Case Study

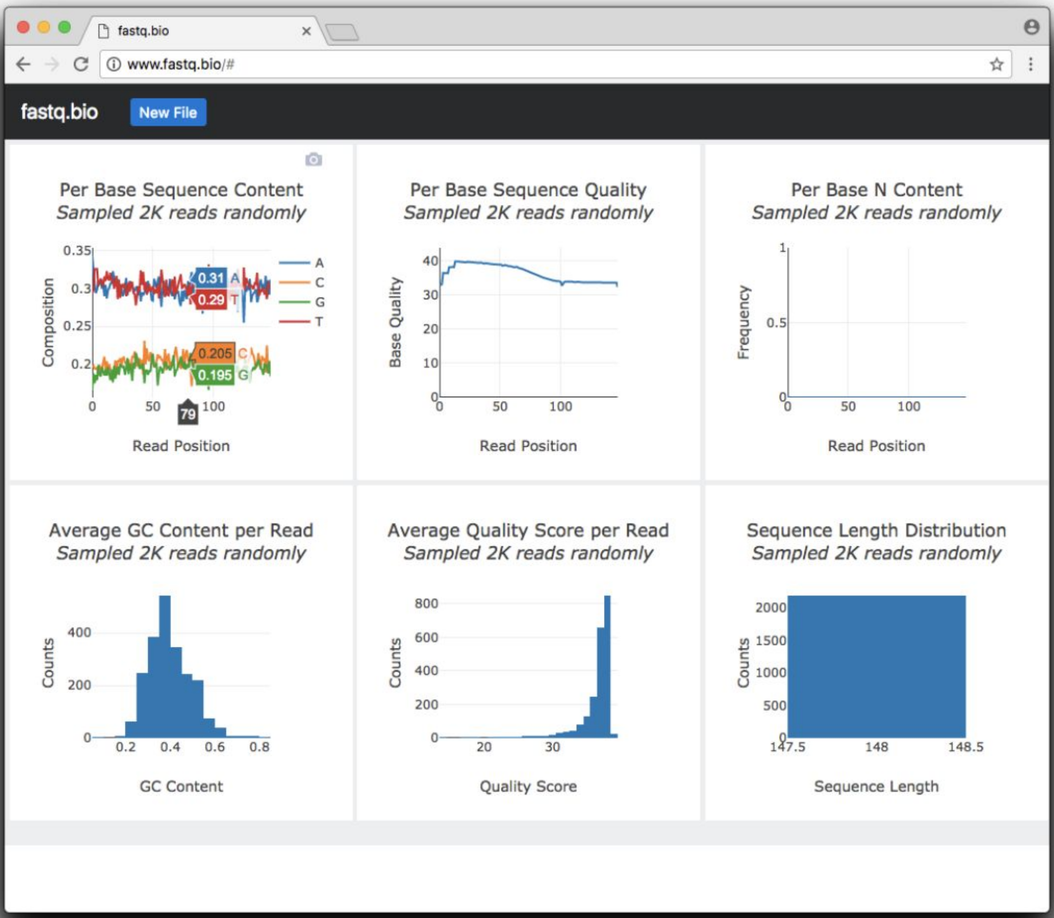
How we got a 20X speedup by leveraging WebAssembly

In this case study, let's consider a real-world scenario where WebAssembly helped improve a web application by speeding up the compute-heavy portion of the app.

Background

The app in question is fastq.bio, which is a web tool built for scientists who want to get a quick preview of the quality of their DNA sequencing data (sequencing is the process by which we read the letters in the DNA).

Here's a screenshot:


fastq.bio
www.fastq.bio/#
fastq.bio
New File
Per Base Sequence Content
Sampled 2K reads randomly
Composition
Read Position
Per Base Sequence Quality
Sampled 2K reads randomly
Base Quality
Read Position
Per Base N Content
Sampled 2K reads randomly
Frequency
Read Position
Average GC Content per Read
Sampled 2K reads randomly
Counts
GC Content
Average Quality Score per Read
Sampled 2K reads randomly
Counts
Quality Score
Sequence Length Distribution
Sampled 2K reads randomly
Counts
Sequence Length

The plots above give scientists information about how well the sequencing was done, and can be used to identify data quality issues at a glance.

Although there are dozens of command line tools available to generate such QC reports, the goal of fastq.bio is to give a preview of the quality of the data without having to go on the terminal. This is especially useful for scientists who are not comfortable with the command line.

The input file to the app is a plain text format called FASTQ, where each DNA sequence is encoded with four lines (warning: a facepalm is imminent):

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

(example from Wikipedia)

Briefly:

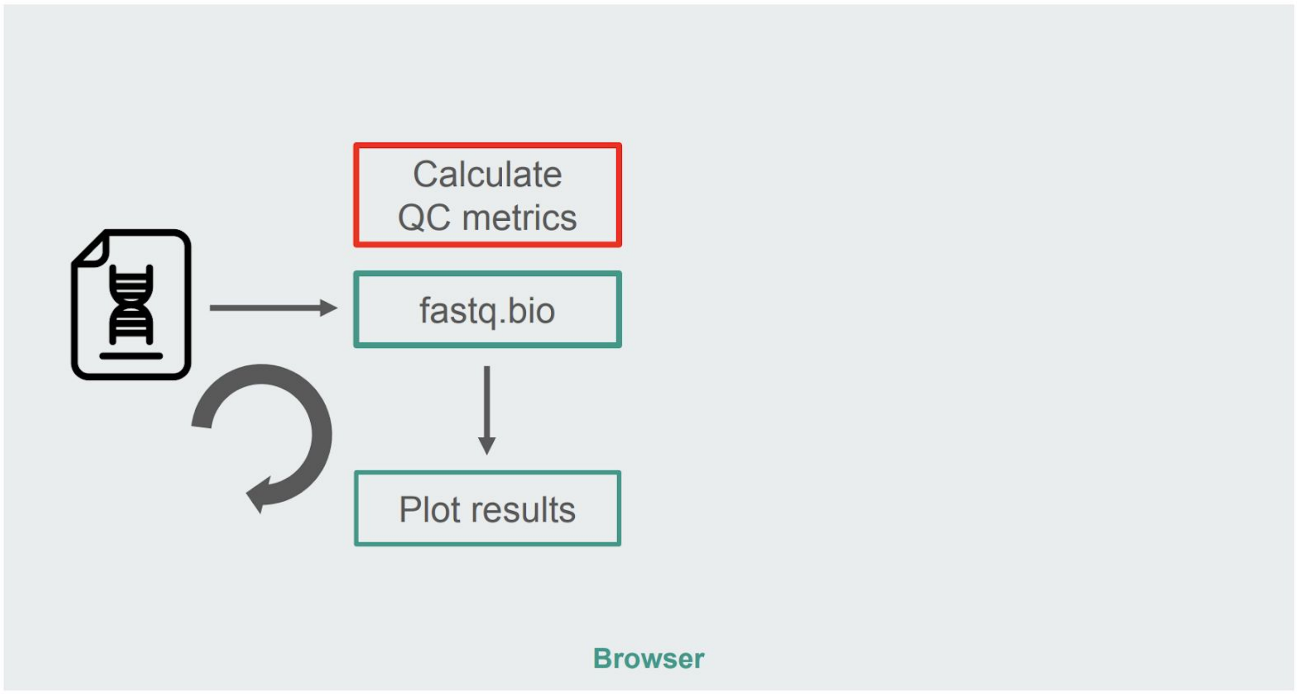
- The first line starts with an @ sign, followed by the name of the sequence.
- The second line contains the DNA sequence
- The third line is a + sign; yep, you can ignore this line ¯_(ツ)_/¯
- The fourth line denotes the quality of each letter in the 2nd line, where the quality score is encoded as an ASCII number to reduce the file size

fastq.bio v1

In the first version of fastq.bio, once the user chooses a file from their computer, we use the FileReader API (see Guide to FileReader/WebWorkers) to sample randomly through the file, extract a chunk of the file, and perform string manipulations to do things such as calculating how often we see an A at position *i* of a DNA sequence, what is the distribution of the quality score for DNA sequences in the file, and what is the distribution of the length of the DNA sequences, to name a few.

Once the metrics are calculated for that file chunk, we plot the results interactively with the Plotly.js library, and move on to the next chunk. The reason for processing the file in smaller chunks is simply to improve the user experience: processing the whole file at once would take too long, because FASTQ files are generally in the hundreds of gigabytes. We found that a chunk size between 0.5 MB and 1 MB would make the application more seamless and would return information to the user more quickly.

The architecture of v1 was fairly straightforward:



The box in red is where we did the string manipulations, which were the more intensive part of the application, which naturally makes it a good candidate for optimizing it with WebAssembly.

fastq.bio v2

Next, we wanted to explore whether WebAssembly could speed up the calculation of our QC metrics. To that end, we looked for an off-the-shelf tool for calculating QC metrics, given a FASTQ file. The requirements were:

- A tool that has been validated, and is trusted by the scientific community
- Written in C/C++ (or Rust!), so we can compile it to WebAssembly
- Not too many dependencies, so that compilation isn't too painful

After some research, we decided to go with `seqtk`, a commonly-used C tool for generating QC (and more generally for manipulating sequencing data files). What's more, the code looked fairly simple, with only one `.c` file and two `.h` files.

Armed with our tool and Emscripten, we compiled seqtk to WebAssembly (see Chapter 7 for details on compiling existing tools):

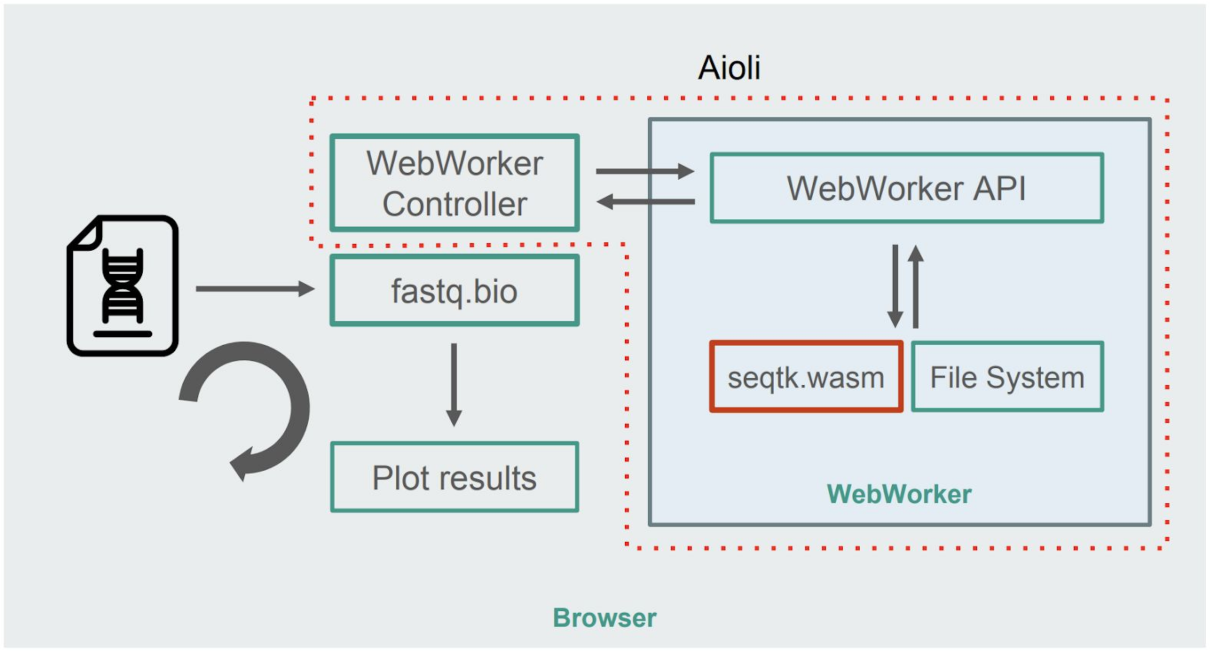
```
$ emcc seqtk.c \
    -o seqtk.js \         # Compile to WebAssembly (produces .js and .wasm)
    -O2 \                 # Ask compiler to optimize the code (docs)
    -lm \                 # Include the math library
    -s USE_ZLIB=1         # Include the zlib library (see Chapter 6)
```

To show how similar this is to the `gcc` command we would have used if we were compiling seqtk to binary instead of WebAssembly, here they are side-by-side, with differences in red:

```
$ emcc seqtk.c \          $ gcc seqtk.c \
    -o seqtk.js \               -o seqtk \
    -O2 \                       -O2 \
    -lm \                       -lm \
    -s USE_ZLIB=1               -lz
```

Since FASTQ files are a plain text format, they can be gzipped to reduce their footprint significantly, so we enable the `USE_ZLIB` flag to ensure that .gz files are properly supported in the browser.

With that in hand, here's what the fastq.bio v2 architecture looks like:



In v2, instead of running the calculations in the browser's main thread, we make use of WebWorkers, which allows us to run the calculations in a background thread to avoid negatively affecting the responsiveness of the browser. For an in-depth introduction to WebWorkers, see the attached Guide to WebWorkers + FileReader, as well as Chapter 10, which explores how to use WebWorkers with WebAssembly modules.

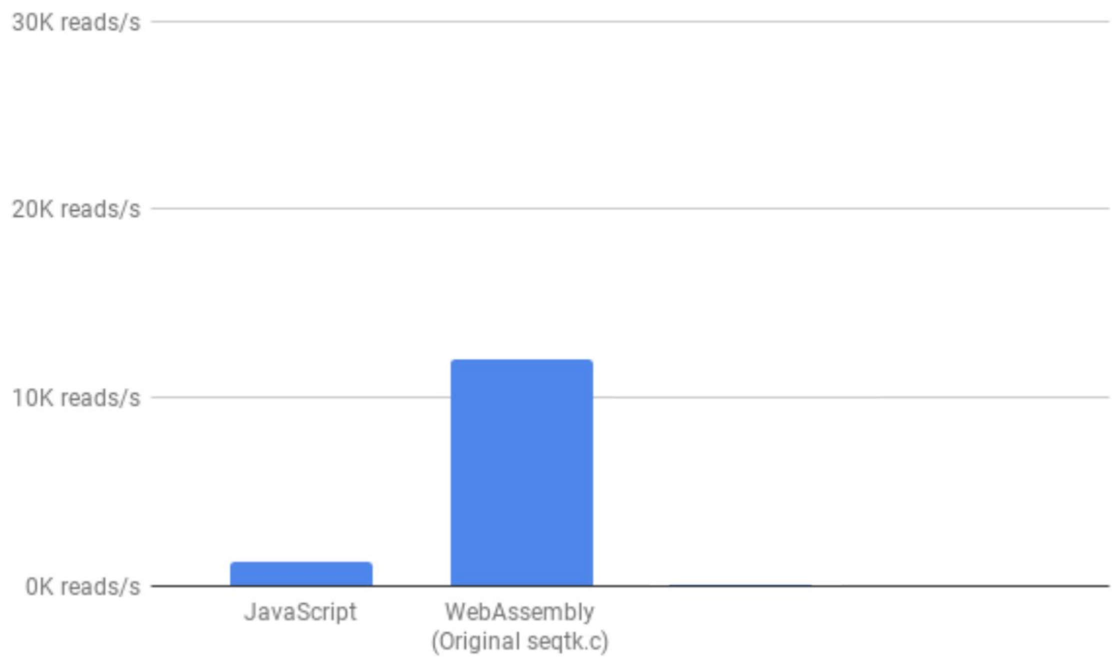
Specifically, the WebWorker controller is in charge of launching the Worker and managing the communication between the main thread and the worker. On the Worker's side, there is an API that accepts messages and acts on them. For example, you can send a File object to the Worker, and it will mount it onto the virtual file system (see Chapter 6 for details about how this works). We can then ask the Worker to run a seqtk command on the file we just mounted. When seqtk finishes running, the Worker sends the result back to the main thread via a message, which the main thread is waiting for. Once it receives the message, the main thread uses the resulting output to update the charts. Similar to v1, we also process the files in chunks and update the visualizations each time.

To automate commonly needed functionality that can be useful for future genomics web tools, we wrapped all the WebWorker + WebAssembly logic into a homegrown framework that we call Aioli.

Performance Optimization

Next, we evaluated how much faster our new implementation was. To that end, we calculated how many reads per second could be processed by each (ignoring the time it took to update the visualizations, which is constant across both implementations).

Out of the box, we already see a ~9X speedup:



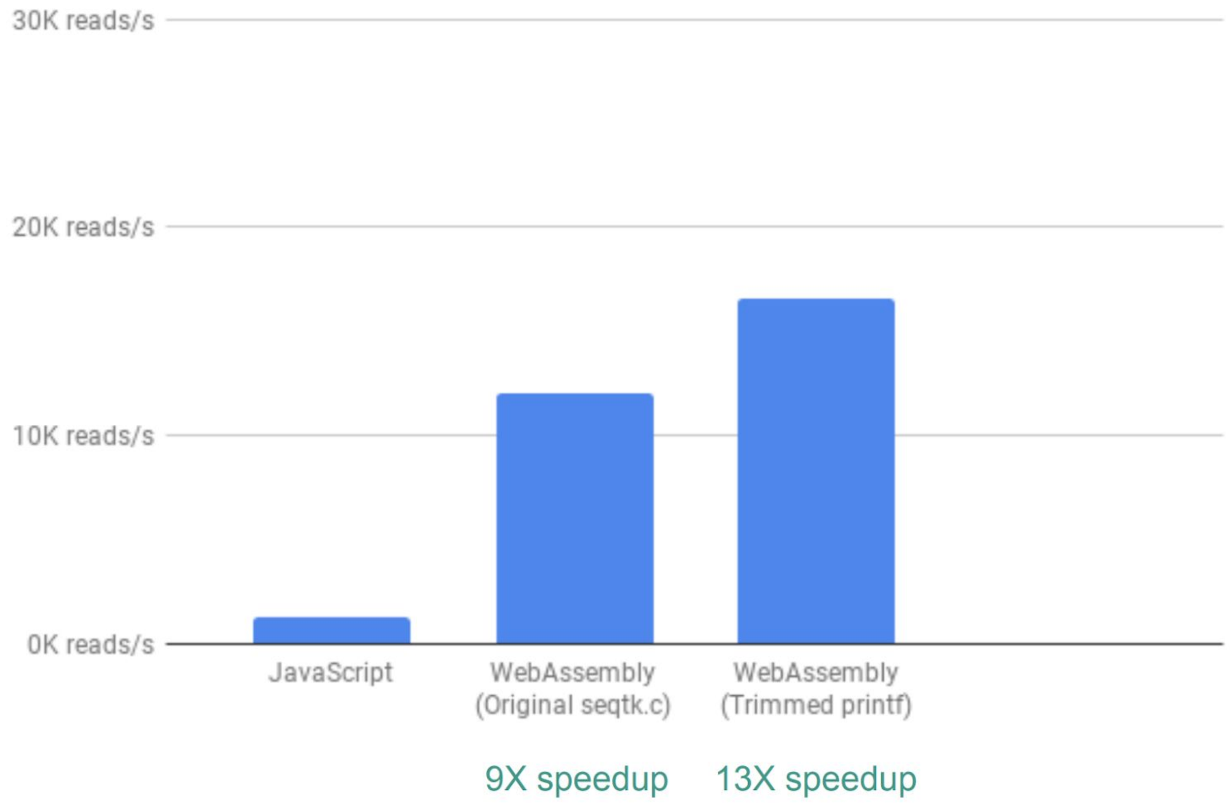
9X speedup

This is already very good, given that it was a relatively easy speedup to achieve (that is once you understand WebAssembly!).

Next, we noticed that although seqtk outputs a lot of generally useful QC metrics, many of these metrics are not actually used or plotted by our web app. By removing some of the output for the metrics we didn't need, we were able to see an even greater speedup of 13X:

Note

Removing unnecessary `printf()` statements can make a significant difference if the function that does the output is called very often by your application (as was the case in fastq.bio, since it was called on each file chunk).



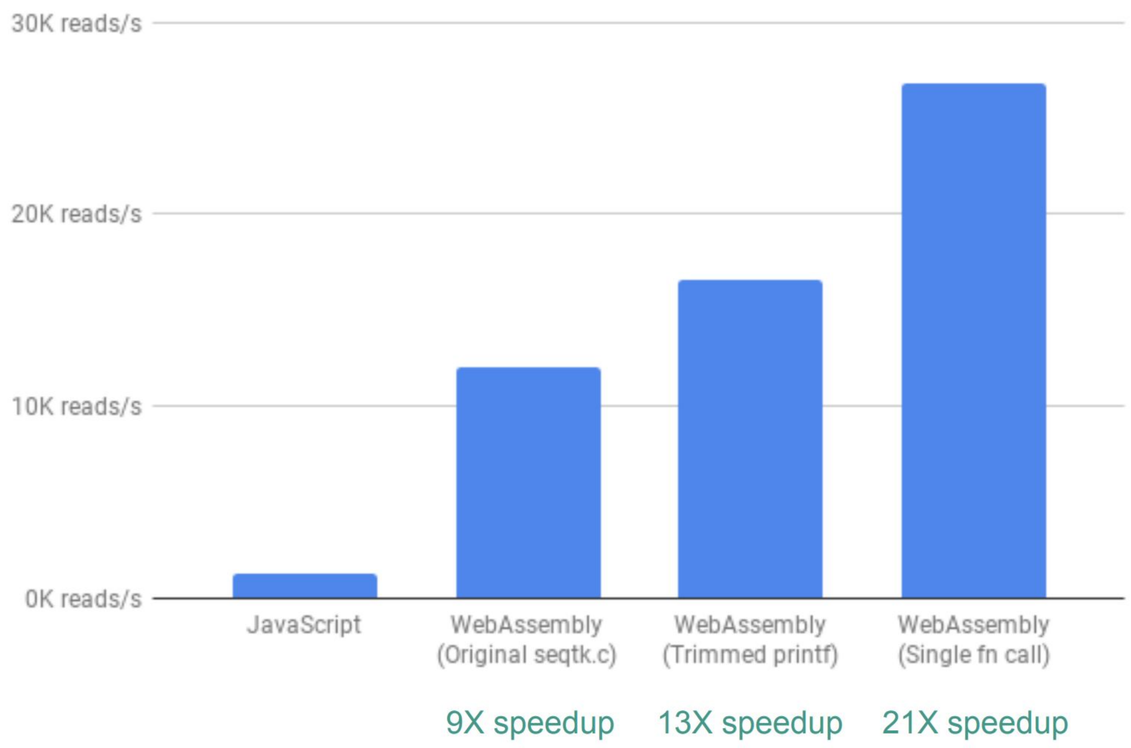
This again is a great improvement given how easy it was to achieve, by literally commenting out print statements that were not needed.

Finally, there is one more improvement we looked into. At this point, the way fastq.bio obtains the QC metrics of interest was by calling two different C functions, each of which calculate different metrics of interest. Specifically, one function returns information in the form of a histogram (a list of values that we can bin), whereas another function returns information as a function of DNA sequence position. Unfortunately, this means that the same chunk of file is read twice, which suggested further optimizations.

Note

Another way to improve performance is to analyze how your program parses files, and to refactor that code to only do it once. In our case, we noticed that we were reading the same file chunk twice, which adds unnecessary overhead. On the other hand, the downside is that if you're compiling a 3rd party software, you've now diverged from their code, and would need to apply this conversion for future releases of that 3rd party tool.

To that end, we merged the code for the two functions into one (albeit messy) function that returns the two outputs in one go. The two outputs have different numbers of columns in their outputs so we did some more wrangling when receiving the output on the JavaScript side, but it was worth it: Doing so allowed us to achieve a >20X speedup!



Warning

Now is a good time for a caveat. Don't expect to always get 20X speedup when using WebAssembly. You might only get a 2X speedup, or a 20% speedup. **Or even a slow down** if you need a lot of communication back and forth between your WebAssembly and JavaScript logic.

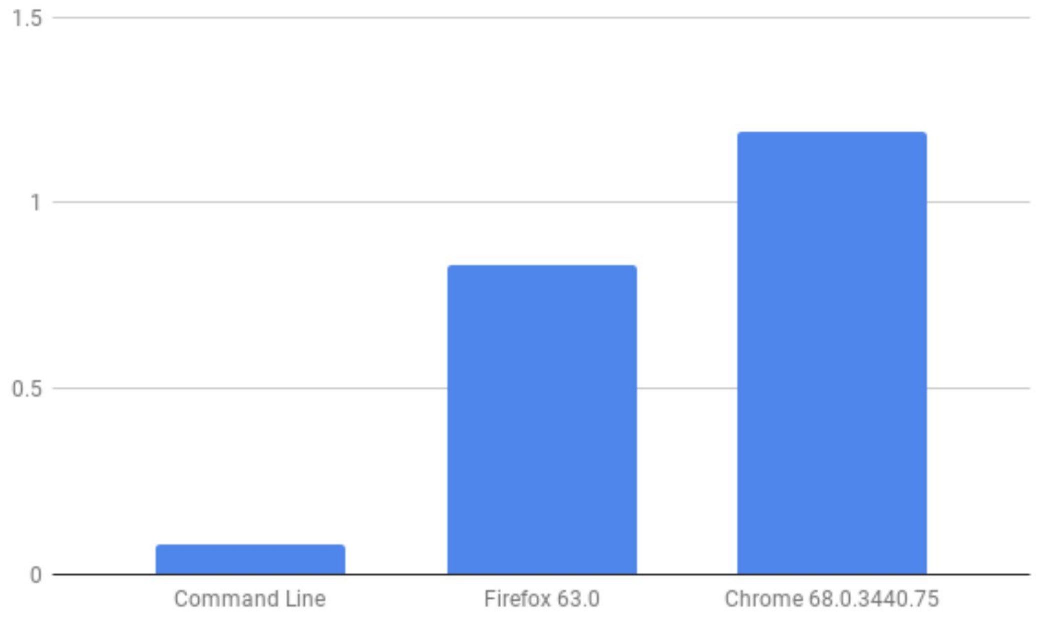
One final note: the performance comparisons above are done by comparing our WebAssembly implementation to our original custom JavaScript implementation. We made some attempts to optimize the original JavaScript code to make it faster, but we didn't place too much effort on that.

And that's the point of WebAssembly! In this use case, we already have a great off-the-shelf tool that does what we need it too, so there's no need to write our own or try to optimize it.

Near-native performance?

One of the selling points of WebAssembly is that you can get the runtime of your compiled WebAssembly code to run at near-native speeds, i.e. however long it would take on the command line.

We tested this for our application and found that, **even with the 20X speedup, we're still ~10X slower than running seqtk directly on the command line**:



Command line ~10X faster

There are a few reasons that could explain this discrepancy: **(1)** this application (as does much of genomics) is IO-bound, which could be slowing it down; **(2)** we're breaking down the file in small chunks, which has added overhead of message passing between the worker and the main thread.

Note

A good resource for learning more about optimizing your code and compilation approach for WebAssembly is Emscripten's Optimization Tips.