built on 2023/03/20 at 11:27:07 ***due:*** *mon apr 3 @ 11:59pm*

This assignment aims to give you more practice writing and reasoning about shared-memory parallel code. We're providing some starter files; please *download the starter code from the course website.*

You will zip all your work in one zip file and upload it to Canvas before the due date. **Make sure to `cargo clean` before creating your zip. Rust builds are bloated and will make your TAs unhappy.**

- You can define as many helper functions as necessary.
- You are going to be graded on style as well as on correctness.
- Test your code! Write your own tests (like we did in the previous assignment).

Cargo Tips

Running Tests. Use `cargo test` to run all the unit tests. Check out provided tests.

Cleaning Up Garbage. Use `cargo clean` to clean up files resulting from your various builds.

Code Formatting. Use `cargo fmt` to auto-format all your source files.

Code Linting. Use `cargo clippy` for suggestions on how to write more idiomatic Rust code.

Task 1: Fibonacci Sequence (8 points)

For this task, save your code in `fib/mod.rs`

Remember that the Fibonacci sequence is $f_1 = f_2 = 1$ and $f_{n+2} = f_{n+1} + f_n$. Now here is a neat math trick. If we define

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

then

$$A \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a+b \\ a \end{pmatrix}.$$

This means,

$$A \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} f_2 + f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} f_3 \\ f_2 \end{pmatrix} \qquad A \begin{pmatrix} f_3 \\ f_2 \end{pmatrix} = \begin{pmatrix} f_3 + f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} f_4 \\ f_3 \end{pmatrix} \qquad \cdots \qquad A \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} f_{n+2} \\ f_{n+1} \end{pmatrix}.$$

In general, this means for $k \geq 0$,

$$A^k \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} f_{k+2} \\ f_{k+1} \end{pmatrix},$$

which can be easily shown via induction. Because matrix multiplication is associative, we'll use prefix scan to construct a Fibonacci sequence in parallel.

Your Task: First, you'll implement a function

```
pub fn par_fib_seq(n: u32) -> Vec<num_bigint::BigUint>
```

that returns the sequence $[f_1, f_2, f_3, \ldots, f_n]$. We'll pretend that big num addition is a $O(1)$ work/span operation. Under this assumption, we expect your code to run in $O(n)$ work and $O(\log^2 n)$ span.

After that, in a block comment in the same file, you'll derive the work and span bounds of your code. Provide sufficient reasoning.

(*Hint:* You can easily represent a 2x2 matrix as a tuple of size 4 and write your own multiply function.)

Task 2: Filter (8 points)

For this task, save your code in `filter/mod.rs`

The `filter` function, like in the standard library, takes as input a sequence and a predicate function `p`, and creates a new sequence retaining only the elements that the predicate `p` returns true.

Write a function

```
pub fn par_filter<F>(xs: &[i32], p: F) -> Vec<i32>
where F: Fn(i32) -> bool + Send + Sync
```

that returns a new vector containing only the elements of `xs` for which `p` returns true. Your code will make use of prefix sum and should run in $O(n)$ work and $O(\log^2 n)$ span, where n is the length of the input array slice.

As we saw in class, the offsets to store the selected element can be computed in parallel using a prefix sum computation; however, there is one more obstacle—writing into the respective offsets in parallel. Two compelling options are available:

- Option 1: Since we know the number of selected elements, we can allocate a vector of exactly that size and resort to `UnsafeSlice` to allow the writing to happen concurrently in parallel. The code below allocates *without zeroing out* an output vector.

  ```
  let output = Vec::with_capacity(output_len);
  unsafe { output.set_len(output_len); }
  ```

- Option 2: We can create the output vector like the above and proceed to recursively split the relevant vectors using `.split` and `.split_mut`. This option doesn't require **`unsafe`** but makes the code rather messy.

Task 3: Ontime Performance (16 points)

For this task, save your code in `ontime/mod.rs`

You're given a data file containing all flight records for one year. An example for year 2008 is available for download at

`https://cs.muic.mahidol.ac.th/~ktangwon/2008.csv.zip`

This file has been zip compressed, so you should uncompress it before use. The input file (e.g., `2008.csv`) is comma-separated and the first line contains the field headers. The fields are explained here `https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/HG7NV7`

You will write a function that will give us a glimpse into this rich dataset. Ultimately, you will implement a function

```
pub fn ontime_rank(filename: &str) -> Result<Vec<(String, f64)>, io::Error>
```

that takes in a path/filename (e.g., `2008.csv`) and returns a vector of (airline code, on-time percentage), sorted from best to worst on-time percentage. A flight is on-time if the `ArrDelay` field is a negative number, 0, or not a number. The field `UniqueCarrier` provides the airline code. The on-time percentage is calculated as the percentage of on-time flights compared to all flights using that carrier code. Your percentage number is a floating-point number between 0 and 100 (inclusive).

You will **use parallelism** to make this computation fast. At minimum, you will parallelize the steps where the file is split into lines, each line is split into comma-separated fields, and the fields are parsed.

Subtask I: To this send, you will begin by writing a function

```
pub fn par_split<'a>(st_buf: &'a str, split_char: char) -> Vec<&'a str>
```

that takes in a string slice and a character `split_char`, and returns a vector of string slices resulting from splitting `st_buf` where `split_char` is found. As an example:

```
let demo = "a,hhh,ab,hello,world,,meh";
par_split(demo) // ==> ["a", "hhh", "ab", "hello", "world", "", "meh"]
```

Two details are worth pointing out: First, the string slices returned are slices into the original string (not a new string). Second, the `'a` annotation is called lifetime annotation. This makes it possible to return a vector of string slices into `st_buf` because that annotation says essentially the slices in the output obtain the lifetime from the slice in `st_buf`.

Your `par_split` code should run in $O(n)$ work and at most $O(\log^2 n)$ span, where n is the length of the string.

Remarks: For `par_split` to meet the cost bounds, your implementation can't just use the built-in `.par_split`. You're expected to determine the split points yourself, perhaps using `.map` in `par_iter`. Your `filter` implementation may come in handy after that.

Subtask II: Write a function

```
fn parse_line(line: &str) -> Option<FlightRecord>
```

that turns each line into a `FlightRecord` struct, defined as

```
struct FlightRecord {
    unique_carrier: String,
    actual_elapsed_time: i32,
    arrival_delay: i32,
}
```

where the fields map directly the fields with the same name (though with a different casing style). For integer valued fields, if parsing fails, store 0 in that field. If parsing fails otherwise (e.g., with insufficient number of fields), return `None`. This function is expected to be parallel by taking advantage of `par_split` and parallel iterators.

Subtask III: Complete the implementation of `ontime_rank`. Using the above functions, your implementation is expected to derive parallelism from splitting the input into lines in parallel and splitting each line into fields in parallel. In addition, there are further parallelization opportunities: Computing the on-time statistics is similar to computing a histogram. This, too, can be parallelized. There are at least two options to try:

- Option 1: Use a parallel sorting routine and work from there, or
- Option 2: Use a concurrent hash map so you can do "group by" in parallel. If you're going with this option, you might find `chashmap` useful.

Below is an example of how to read a file completely into a string:

```
use std::fs;
// read the whole file into a string
let file_contents = fs::read_to_string(filename)?;
```

Task 4: Parallel Closest Pairs (16 points)

For this task, save your code in `cp/mod.rs`

You will implement the parallel closest pair algorithm from class. Recall that the distance between two points (x_1, y_1) and (x_2, y_2) is

$$\text{dist} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Notice that this can pose an implementation challenge because the square-root function can turn a whole number into a floating-point number, which is more difficult to work with. Instead, here is a trick: work directly with the square of distance—i.e., use the number before taking the square root. As an example:

> The distance between $(2,5)$ and $(4,10)$ is
>
> $$\sqrt{(2-5)^2 + (5-10)^2} = \sqrt{3^2 + 5^2} = \sqrt{34}$$
>
> and the square of this distance is 34.

Your Task: Write a function

```
pub fn par_closest_distance(points: &[(i32, i32)]) -> i64
```

that takes as input an array slice of points (represented as a pair of `i32`) and returns a single number representing the square of distance between the closest pair of points in the input. Your code must run in $O(n \log n)$ work and at most $O(\log^4 n)$ span.

(*Hints:* Write helper functions.)

Task 5: Extra-Credit: Parallel Sudoku Solver (8 points)

For this task, save your code in `sudoku/mod.rs`

Peter Norvig is the world's top expert on AI. Years ago he wrote a short program Sudoku solver in Python capable of solving any Sudoku puzzle blazing fast. He describes his solution in this essay:

> `https://norvig.com/sudoku.html`

There are two main ideas in his code:

- **Constraint propagation:** What has to go into a particular cell is already completely dictated by existing entries. In this case, one would just put the right number into that cell without resorting to trial and error.
- **Search:** When a cell can take on several possibilities and isn't yet constrained by existing entries, we will try all the numbers that are possible and see which one pans out. This is akin to searching in a (conceptual) tree.

You will reimplement his solution in Rust and add the following bit of your own: *When a cell can take on several possibilities, we'll try all possible numbers in parallel.*

Specifically, you will write a function

```
pub fn solve(board: &str) -> Vec<String>
```

that takes as input a board string (the format is described below) and returns all solutions to this Sudoku puzzle. (*Hints:* Use lots of helper functions. The `bit-set` crate is extremely useful for keeping the "viable" options for each cell; it's much more compact and efficient than a normal set for this setup.)

Input/Output Format. We'll adopt the texual representation that Norvig uses. In this format, a board configuration is represented by a string of length 81, making up of only `.` and the digits `0`–`9`. For example, the string

```
"4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......"
```

represents the board

```
4 . . |. . . |8 . 5
. 3 . |. . . |. . .
. . . |7 . . |. . .
------+------+------
. 2 . |. . . |. 6 .
. . . |. 8 . |4 . .
. . . |. 1 . |. . .
------+------+------
. . . |6 . 3 |. 7 .
5 . . |2 . . |. . .
1 . 4 |. . . |. . .
```