Assignment 7: SparkLite

Due: Friday, November 17, 2017 at 4:20pm
Submission deadline: Monday, November 20, 2017 at 4:20pm

After examining the source code for Spark and realizing it's written in Java, you've concluded that the COST of using high-level data analytics frameworks is too great. Subsequently, in this assignment, you will implement SparkLite, a basic data-parallel/streaming library for processing lists at scale (on one machine) using Rust.

Setup

Copying the files is same as always—copy the assignment out of our folder on Rice.

On the Rice machines:

```
cp -r /afs/ir/class/cs242/assignments/assign7 assign7
```

On your local machine:

```
scp -r <SUNetID>@rice.stanford.edu:/afs/ir/class/cs242/assignments/assign7 assign7
```

Requirements

You will turn in these files:

- `src/collect_block.rs`
- `src/collect_stream.rs`
- `src/collect_par.rs`

Prelude: Iterators and trait objects

A core primitive for dealing with data structures in Rust is the `Iterator` trait which defines an interface for things that can be iterated. An iterator in Rust is similar to the iterators we saw in Lua, but with more types.

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

You can read this as: something that is an iterator has a function `next` that takes in the iterator and produces either `Some(something)` or `None` (indicating the end of the iterator, like `nil` in Lua). The type returned is `Self::Item` (akin to `Self::Node` from the last assignment), which represents the thing the iterator is producing. Almost every Rust data structure has an interface for creating iterators, most notably (for us) vectors:

```
let v: Vec<i32> = vec![1, 2, 3];
let it: std::vec::IntoIter<i32> = v.into_iter();
println!("{:?}", it.next()); // Some(1)
println!("{:?}", it.next()); // Some(2)
```

Here, `std::vec::IntoIter<i32>` is a concrete data structure that implements the trait `Iterator<Item = i32>`, i.e. it is an iterator that produces 32-bit integers. It was produced by `into_iter()` which consumes ownership of the object being

iterated. Notably, there are non-consuming iterators:

```
let mut v: Vec<i32> = vec![1, 2, 3];
{
    let mut it: std::slice::Iter<i32> = v.iter();
    let n: Option<&i32> = it.next();
}

{
    let mut it: std::slice::IterMut<i32> = v.iter_mut();
    let n: Option<&mut i32> = it.next();
    *n.unwrap() = 3;
}
```

Here, `iter` and `iter_mut` are iterators that produce *references* to the elements of the collection being iterated instead of the elements themselves. A benefit of iterators is that anything iterable can access the standard library of iterator functions. For example:

```
let v1: Vec<i32> = vec![1, 2, 3];
let v2: Vec<i32> = v1
    .into_iter()
    .map(|x| x * 3)     // applies the function to each element
    .filter(|x| *x <= 6) // removes elements that return false
    .collect();         // turns the iterator into a vector
println!("{:?}", v2);   // [3, 6]
```

In this assignment, you will need to make use of both the iterator abstraction (producing elements one at a time instead of all at once) and potentially as well as the standard library of iterator functions. Additionally, you will need to write code that should be generic over the implementation of the iterator it uses. For example, here's a function that takes in a generic iterator over integers:

```
fn print_next<T: Iterator<Item=i32>>(mut it: T) {
    println!("i32: {:?}", it.next());
}

fn main() {
    let v = vec![1, 2, 3];
    print_next(v.clone().into_iter().map(|x| x + 1)); // prints 2
    print_next(v.into_iter()); // prints 1
}
```

Sometimes, however, you will want to use an iterator where the type of the iterator changes within a function. For example, if we try to write:

```
fn main() {
    let v = vec![1, 2, 3];
    let mut it = v.clone().into_iter().map(|x| x + 1);
    println!("{:?}", it.next());
    it = v.into_iter();
    println!("{:?}", it.next());
}
```

This will fail to compile with the error:

```
error[E0308]: mismatched types
 --> test.rs:5:10
  |
5 |     it = v.into_iter();
  |          ^^^^^^^^^^^^^ expected struct `std::iter::Map`, found struct `std::vec::IntoIter`
  |
  = note: expected type `std::iter::Map<std::vec::IntoIter<_>, ...>`
             found type `std::vec::IntoIter<_>`
```

Instead, we can use a trait object to solve this behavior.

```
fn main() {
    let v = vec![1, 2, 3];
    let mut it: Box<Iterator<Item=i32>> =
        Box::new(v.clone().into_iter().map(|x| x + 1));
    println!("{:?}", it.next());
    it = Box::new(v.into_iter());
    println!("{:?}", it.next());
}
```

This mechanism allows the variable `it` to concretely represent different structures that implement the same interface (this is an example of dynamic dispatch).

SparkLite tutorial

In SparkLite, the only data type is `Vec<f64>`, or a vector of doubles (64-bit floats). All programs take in and produce a single `Vec<f64>`. A program is described by pipelines of operations, or `Op`s. In our simple language, there are three ops defined in `op.rs`:

```
pub enum Op {
    Map(Arc<(Fn(f64) -> f64) + Send + Sync>),
    Filter(Arc<(Fn(&f64) -> bool) + Send + Sync>),
    GroupBy(
        Arc<(Fn(f64) -> i64) + Send + Sync>,
        Arc<(Fn(&Vec<f64>) -> f64) + Send + Sync>
    )
}
```

> Note: `Fn` describes the `Fn` trait, which you can think about like a function pointer. So `Fn(f64) -> f64` means "a function that takes an `f64` and returns an `f64`". The `Send + Sync` and `Arc` allow us to share functions across thread boundaries.

The `Map` and `Filter` ops are akin to `List.map` and `List.filter` that you've seen in OCaml, i.e. map transforms each element independently and filter eliminates elements according to a predicate. For example, here's a program that multiplies each element of an input by 3 and then filters out negative numbers.

```
let ops = vec![
    Op::Map(Arc::new(|x| x * 3.0)),
    Op::Filter(Arc::new(|x| x > 0))
];
```

Here, a program is a vector of ops (`Vec<Op>`), which correspond to a sequence of operations to be applied to an input list. Map and filter are trivially parallel, but the third operator `GroupBy` less so. `GroupBy` takes two functions, a `key` that determines for each element in the input a "bucket" index (`i64`), and a function `group` that takes in a bucket at a time (`Vec<f64>`) and produces one output (`f64`). For example, to count the number of positive and negative elements in the input:

```
Op::GroupBy(
    Arc::new(|x| if x < 0 { 0 } else { 1 }),
    Arc::new(|xs| xs.len()))
```

This operator assigns a 0 to each negative element and a 1 to each positive element. Then the grouping function will run twice over elements in the 0 bucket and those in the 1 bucket, returning the size of each bucket to produce a final output vector with two `f64`.

> Note: The output of a GroupBy should be sorted in order of the keys, e.g. the result for bucket 0 precedes the result for bucket 1 in the output list. Note: The inputs to GroupBy should be processed in the order that they came in. You shouldn't be modifying the order of the input vector.

Lastly, to run a pipeline over some data, we have a series of `collect_*` functions with the following signature:

```
pub fn collect<T: Iterator<Item = f64>>(source: T, ops: Vec<Op>) -> Vec<f64>
```

This function takes as input some `source`, an iterator that produces `f64`, and `ops`, the list of vector processing operations, and produces a final `Vec<f64>` with the results. For example:

```
let ops = vec![
    Op::Map(Arc::new(|x| x * 3.0)),
    Op::Filter(Arc::new(|x| x > 0))
];
assert_eq!(collect(vec![4.0, -2.0, 1.0].into_iter(), ops), &[12.0, 3.0]);
```

Part 1: Single-thread (25%)

Your first task is to create a single-threaded implementation of the SparkLite runtime. Specifically you will implement the simplest possible version in `collect_block` and optimize it in `collect_stream`.

1a: `collect_block`

The naive implementation of a SparkLite runtime starts by bringing the entirety of the input into memory (fully *materializing* it), and then iteratively applying each op in the pipeline, generating the complete output per-stage. We have provided you a skeleton of this solution in `collect_block.rs`.

Our skeleton solution maintains a vector `v` containing the input data at each stage. Initially this is the `source`, and then `v` changes after applying each operation, getting returned at the end. Your task is to implement each operation (`Map`, `Filter`, and `GroupBy`) however you'd like. You will want to use a `for` loop or an `Iterator` (you may use the built-in `map` and `filter` functions).

1b: `collect_stream`

A major issue with the `collect_block` implementation is that it requires each stage of the pipeline to fully materialize the entire vector into memory. For example, consider the following pipeline:

```
let v = (0..(1024i64 * 1024 * 1024 * 4)).into_iter().map(|x| x as f64);
let ops = vec![Op::Filter(Arc::new(|x| false))];

collect_stream::collect_stream(v, ops);
```

Our `collect_block` would create the entire `v` vector with 32 GB of data in memory. However, an optimized implementation here would not need to do that. In this simple example, we filter out every single element, so the output would be empty. And the input would only need to be materialized an element at a time, instead of getting the whole thing at once.

More generally, a vector only needs to be fully materialized if it's the input to a `GroupBy` or at the end of the pipeline. You will adapt your previous implementation into a *streaming* solution in `collect_stream`, i.e. a single-threaded implementation that respects this constraint and only materializes when necessary.

To get you started, we have again provided a skeleton solution in `collect_stream.rs`. The major difference is that the type of the accumulated variable has changed from `Vec<f64>` to `Box<Iterator<Item = f64>>`, which reads as: "a pointer to something that will produce `f64`s". The idea is that now each stage returns an *iterator* instead of a vector, and that iterator is only `collect`ed (or materialized) when necessary.

Testing

We have provided you a suite of correctness tests in `lib.rs` which you can execute by running:

```
cargo test --release
```

The `--release` tells the Rust compiler to compile with optimizations enabled. Note that you may want to comment out the `correctness!` calls that use functions you have not yet implemented.

Part 2: Multi-thread (75%)

Now that you can successfully deal with one thread, it's time to use all the threads! Specifically, your task is to implement the `collect_par` function that performs vector operations in parallel where possible. This time, we have not provided you any skeleton solution—how to parallelize is entirely up to you. Here's a few considerations for your solution.

- While `Map` and `Filter` are easily parallelizable, `GroupBy` represents a synchronization point. One parallelization strategy is to batch together groups of maps and filters and execute them in parallel, only synchronizing when hitting a `GroupBy` or the end of the pipeline.
- When creating multiple threads to run in parallel, you should run `NUM_WORKERS` threads and assume it will be enough to saturate all the cores on the benchmark machine.
- The simplest way to allocate work to threads is to use a *static* allocation strategy, e.g. if you're running a map with 100 elements and 5 threads, then assign elements 0..19 to thread 0, 20..39 to thread 1, and so on. However, bear in mind some tests (like "imbalanced") will suffer under a static assignment.
- Memory allocations can be expensive, so make sure you aren't doing any unnecessary clones or copies.
- You do not need to worry about datasets larger than memory for the parallel collect (i.e. it's alright to fully materialize inputs and outputs where necessary).

Benchmarks

This portion of the assignment will be graded not just on correctness, but also performance. We have provided you a moderately optimized reference solution which we will compare your solution against. Your goal is to get within a reasonable fraction of the reference solution on all the benchmarks.

You can view the benchmarks in `src/bin/bench.rs`. There are five benchmarks, named `Basic`, `HighCompute`, `Imbalanced`, `ManyGroups`, and `ManyMaps`. Each benchmark defines a pipeline, and that pipeline is run over a vector of 64 million doubles. Performance is measured as end-to-end latency of executing the pipeline on the input vector. To run the benchmarks:

```
python3 grading.py
```

This will tell you how your implementation compares to the reference and provides you a score for this section. (Note: this is a largely accurate indicator of the actual grade you will receive, but we reserve the right to add additional benchmarks.) We will evaluate your solution on Rice, so run the benchmarks there for an accurate performance profile. Additionally, you can increase the number of iterations run to average out variance:

```
python grading.py -i 3
```

Lastly, you can run individual tests by providing the name of the test (one of the five listed above):

```
python grading.py -t HighCompute
```

If you want to run just your own solution, you can run the benchmark binary without the Python wrapper:

```
cargo run --release -- All
```

Submitting

Once you have completed the assignment, upload the files in your copy of `assign7` to Rice. To submit, navigate to the assignment root and execute:

```
python /afs/ir/class/cs242/bin/submit.py 7
```