Rayon

Beautiful Parallelism in Rust

Josh Stone (cuviper)
PDXRust, July 3, 2019
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

<table>
<thead>
<tr>
<th>Item</th>
<th>std</th>
<th>rayon</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>into_iter()</td>
<td>into_par_iter()</td>
</tr>
<tr>
<td>&amp;T</td>
<td>iter()</td>
<td>par_iter()</td>
</tr>
<tr>
<td>&amp;mut T</td>
<td>iter_mut()</td>
<td>par_iter_mut()</td>
</tr>
</tbody>
</table>
## ParallelSlice

<table>
<thead>
<tr>
<th>std</th>
<th>rayon</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunks(n)</td>
<td>par_chunks(n)</td>
</tr>
<tr>
<td>windows(n)</td>
<td>par_windows(n)</td>
</tr>
<tr>
<td>split(f)</td>
<td>par_split(f)</td>
</tr>
</tbody>
</table>
## ParallelSliceMut

<table>
<thead>
<tr>
<th>std</th>
<th>rayon</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunks_mut(n)</td>
<td>par_chunks_mut(n)</td>
</tr>
<tr>
<td>split_mut(f)</td>
<td>par_split_mut(f)</td>
</tr>
<tr>
<td>sort()</td>
<td>par_sort()</td>
</tr>
<tr>
<td>sort_unstable()</td>
<td>par_sort_unstable()</td>
</tr>
</tbody>
</table>
## ParallelString

<table>
<thead>
<tr>
<th>std</th>
<th>rayon</th>
</tr>
</thead>
<tbody>
<tr>
<td>chars()</td>
<td>par_chars()</td>
</tr>
<tr>
<td>bytes()</td>
<td>par_bytes()</td>
</tr>
<tr>
<td>lines()</td>
<td>par_lines()</td>
</tr>
<tr>
<td>split(f)</td>
<td>par_split(f)</td>
</tr>
<tr>
<td>split_whitespace()</td>
<td>par_split_whitespace()</td>
</tr>
</tbody>
</table>
Demo: Wasm Parallel Raytracing

```rust
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_rgb();
        chunk[0] = result.data[0];
        chunk[1] = result.data[1];
        chunk[2] = result.data[2];
        chunk[3] = result.data[3];
    });
```
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:

\[ T : \text{Send} \] means \( T \) can move control to another thread (\( T \) or \&mut \( T \))
\[ T : \text{Sync} \] 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

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

```rust
#![feature(stmt_expr_attributes)]
#![parallel]
for x in y { ... }
```
Translating for loops

```rust
for x in y { ... }

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

Sometimes beneficial for sequential iterators too -- internal iteration

```rust
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`
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)  map(f)
any(p)   max()
chain(iter)  max_by(f)
cloned()  max_by_key(f)
collect()  min()
count()  min_by(f)
filter(f)  min_by_key(f)
filter_map(f)  partition(f)
flat_map(f)  product()
flatten()  sum()
for_each(f)  try_for_each(f)
inspect(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

collection
.par_iter()
.fold(...)
.reduce(...)

Diagram:
- Collection node
  - par_iter() node
    - fold(...) node
      - reduce(...) node
Iterator => IndexedParallelIterator

cmp(iter)    rev()
enumerate()   skip(n)
eq(iter)      take(n)
ge(iter)      zip(iter)
gt(iter)      zip_eq(iter)
le(iter)      
lt(iter)      
ne(iter)      
partial_cmp(iter)

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

intersperse(x)  chunks(n)
partition_map(f)  interleaves(iter)
update(f)  interleaves_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:

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

```rust
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,
```

```rust
e.g. T = mpsc::Sender
```

```rust
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,
```

```rust
e.g. T = ThreadRng
```

See also `for_each_with` and `for_each_init`
ParallelBridge

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

```rust
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
Thread Pool

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

```rust
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

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

See also `spawn_fifo` (new in 1.5)
Create a scope for spawning closures, possibly non-\texttt{static}, waiting for them to complete before returning from the scope.

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

See also \texttt{scope_fifo} (new in 1.5)
3rd-party
Trait implementations

hashbrown - replacement HashMap and HashSet
im - immutable data structures
ndarray-parallel - n-dimensional arrays
specs - Specs Parallel Entity-Component System
Extensions

rayon_croissant - mapfold_reduce_into

rayon_adaptive - alternative parallel iterator implementation

rayon_logs - execution tracing
// 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