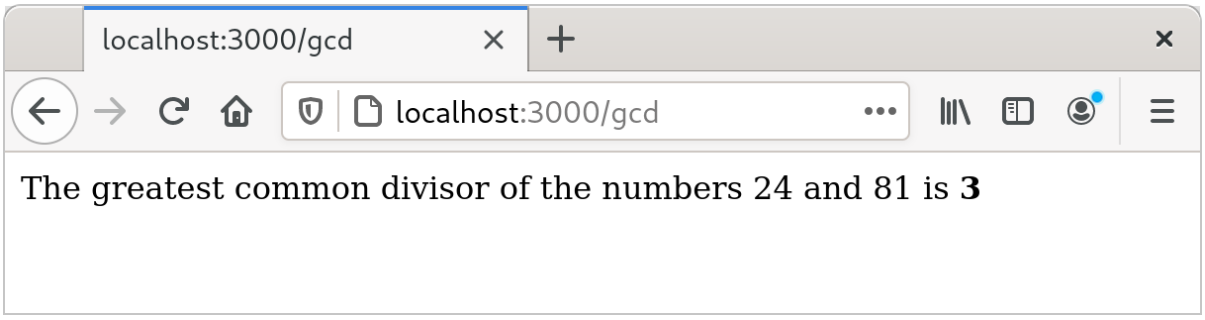

Figure 2-2. Web page showing results of computing GCD

Concurrency

One of Rust's great strengths is its support for concurrent programming. The same rules that ensure Rust programs are free of memory errors also ensure threads can share memory only in ways that avoid data races. For example:

- If you use a mutex to coordinate threads making changes to a shared data structure, Rust ensures that you can't access the data except when you're holding the lock, and releases the lock automatically when you're done. In C and C++, the relationship between a mutex and the data it protects is left to the comments.
- If you want to share read-only data among several threads, Rust ensures that you cannot modify the data accidentally. In C and C++, the type system can help with this, but it's easy to get it wrong.
- If you transfer ownership of a data structure from one thread to another, Rust makes sure you have indeed relinquished all access to it. In C and C++, it's up to you to check that nothing on the sending thread will ever touch the data again. If you don't get it right, the effects can depend on what happens to be in the processor's cache and how many writes to memory you've done recently. Not that we're bitter.

In this section, we'll walk you through the process of writing your second multi-threaded program.

You've already written your first: the Actix web framework you used to implement the Greatest Common Divisor server uses a pool of threads to run request handler functions. If the server receives simultaneous requests, it may run the `get_form` and `post_gcd` functions in several threads at once. That may come as a bit of a shock, since we certainly didn't have concurrency in mind when we wrote those functions. But Rust guarantees this is safe to do, no matter how elaborate your server gets: if your program compiles, it is free of data races. All Rust functions are thread-safe.

This section's program plots the Mandelbrot set, a fractal produced by iterating a simple function on complex numbers. Plotting the Mandelbrot set is often called an *embarrassingly parallel* algorithm, because the pattern of communication between the threads is so simple; we'll cover more complex patterns in Chapter 19, but this task demonstrates some of the essentials.

To get started, we'll create a fresh Rust project:

```
$ cargo new mandelbrot
     Created binary (application) `mandelbrot` package
```

```
$ cd mandelbrot
```

All the code will go in *mandelbrot/src/main.rs*, and we'll add some dependencies to *mandelbrot/Cargo.toml*.

Before we get into the concurrent Mandelbrot implementation, we need to describe the computation we're going to perform.

What the Mandelbrot Set Actually Is

When reading code, it's helpful to have a concrete idea of what it's trying to do, so let's take a short excursion into some pure mathematics. We'll start with a simple case and then add complicating details until we arrive at the calculation at the heart of the Mandelbrot set.

Here's an infinite loop, written using Rust's dedicated syntax for that, a `loop` statement:

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

In real life, Rust can see that `x` is never used for anything and so might not bother computing its value. But for the time being, assume the code runs as written. What happens to the value of `x`? Squaring any number smaller than 1 makes it smaller, so it approaches zero; squaring 1 yields 1; squaring a number larger than 1 makes it larger, so it approaches infinity; and squaring a negative number makes it positive, after which it behaves like one of the prior cases (Figure 2-3).

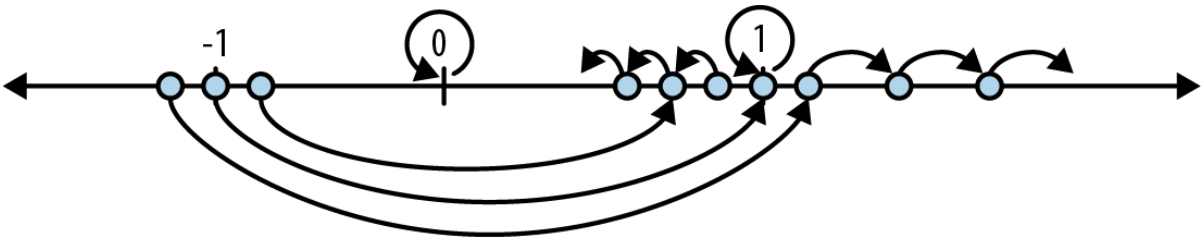

Figure 2-3. Effects of repeatedly squaring a number

So depending on the value you pass to `square_loop`, `x` stays at either zero or one, approaches zero, or approaches infinity.

Now consider a slightly different loop:

```
fn square_add_loop(c: f64) {
    let mut x = 0.;
    loop {
        x = x * x + c;
    }
}
```

This time, `x` starts at zero, and we tweak its progress in each iteration by adding in `c` after squaring it. This makes it harder to see how `x` fares, but some experimentation shows that if `c` is greater than 0.25 or less than –2.0, then `x` eventually becomes infinitely large; otherwise, it stays somewhere in the neighborhood of zero.

The next wrinkle: instead of using `f64` values, consider the same loop using complex numbers. The `num` crate on crates.io provides a complex number type we can use, so we must add a line for `num` to the `[dependencies]` section in our program's *Cargo.toml* file. Here's the entire file, up to this point (we'll be adding more later):

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
num = "0.4"
```

Now we can write the penultimate version of our loop:

```
use num::Complex;

fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
    loop {
        z = z * z + c;
    }
}
```

It's traditional to use `z` for complex numbers, so we've renamed our looping variable. The expression `Complex { re: 0.0, im: 0.0 }` is the way we write complex zero using the `num` crate's `Complex` type. `Complex` is a Rust structure type (or *struct*), defined like this:

```
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

The preceding code defines a struct named `Complex`, with two fields, `re` and `im`. `Complex` is a *generic* structure: you can read the `<T>` after the type name as "for any type `T`." For example, `Complex<f64>` is a complex number whose `re` and `im` fields are `f64` values, `Complex<f32>` would use 32-bit floats, and so on. Given this definition, an expression like `Complex { re: 0.24, im: 0.3 }` produces a `Complex` value with its `re` field initialized to 0.24, and its `im` field initialized to 0.3.

The `num` crate arranges for `*`, `+`, and other arithmetic operators to work on `Complex` values, so the rest of the function works just like the prior version, except that it operates on points on the complex plane, not just points along the real number line. We'll explain how you can make Rust's operators work with your own types in Chapter 12.

Finally, we've reached the destination of our pure math excursion. The Mandelbrot set is defined as the set of complex numbers `c` for which `z` does not fly out to infinity. Our original simple squaring loop was predictable enough: any number greater than 1 or less than –1 flies away. Throwing a `+ c` into each iteration makes the behavior a little harder to anticipate: as we said earlier, values of `c` greater than 0.25 or less than –2 cause `z` to fly away. But expanding the game to complex numbers produces truly bizarre and beautiful patterns, which are what we want to plot.

Since a complex number `c` has both real and imaginary components `c.re` and `c.im`, we'll treat these as the `x` and `y` coordinates of a point on the Cartesian plane, and color the point black if `c` is in the Mandelbrot set, or a lighter color otherwise. So for each pixel in our image, we must run the preceding loop on the corresponding point on the complex plane, see whether it escapes to infinity or orbits around the origin forever, and color it accordingly.

The infinite loop takes a while to run, but there are two tricks for the impatient. First, if we give up on running the loop forever and just try some limited number of iterations, it turns out that we still get a decent approximation of the set. How many iterations we need depends on how precisely we want to plot the boundary. Second, it's been shown that, if `z` ever once leaves the circle of radius 2 centered at the origin, it will definitely fly infinitely far away from the origin eventually. So here's the final version of our loop, and the heart of our program:

```
use num::Complex;

/// Try to determine if `c` is in the Mandelbrot set, using at most `limit`
/// iterations to decide.
///
/// If `c` is not a member, return `Some(i)`, where `i` is the number of
/// iterations it took for `c` to leave the circle of radius 2 centered on the
/// origin. If `c` seems to be a member (more precisely, if we reached the
/// iteration limit without being able to prove that `c` is not a member),
/// return `None`.
fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
        z = z * z + c;
    }

    None
}
```

This function takes the complex number `c` that we want to test for membership in the Mandelbrot set and a limit on the number of iterations to try before giving up and declaring `c` to probably be a member.

The function's return value is an `Option<usize>`. Rust's standard library defines the `Option` type as follows:

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` is an *enumerated type*, often called an *enum*, because its definition enumerates several variants that a value of this type could be: for any type `T`, a value of type `Option<T>` is either `Some(v)`, where `v` is a value of type `T`, or `None`, indicating no `T` value is available. Like the `Complex` type we discussed earlier, `Option` is a generic type: you can use `Option<T>` to represent an optional value of any type `T` you like.

In our case, `escape_time` returns an `Option<usize>` to indicate whether `c` is in the Mandelbrot set—and if it's not, how long we had to iterate to find that out. If `c` is not in the set, `escape_time` returns `Some(i)`, where `i` is the number of the iteration at which `z` left the circle of radius 2. Otherwise, `c` is apparently in the set, and `escape_time` returns `None`.

```
for i in 0..limit {
```

The earlier examples showed `for` loops iterating over command-line arguments and vector elements; this `for` loop simply iterates over the range of integers starting with `0` and up to (but not including) `limit`.

The `z.norm_sqr()` method call returns the square of `z`'s distance from the origin. To decide whether `z` has left the circle of radius 2, instead of computing a square root, we just compare the squared distance with 4.0, which is faster.

You may have noticed that we use `///` to mark the comment lines above the function definition; the comments above the members of the `Complex` structure start with `///` as well. These are *documentation comments*; the `rustdoc` utility knows how to parse them, together with the code they describe, and produce online documentation. The documentation for Rust's standard library is written in this form. We describe documentation comments in detail in Chapter 8.

The rest of the program is concerned with deciding which portion of the set to plot at what resolution and distributing the work across several threads to speed up the calculation.

Parsing Pair Command-Line Arguments

The program takes several command-line arguments controlling the resolution of the image we'll write and the portion of the Mandelbrot set the image shows. Since these command-line arguments all follow a common form, here's a function to parse them:

```
use std::str::FromStr;

/// Parse the string `s` as a coordinate pair, like `"400x600"` or `"1.0,0.5"`.
```

```
/// Specifically, `s` should have the form <left><sep><right>, where <sep> is
/// the character given by the `separator` argument, and <left> and <right> are
/// both strings that can be parsed by `T::from_str`. `separator` must be an
/// ASCII character.
///
/// If `s` has the proper form, return `Some<(x, y)>`. If it doesn't parse
/// correctly, return `None`.
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",        ','), None);
    assert_eq!(parse_pair::<i32>("10,",     ','), None);
    assert_eq!(parse_pair::<i32>(",10",     ','), None);
    assert_eq!(parse_pair::<i32>("10,20",   ','), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",    'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5, 1.5)));
}
```

The definition of `parse_pair` is a *generic function*:

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
```

You can read the clause `<T: FromStr>` aloud as, “For any type `T` that implements the `FromStr` trait…” This effectively lets us define an entire family of functions at once: `parse_pair::<i32>` is a function that parses pairs of `i32` values, `parse_pair::<f64>` parses pairs of floating-point values, and so on. This is very much like a function template in C++. A Rust programmer would call `T` a *type parameter* of `parse_pair`. When you use a generic function, Rust will often be able to infer type parameters for you, and you won’t need to write them out as we did in the test code.

Our return type is `Option<(T, T)>`: either `None` or a value `Some((v1, v2))`, where `(v1, v2)` is a tuple of two values, both of type `T`. The `parse_pair` function doesn’t use an explicit return statement, so its return value is the value of the last (and the only) expression in its body:

```
match s.find(separator) {
    None => None,
    Some(index) => {
        ...
```

```
    }
}
```

The `String` type's `find` method searches the string for a character that matches `separator`. If `find` returns `None`, meaning that the separator character doesn't occur in the string, the entire `match` expression evaluates to `None`, indicating that the parse failed. Otherwise, we take `index` to be the separator's position in the string.

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None
}
```

This begins to show off the power of the `match` expression. The argument to the match is this tuple expression:

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

The expressions `&s[..index]` and `&s[index + 1..]` are slices of the string, preceding and following the separator. The type parameter `T`'s associated `from_str` function takes each of these and tries to parse them as a value of type `T`, producing a tuple of results. This is what we match against:

```
(Ok(l), Ok(r)) => Some((l, r)),
```

This pattern matches only if both elements of the tuple are `Ok` variants of the `Result` type, indicating that both parses succeeded. If so, `Some((l, r))` is the value of the match expression and hence the return value of the function.

```
_ => None
```

The wildcard pattern `_` matches anything and ignores its value. If we reach this point, then `parse_pair` has failed, so we evaluate to `None`, again providing the return value of the function.

Now that we have `parse_pair`, it's easy to write a function to parse a pair of floating-point coordinates and return them as a `Complex<f64>` value:

```
/// Parse a pair of floating-point numbers separated by a comma as a complex
/// number.
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
```

```
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im: -0.0625 }));
    assert_eq!(parse_complex(",-0.0625"), None);
}
```

The `parse_complex` function calls `parse_pair`, builds a `Complex` value if the coordinates were parsed successfully, and passes failures along to its caller.

If you were reading closely, you may have noticed that we used a shorthand notation to build the `Complex` value. It's common to initialize a struct's fields with variables of the same name, so rather than forcing you to write `Complex { re: re, im: im }`, Rust lets you simply write `Complex { re, im }`. This is modeled on similar notations in JavaScript and Haskell.

Mapping from Pixels to Complex Numbers

The program needs to work in two related coordinate spaces: each pixel in the output image corresponds to a point on the complex plane. The relationship between these two spaces depends on which portion of the Mandelbrot set we're going to plot, and the resolution of the image requested, as determined by command-line arguments. The following function converts from *image space* to *complex number space*:

```
/// Given the row and column of a pixel in the output image, return the
/// corresponding point on the complex plane.
///
/// `bounds` is a pair giving the width and height of the image in pixels.
/// `pixel` is a (column, row) pair indicating a particular pixel in that image.
/// The `upper_left` and `lower_right` parameters are points on the complex
/// plane designating the area our image covers.
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                           upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width  / bounds.0 as f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as f64
        // Why subtraction here? pixel.1 increases as we go down,
        // but the imaginary component increases as we go up.
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 200), (25, 175),
                              Complex { re: -1.0, im:  1.0 },
                              Complex { re:  1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.75 });
}
```

Figure 2-4 illustrates the calculation `pixel_to_point` performs.

The code of `pixel_to_point` is simply calculation, so we won't explain it in detail. However, there are a few things to point out. Expressions with this form refer to tuple elements:

```
pixel.0
```

This refers to the first element of the tuple `pixel`.

```
pixel.0 as f64
```

This is Rust's syntax for a type conversion: this converts `pixel.0` to an `f64` value. Unlike C and C++, Rust generally refuses to convert between numeric types implicitly; you must write out the conversions you need. This can be tedious, but being explicit about which conversions occur and when is surprisingly helpful. Implicit integer conversions seem innocent enough, but historically they have been a frequent source of bugs and security holes in real-world C and C++ code.

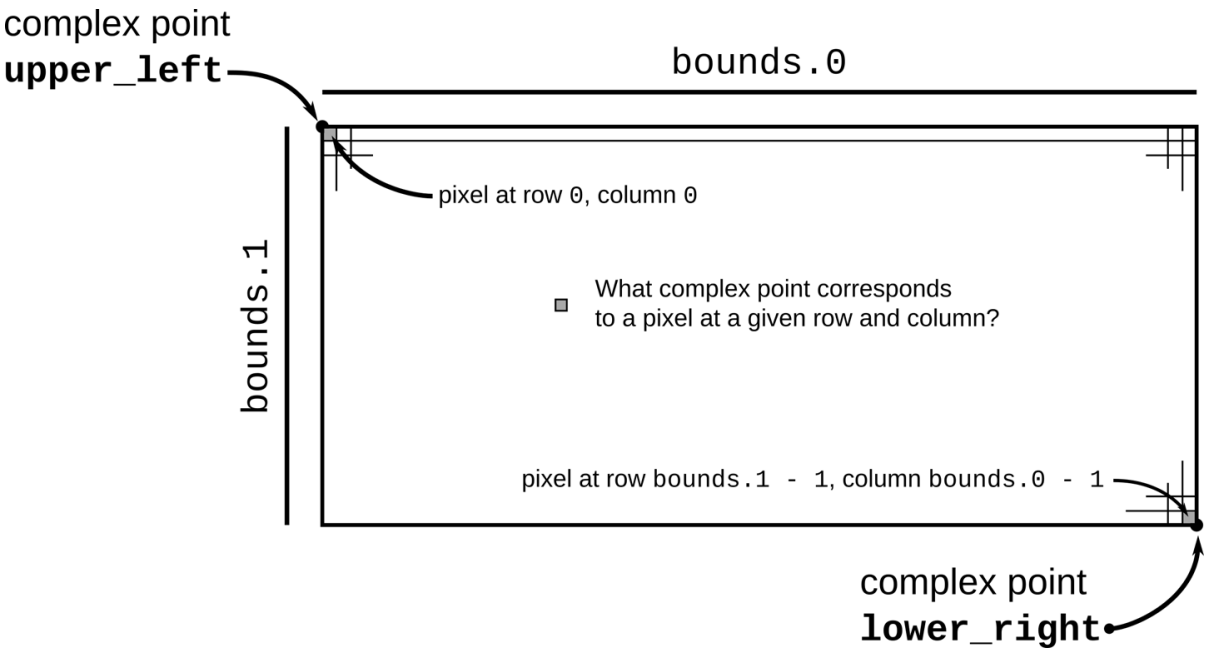

Figure 2-4. The relationship between the complex plane and the image's pixels

Plotting the Set

To plot the Mandelbrot set, for every pixel in the image, we simply apply `escape_time` to the corresponding point on the complex plane, and color the pixel depending on the result:

```
/// Render a rectangle of the Mandelbrot set into a buffer of pixels.
///
/// The `bounds` argument gives the width and height of the buffer `pixels`,
/// which holds one grayscale pixel per byte. The `upper_left` and `lower_right`
/// arguments specify points on the complex plane corresponding to the upper-
/// left and lower-right corners of the pixel buffer.
fn render(pixels: &mut [u8],
          bounds: (usize, usize),
          upper_left: Complex<f64>,
```

```
            lower_right: Complex<f64>)
{
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                       upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}
```

This should all look pretty familiar at this point.

```
pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

If `escape_time` says that `point` belongs to the set, `render` colors the corresponding pixel black (`0`). Otherwise, `render` assigns darker colors to the numbers that took longer to escape the circle.

Writing Image Files

The `image` crate provides functions for reading and writing a wide variety of image formats, along with some basic image manipulation functions. In particular, it includes an encoder for the PNG image file format, which this program uses to save the final results of the calculation. In order to use `image`, add the following line to the `[dependencies]` section of *Cargo.toml*:

```
image = "0.13.0"
```

With that in place, we can write:

```
use image::ColorType;
use image::png::PNGEncoder;
use std::fs::File;

/// Write the buffer `pixels`, whose dimensions are given by `bounds`, to the
/// file named `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{
    let output = File::create(filename)?;
```

```
    let encoder = PNGEncoder::new(output);
    encoder.encode(pixels,
                   bounds.0 as u32, bounds.1 as u32,
                   ColorType::Gray(8))?;

    Ok(())
}
```

The operation of this function is pretty straightforward: it opens a file and tries to write the image to it. We pass the encoder the actual pixel data from `pixels`, and its width and height from `bounds`, and then a final argument that says how to interpret the bytes in `pixels`: the value `ColorType::Gray(8)` indicates that each byte is an eight-bit grayscale value.

That's all straightforward. What's interesting about this function is how it copes when something goes wrong. If we encounter an error, we need to report that back to our caller. As we've mentioned before, fallible functions in Rust should return a `Result` value, which is either `Ok(s)` on success, where `s` is the successful value, or `Err(e)` on failure, where `e` is an error code. So what are `write_image`'s success and error types?

When all goes well, our `write_image` function has no useful value to return; it wrote everything interesting to the file. So its success type is the *unit* type `()`, so called because it has only one value, also written `()`. The unit type is akin to `void` in C and C++.

When an error occurs, it's because either `File::create` wasn't able to create the file or `encoder.encode` wasn't able to write the image to it; the I/O operation returned an error code. The return type of `File::create` is `Result<std::fs::File, std::io::Error>`, while that of `encoder.encode` is `Result<(), std::io::Error>`, so both share the same error type, `std::io::Error`. It makes sense for our `write_image` function to do the same. In either case, failure should result in an immediate return, passing along the `std::io::Error` value describing what went wrong.

So to properly handle `File::create`'s result, we need to `match` on its return value, like this:

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(e) => {
        return Err(e);
    }
};
```

On success, let `output` be the `File` carried in the `Ok` value. On failure, pass along the error to our own caller.

This kind of `match` statement is such a common pattern in Rust that the language provides the `?` operator as shorthand for the whole thing. So, rather than writing out this logic explicitly every time we attempt something that could fail, you can use the following equivalent and much more legible statement:

```
let output = File::create(filename)?;
```

If `File::create` fails, the `?` operator returns from `write_image`, passing along the error. Otherwise, `output` holds the successfully opened `File`.

NOTE

It's a common beginner's mistake to attempt to use `?` in the `main` function. However, since `main` itself doesn't return a value, this won't work; instead, you need to use a `match` statement, or one of the short-hand methods like `unwrap` and `expect`. There's also the option of simply changing `main` to return a `Result`, which we'll cover later.

A Concurrent Mandelbrot Program

All the pieces are in place, and we can show you the `main` function, where we can put concurrency to work for us. First, a nonconcurrent version for simplicity:

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() != 5 {
        eprintln!("Usage: {} FILE PIXELS UPPERLEFT LOWERRIGHT",
                  args[0]);
        eprintln!("Example: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",
                  args[0]);
        std::process::exit(1);
    }

    let bounds = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_complex(&args[3])
        .expect("error parsing upper left corner point");
    let lower_right = parse_complex(&args[4])
        .expect("error parsing lower right corner point");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
        .expect("error writing PNG file");
}
```

After collecting the command-line arguments into a vector of `String`s, we parse each one and then begin calculations.

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

A macro call `vec![v; n]` creates a vector `n` elements long whose elements are initialized to `v`, so the preceding code creates a vector of zeros whose length is `bounds.0 * bounds.1`, where `bounds` is the image resolution parsed from the command line. We'll use this vector as a rectangular array of one-byte grayscale pixel values, as shown in Figure 2-5.

The next line of interest is this:

```
render(&mut pixels, bounds, upper_left, lower_right);
```

This calls the `render` function to actually compute the image. The expression `&mut pixels` borrows a mutable reference to our pixel buffer, allowing `render` to fill it with computed grayscale values, even while `pixels` remains the vector's owner. The remaining arguments pass the image's dimensions and the rectangle of the complex plane we've chosen to plot.

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

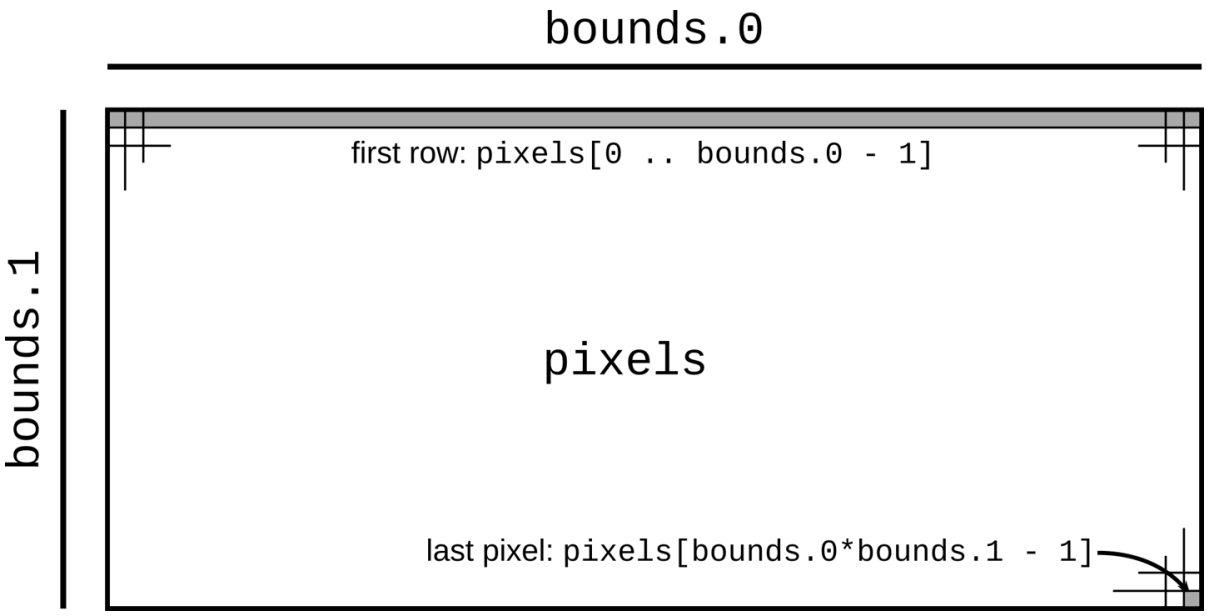

Figure 2-5. Using a vector as a rectangular array of pixels

Finally, we write the pixel buffer out to disk as a PNG file. In this case, we pass a shared (non-mutable) reference to the buffer, since `write_image` should have no need to modify the buffer's contents.

At this point, we can build and run the program in release mode, which enables many powerful compiler optimizations, and after several seconds, it will write a beautiful image to the file *mandel.png*:

```
$ cargo build --release
    Updating crates.io index
   Compiling autocfg v1.0.1
   ...
   Compiling image v0.13.0
   Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished release [optimized] target(s) in 25.36s
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
```

```
real    0m4.678s
user    0m4.661s
sys     0m0.008s
```

This command should create a file called *mandel.png*, which you can view with your system's image viewing program or in a web browser. If all has gone well, it should look like Figure 2-6.

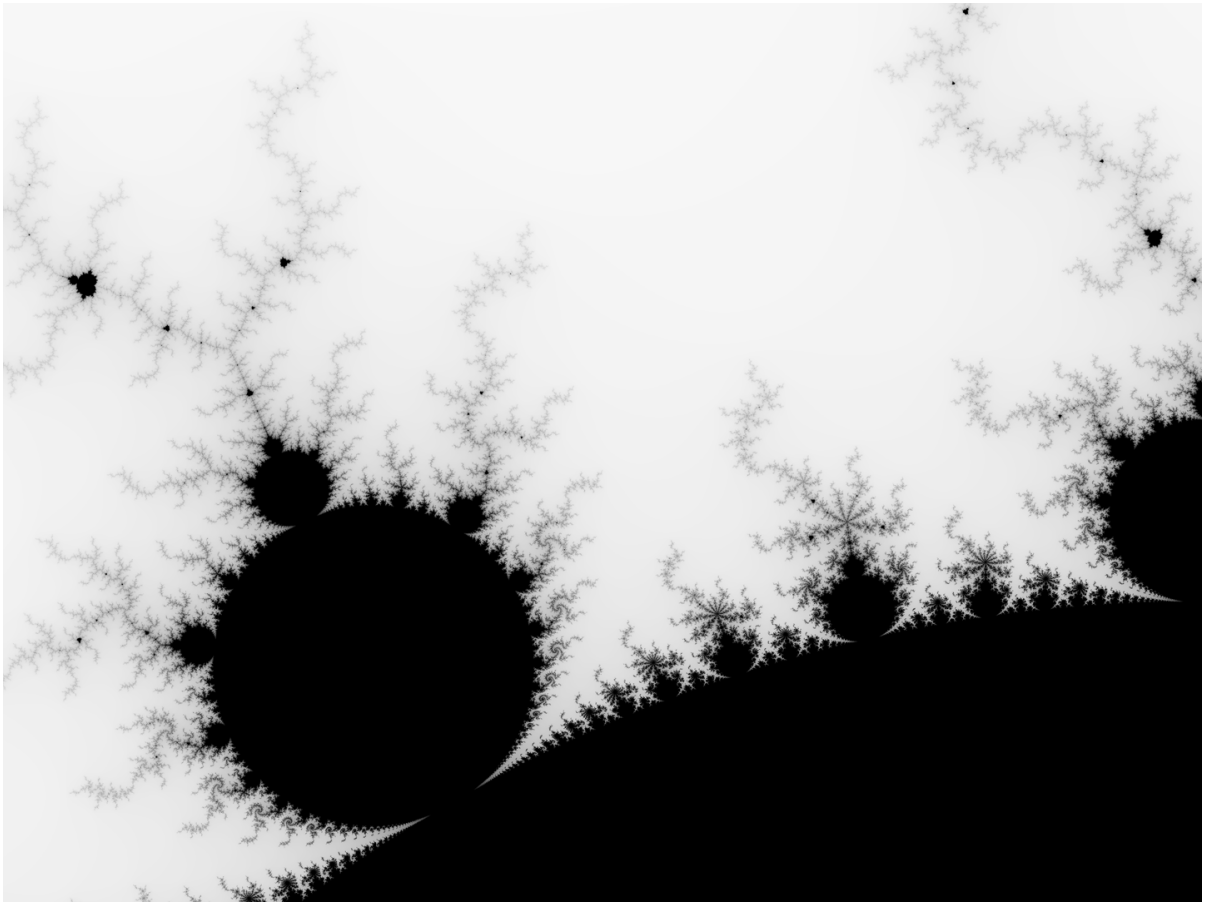

Figure 2-6. Results from parallel Mandelbrot program

In the previous transcript, we used the Unix `time` program to analyze the running time of the program: it took about five seconds total to run the Mandelbrot computation on each pixel of the image. But almost all modern machines have multiple processor cores, and this program used only one. If we could distribute the work across all the computing resources the machine has to offer, we should be able to complete the image much more quickly.

To this end, we'll divide the image into sections, one per processor, and let each processor color the pixels assigned to it. For simplicity, we'll break it into horizontal bands, as shown in Figure 2-7. When all processors have finished, we can write out the pixels to disk.

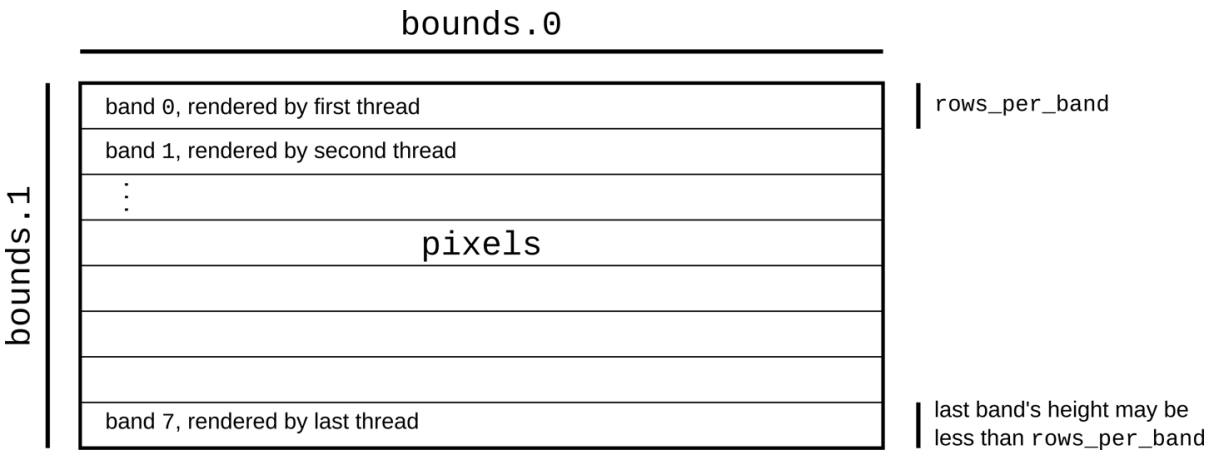

Figure 2-7. Dividing the pixel buffer into bands for parallel rendering

The `crossbeam` crate provides a number of valuable concurrency facilities, including a *scoped thread* facility that does exactly what we need here. To use it, we must add the following line to our *Cargo.toml* file:

```
crossbeam = "0.8"
```

Then we need to take out the single line calling `render` and replace it with the following:

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left, lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height),
                               upper_left, lower_right);

            spawner.spawn(move |_| {
                render(band, band_bounds, band_upper_left, band_lower_right);
            });
        }
    }).unwrap();
}
```

Breaking this down in the usual way:

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;
```

Here we decide to use eight threads.[1] Then we compute how many rows of pixels each band should have. We round the row count upward to make sure the bands cover the entire image even if the height isn't a multiple of `threads`.

```
let bands: Vec<&mut [u8]> =
    pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

Here we divide the pixel buffer into bands. The buffer's `chunks_mut` method returns an iterator producing mutable, nonoverlapping slices of the buffer, each of which encloses `rows_per_band * bounds.0` pixels—in other words, `rows_per_band` complete rows of pixels. The last slice that `chunks_mut` produces may contain fewer rows, but each row will contain the same number of pixels. Finally, the iterator's `collect` method builds a vector holding these mutable, nonoverlapping slices.

Now we can put the `crossbeam` library to work:

```
crossbeam::scope(|spawner| {
    ...
}).unwrap();
```

The argument `|spawner| { ... }` is a Rust closure that expects a single argument, `spawner`. Note that, unlike functions declared with `fn`, we don't need to declare the types of a closure's arguments; Rust will infer them, along with its return type. In this case, `crossbeam::scope` calls the closure, passing as the `spawner` argument a value the closure can use to create new threads. The `crossbeam::scope` function waits for all such threads to finish execution before returning itself. This behavior allows Rust to be sure that such threads will not access their portions of `pixels` after it has gone out of scope, and allows us to be sure that when `crossbeam::scope` returns, the computation of the image is complete. If all goes well, `crossbeam::scope` returns `Ok(())`, but if any of the threads we spawned panicked, it returns an `Err`. We call `unwrap` on that `Result` so that, in that case, we'll panic too, and the user will get a report.

```
for (i, band) in bands.into_iter().enumerate() {
```

Here we iterate over the pixel buffer's bands. The `into_iter()` iterator gives each iteration of the loop body exclusive ownership of one band, ensuring that only one thread can write to it at a time. We explain how this works in detail in Chapter 5. Then, the `enumerate` adapter produces tuples pairing each vector element with its index.

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left =
    pixel_to_point(bounds, (0, top), upper_left, lower_right);
let band_lower_right =
    pixel_to_point(bounds, (bounds.0, top + height),
                   upper_left, lower_right);
```

Given the index and the actual size of the band (recall that the last one might be shorter than the others), we can produce a bounding box of the sort `render` requires, but one that refers only to this band of the buffer, not the entire image. Similarly, we repurpose the renderer's `pixel_to_point` function to find where the band's upper-left and lower-right corners fall on the complex plane.

```
spawner.spawn(move |_| {
    render(band, band_bounds, band_upper_left, band_lower_right);
});
```

Finally, we create a thread, running the closure `move |_| { ... }`. The `move` keyword at the front indicates that this closure takes ownership of the variables it uses; in particular, only the closure may use the mutable slice `band`. The argument list `|_|` means that the closure takes one argument, which it doesn't use (another spawner for making nested threads).

As we mentioned earlier, the `crossbeam::scope` call ensures that all threads have completed before it returns, meaning that it is safe to save the image to a file, which is our next action.

Running the Mandelbrot Plotter

We've used several external crates in this program: `num` for complex number arithmetic, `image` for writing PNG files, and `crossbeam` for the scoped thread creation primitives. Here's the final *Cargo.toml* file including all those dependencies:

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"

[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

With that in place, we can build and run the program:

```
$ cargo build --release
    Updating crates.io index
   Compiling crossbeam-queue v0.3.2
   Compiling crossbeam v0.8.1
   Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished release [optimized] target(s) in #.## secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m1.436s
user    0m4.922s
sys     0m0.011s
```

Here, we've used `time` again to see how long the program took to run; note that even though we still spent almost five seconds of processor time, the elapsed real time was only about 1.5 seconds. You can verify that a portion of that time is spent writing the image file by commenting out the code that does so and measuring again. On the laptop where this code was tested, the concurrent version reduces the Mandelbrot calculation time proper by a factor of almost four. We'll show how to substantially improve on this in Chapter 19.

As before, this program will have created a file called *mandel.png*. With this faster version, you can more easily explore the Mandelbrot set by changing the command-line arguments to your liking.

Safety Is Invisible

In the end, the parallel program we ended up with is not substantially different from what we might write in any other language: we apportion pieces of the pixel buffer out among the processors, let each one work on its piece separately, and when they've all finished, present the result. So what is so special about Rust's concurrency support?

What we haven't shown here is all the Rust programs we *cannot* write. The code we looked at in this chapter partitions the buffer among the threads correctly, but there are many small variations on that code that do not (and thus introduce data races); not one of those variations will pass the Rust compiler's static checks. A C or C++ compiler will cheerfully help you explore the vast space of programs with subtle data races; Rust tells you, up front, when something could go wrong.

In Chapters 4 and 5, we'll describe Rust's rules for memory safety. Chapter 19 explains how these rules also ensure proper concurrency hygiene.

Filesystems and Command-Line Tools

Rust has found a significant niche in the world of command-line tools. As a modern, safe, and fast systems programming language, it gives programmers a toolbox they can use to assemble slick command-line interfaces that replicate or extend the functionality of existing tools. For instance, the `bat` command provides a syntax-highlighting-aware `cat` alternative with built-in support for paging tools, and `hyperfine` can automatically benchmark anything that can be run with a command or pipeline.

While something that complex is out of scope for this book, Rust makes it easy to dip your toes into the world of ergonomic command-line applications. In this section, we'll show you how to build your own search-and-replace tool, complete with colorful output and friendly error messages.

To start, we'll create a new Rust project:

```
$ cargo new quickreplace
     Created binary (application) `quickreplace` package
$ cd quickreplace
```