these traits for safety:

- Iterators require `Item: Send`
- Function callbacks are `Fn + Send + Sync`

“Fearless concurrency” is real!

Trust that rustc will stop you if threading requirements are not met

Common problem: mutable state -- your closure is `FnMut`

- Try to refactor to avoid mutable sharing
- Use atomics, mutexes, etc.

Common problem: types aren’t thread-safe, e.g. `Rc`, `Sender`, `ThreadRng`

- Look for alternatives like `Arc`

Compiler magic: `for` loops

```
// Desugar `ExprForLoop`
// from: `[opt_ident]: for <pat> in <head> <body>`
ExprKind::ForLoop(ref pat, ref head, ref body, opt_label) => {
    // to:
    //
    //   {
    //     let result = match ::std::iter::IntoIterator::into_iter(<head>) {
    //       mut iter => {
    //         [opt_ident]: loop {
    //           let mut __next;
    //           match ::std::iter::Iterator::next(&mut iter) {
    //             ::std::option::Option::Some(val) => __next = val,
    //             ::std::option::Option::None => break
    //           };
    //           let <pat> = __next;
    //           StmtKind::Expr(<body>);
    //         }
    //       }
    //     };
    //     result
    //   }
```

The compiler rewrites `for` loops entirely in terms of `Iterator`

No parallelism (à la OpenMP) yet…

Proc-macro rewrites are possible:
https://crates.io/crates/rayon-attr

```
#![feature(stmt_expr_attributes)]
#[parallel]
for x in y { … }
```

Translating `for` loops

```
for x in y { … }

y.into_par_iter().for_each(|x| { … });
```

Sometimes beneficial for sequential iterators too -- internal iteration

`continue` the loop => `return` from the closure

No direct replacement for early returns, nor to break the loop or control outer loops, but consider `try_for_each`

crate rayon

Iterator methods

`ParallelIterator` provides the most general functionality.

`IndexedParallelIterator` provides further methods for iterators that know their exact size and can split at arbitrary points.

These try to match `Iterator` as much as possible.

Iterator => ParallelIterator

```
all(p)
any(p)
chain(iter)
cloned()
collect()
count()
filter(f)
filter_map(f)
flat_map(f)
flatten()
for_each(f)
inspect(f)
```

```
map(f)
max()
max_by(f)
max_by_key(f)
min()
min_by(f)
min_by_key(f)
partition(f)
product()
sum()
try_for_each(f)
unzip()
```

```
find(f) => find_any(f),
  find_first(f), find_last(f)
find_map(f) => find_map_any(f),
  find_map_first(f), find_map_last(f)

fold(init, f)
  => fold(init, f)
   + reduce(init, f)
try_fold(init, f)
  => try_fold(init, f)
   + try_reduce(init, f)
```

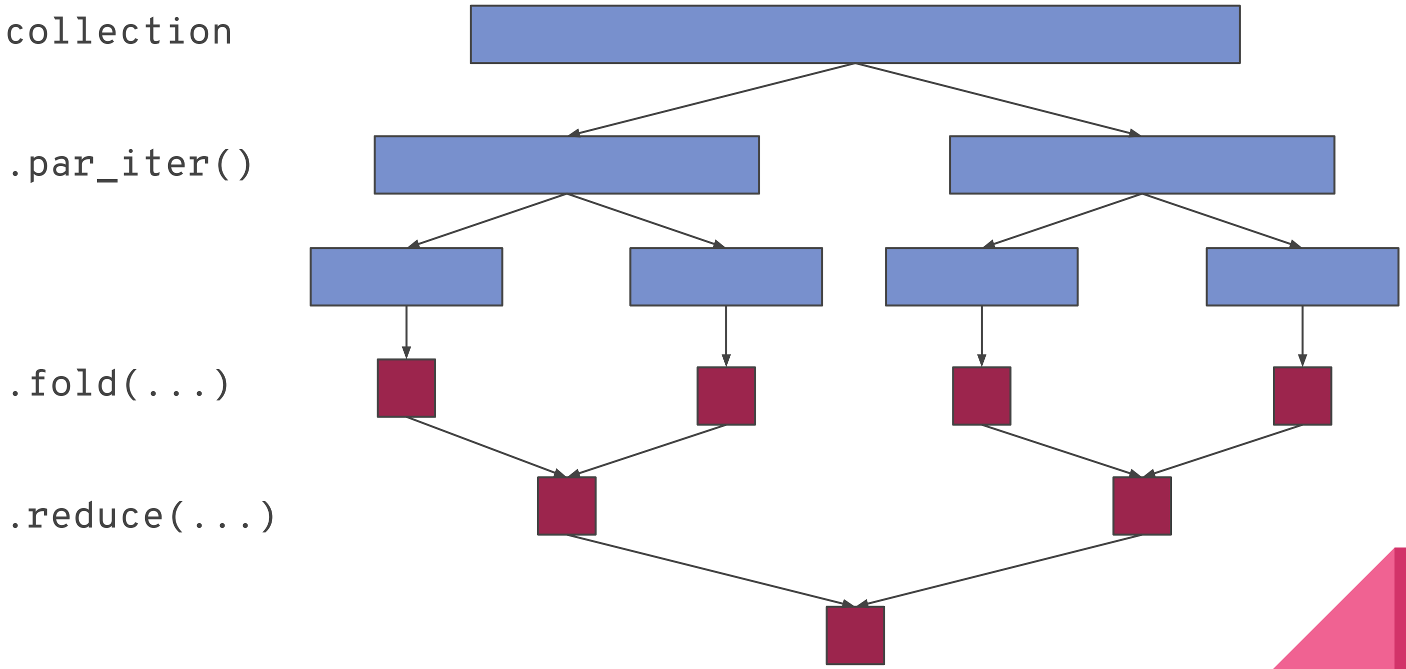
Fold and Reduce

```
fn fold(self, init: B, f: F) -> B
where
    F: FnMut(B, Self::Item) -> B,
```

```
fn fold(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,

fn reduce(self, identity: ID, op: OP) -> Self::Item
where
    OP: Fn(Self::Item, Self::Item) -> Self::Item + Sync + Send,
    ID: Fn() -> Self::Item + Sync + Send,
```

Fold and Reduce



Iterator => IndexedParallelIterator

```
cmp(iter)
enumerate()
eq(iter)
ge(iter)
gt(iter)
le(iter)
lt(iter)
ne(iter)
partial_cmp(iter)
```

```
rev()
skip(n)
take(n)
zip(iter)
zip_eq(iter)
```

```
position(f) => position_any(f),
  position_first(f), position_last(f)
```

Itertools

```
intersperse(x)        chunks(n)
partition_map(f)      interleave(iter)
update(f)             interleave_shortest(iter)
while_some()
```

FromParallelIterator + ParallelExtend

Collecting or extending collections from parallel iterators

```
collect() -> C
unzip() -> (A, B)
partition(pred) -> (A, B)
partition_map(pred) -> (A, B)

collect_into_vec(&mut vec)
unzip_into_vecs(&mut left, &mut right)
```

Short-circuiting `Option<C>` and `Result<C, E>`

Composed `unzip`

```
let (indexes, (squares, cubes)): (Vec<_>, (Vec<_>, Vec<_>))
    = input.par_iter()
           .map(|x| (x * x, x * x * x))
           .enumerate().unzip();
```

Compiler stress test? Here's a nice type:

```
core::option::Option<rayon_core::join::join_context::{{closure}}::{{closure}}<rayon::iter::plumbing::bridge_producer
_consumer::helper::{{closure}}<rayon::iter::enumerate::EnumerateProducer<rayon::iter::map::MapProducer<rayon::slice:
:IterProducer<i32>, main::{{closure}}>>, rayon::iter::unzip::UnzipConsumer<rayon::iter::unzip::Unzip, rayon::iter::f
old::FoldConsumer<rayon::iter::map::MapConsumer<rayon::iter::reduce::ReduceConsumer<rayon::iter::collect::{{impl}}::
par_extend::{{closure}}<usize, rayon::iter::unzip::UnzipA<rayon::iter::enumerate::Enumerate<rayon::iter::map::Map<ra
yon::slice::Iter<i32>, main::{{closure}}>>, rayon::iter::unzip::Unzip, (alloc::vec::Vec<i32>, alloc::vec::Vec<i32>)>
>, fn() -> alloc::collections::linked_list::LinkedList<alloc::vec::Vec<usize>>>, rayon::iter::
collect::{{impl}}::par_extend::{{closure}}<usize, rayon::iter::unzip::UnzipA<rayon::iter::
enumerate::Enumerate<rayon::iter::map::Map<rayon::slice::Iter<i32>, main::{{closure}}>>,
rayon::iter::unzip::Unzip, (alloc::vec::Vec<i32>, alloc::vec::Vec<i32>)>>>,
```

… 132,967 characters!

https://gist.github.com/cuviper/99a38550061f1e3513dfe797639ba3ef

Mutable context

```
fn map_with(self, init: T, map_op: F) -> MapWith
where
    F: Fn(&mut T, Self::Item) -> R + Sync + Send,
    T: Send + Clone,
    R: Send,
```

e.g. T = mpsc::Sender

```
fn map_init(self, init: INIT, map_op: F) -> MapInit
where
    F: Fn(&mut T, Self::Item) -> R + Sync + Send,
    INIT: Fn() -> T + Sync + Send,
    R: Send,
```

e.g. T = ThreadRng

See also `for_each_with` and `for_each_init`

ParallelBridge

Convert any `T: Iterator + Send` to a parallel iterator

```
let reader = BufReader::new(file);
reader.lines().par_bridge().for_each(|line| {
    // process lines in parallel!
});
```

More overhead than a direct parallel iterator

Does not preserve input order

crate rayon-core

What is `rayon-core`?

Low-level interfaces for the common thread pool

Stronger stability intention -- reached 1.0 before `rayon` did, and intended to remain 1.x even if someday `rayon` bumps to 2.0 and beyond

Most of the API is re-exported in `rayon`

ThreadPool

Use the implicit global pool, or create a standalone instance

Customize with a `ThreadPoolBuilder` -- number of threads, stack size, and custom handlers for start/exit/panic

New in 1.5: custom spawn handler (enabled wasm)

New in 1.5: scoped thread pools

`join(fn_a, fn_b)`

Run two closures, possibly in parallel, and return a tuple of their results

```
let (a, b) = rayon::join(
    || compute_a(),
    || compute_b(),
);
```

This is the primary building block of parallel iterators

See also `join_context`

spawn(f)

Run a `'static` closure in the pool, without waiting for it to complete

```
rayon::spawn(|| {
    // do something in the thread pool
})
```

See also `spawn_fifo` (new in 1.5)

scope(f)

Create a scope for spawning closures, possibly non-`'static`, waiting for them to complete before returning from the scope.

```
rayon::scope(|s: &Scope<'_>| {
    …
    s.spawn(|_| do_some_work());
    …
    s.spawn(|_| do_other_work());
    …
}); // waits for completion
```

See also `scope_fifo` (new in 1.5)

3rd-party


crates.io
Rust Package Registry
Click or press 'S' to search...
Back to rayon
Displaying 1-10 of 362 reverse dependencies of rayon

Trait implementations

`hashbrown` - replacement `HashMap` and `HashSet`

`im` - immutable data structures

`ndarray-parallel` - n-dimensional arrays

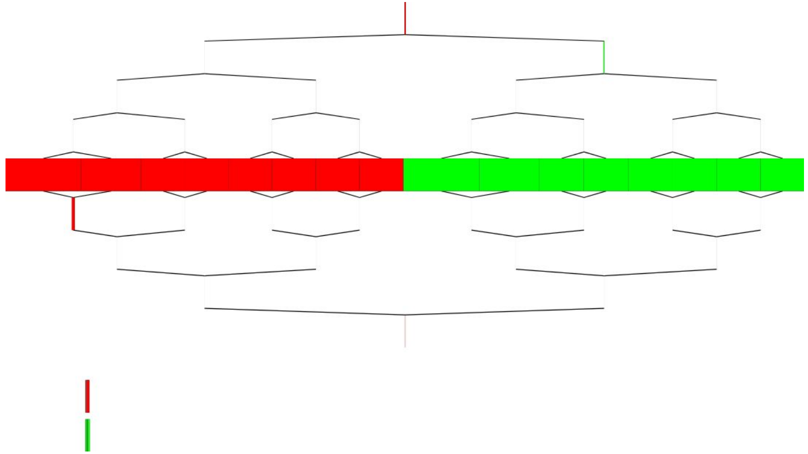
`specs` - **S**pecs **P**arallel **E**ntity-**C**omponent **S**ystem

Extensions

rayon_croissant - `mapfold_reduce_into`

rayon_adaptive - alternative parallel iterator implementation

rayon_logs - execution tracing



Servo

layout/parallel.rs

```
// Spawn a new work unit for each chunk after the first.
let mut chunks = discovered_child_flows.chunks(CHUNK_SIZE);
let first_chunk = chunks.next();
for chunk in chunks {
    let nodes = chunk.iter().cloned().collect::<FlowList>();
    scope.spawn(move |scope| {
        top_down_flow(
            &nodes,
            pool,
            scope,
            &assign_isize_traversal,
            &assign_bsize_traversal,
        );
    });
}
```

style/parallel.rs

```
// In the common case, our children fit within a single work unit, in which
// case we can pass the SmallVec directly and avoid extra allocation.
if nodes.len() <= WORK_UNIT_MAX {
    let work: WorkUnit<E::ConcreteNode> = nodes.collect();
    if may_dispatch_tail {
        top_down_dom(&work, root, traversal_data, scope, pool, traversal, tls);
    } else {
        scope.spawn(move |scope| {
            profiler_label!(Style);
            let work = work;
            top_down_dom(&work, root, traversal_data, scope, pool, traversal, tls);
        });
    }
} else {
    for chunk in nodes.chunks(WORK_UNIT_MAX).into_iter() {
        let nodes: WorkUnit<E::ConcreteNode> = chunk.collect();
        let traversal_data_copy = traversal_data.clone();
        scope.spawn(move |scope| {
            profiler_label!(Style);
            let n = nodes;
            top_down_dom(&*n, root, traversal_data_copy, scope, pool, traversal, tls)
        });
    }
}
```

Rayon

https://github.com/rayon-rs/rayon
https://crates.io/crates/rayon
https://docs.rs/rayon