capstone

Daniel Hackney

December 9, 2013

Contents

1 Writeup

1.1 Abstract

With Moore's Law no longer providing automatic increases in sequential execution speed, software must be written to take advantage of the increase in CPU cores in order to enjoy increased performance. New libraries and languages have arisen to make this task easier and perform better. The Rust programming language is designed to make concurrent programming safe and fast. The experimental web browser engine Servo is written in Rust and aims to bring memory safety and concurrency to the browser engine. One common task performed by web browsers is the tokenization of textual files such as CSS. A Rust implementation of the speculative parallel computation algorithm specified in the Vaswani paper was created and benchmarked for the CSS files of the Alexa top 11 sites.

1.2 Interface

Two functions are provided, as in the Vaswani paper: `spec` and `specfold`.

1.2.1 `spec`

This function takes three functions, a `producer`, a `predictor` and a `consumer`. The `producer` is launched in a separate Rust task (which are multiplexed onto OS threads) as a `Future`. Meanwhile, the `predictor` is called, and its result is used as the input to the `consumer`. When the `producer` completes, its return value is compared to that of the `predictor`; if the values match, then the result of `consumer(predictor())` is returned. If not, then the `consumer` is re-run with the output of the `producer`.

There are a few important assumptions being made:

1. The `consumer` can be re-run. If the prediction is unsuccessful, the `consumer` is re-executed with the result of the `producer`.
2. The `predictor` is much cheaper than the `producer`. All of the parallelism comes from the ability to get an accurate prediction quickly so the `consumer` can run in parallel with the `producer`. If the `predictor` is too slow, then there is no advantage to using a predictive scheme.

Note that assumption 1 is required for correctness while assumption 2 is for performance.

1.2.2 `specfold`

This is the iterative version of the `spec` function. It allows speculative execution of the same `loop_body` multiple times. It starts `iters` parallel tasks, and calls `loop_body(i, predictor(i))`. At each iteration, it stores the predicted value and the result. At the end, it validates each of the predictions, re-running any mispredictions. The program uses the "sequential" validation mode specified in the paper. There is the potential for a cascade of recomputations if early results are incorrect. In the worst case, all iterations other than the first will be run twice. As with the one-shot `spec` version, this depends on a fast and relatively accurate `predictor` function.

The reason the arguments are functions which return functions is due to the borrowing semantics of Rust. If the arguments were called directly, they would be shared between all of the spawned Rust tasks, which would not be safe. The closures might capture local variables which would then be unsafely shared among multiple Rust tasks. Instead, `loop_body` and `predictor` are only called in the main task, and their return value (a function) is passed into the task, ensuring it is called only once.

1.3 CSS tokenization

The CSS tokenizer used is a modification of the pure-Rust CSS parser produced by Mozilla for the Servo project. For this project, it was modified to separate the tokenization from the parsing steps. This project is concerned only with tokenization, so all of the parsing machinery was removed.

In the speculative mode, the input string is split into equal-sized chunks, each of which is tokenized independently in a concurrently-running task. For prediction, the tokenizer looks back from the chunk beginning a fixed number of characters (currently 10) and begins lexing until it consumes the first full token which crosses the chunk boundary. For the most part, CSS tokens are short, so this strategy generally works pretty well.

1.3.1 Mispredictions

The following is a table of the fraction of incorrect predictions for a given number of tasks:

Number of Tasks	Misprediction rate
5	0.1263
9	0.1303
13	0.1202
17	0.1217
21	0.1195
25	0.1211
29	0.1174
33	0.1184

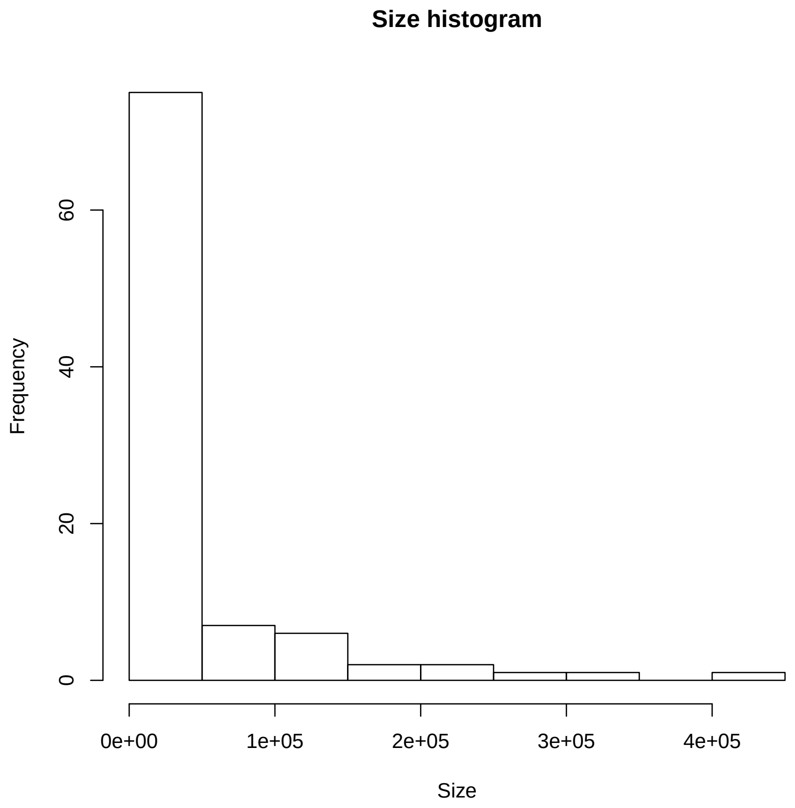
The first row, in which only a single task is used, is excluded because the first "prediction" is always correct. The mispredictions are clustered in specific files, with many files having no mispredictions.

1.4 Performance testing

Execution speed was tested on an Amazon EC2 virtual server using the c3.8xlarge instance type, which has 32 cores. A 64-bit Ubuntu 13.10 Linux image was launched with Rust 0.8. The CSS files from the following sites were extracted using the "Save Page As" functionality of Firefox:

- Amazon
- Baidu
- Facebook
- Google
- LinkedIn
- Outlook
- QQ
- Twitter
- Wikipedia
- Yahoo
- YouTube

The size of the files ranged from 23 to 428,335 bytes with an mean of 38,805 bytes and a standard deviation of 71,976 bytes. Below is a histogram of file sizes:

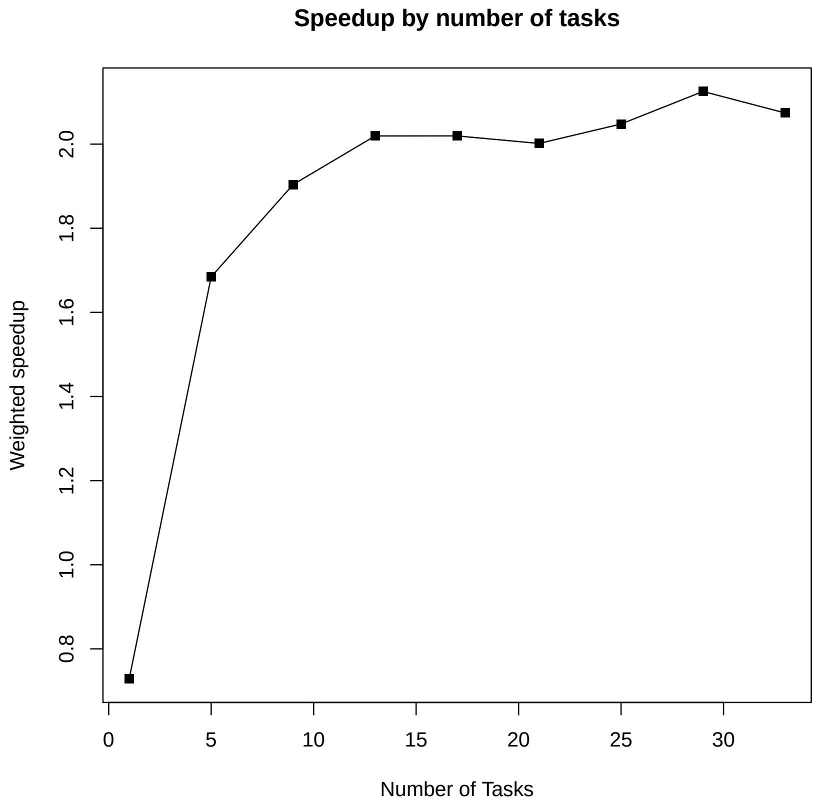


Performance is compared by running the CSS lexer on the entire input sequentially followed by running it in parallel with a specified number of parallel tasks and comparing the execution speed.

The following is a table of weighted speedups achieved for a given number of tasks:

Tasks	Weighted_Speedup
1	0.728471789655285
5	1.68442157020713
9	1.90393208201773
13	2.01942937241626
17	2.01952268941983
21	2.00156000021701
25	2.0478170857611
29	2.12520910227842
33	2.07411695745581

And a graph of the same:



1.4.1 Factors in performance

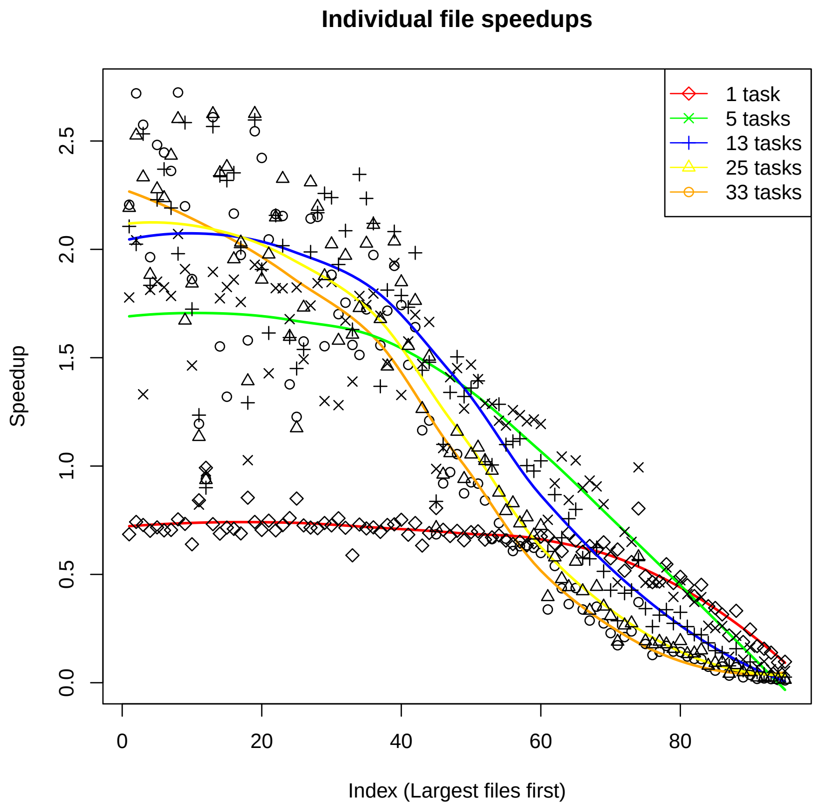
The interface of `specfold` was modified to return a statistics object which records the number of failed predictions. Although the most significant factor in the speedup achieved is the size of the file, the number of mispredictions is also very important, since entire sections of the file may need to be reprocessed if a prediction fails. The following table shows the correlation between the file size and speedup:

Number of Tasks	Size-speedup correlation
1	0.3367
5	0.4749
9	0.5039
13	0.5388
17	0.5729
21	0.5707
25	0.5973
29	0.636
33	0.6436

And the misprediction–speedup correlation:

Number of Tasks	Size-speedup correlation
1	nil
5	-0.3266
9	-0.26
13	-0.2515
17	-0.2311
21	-0.1963
25	-0.2066
29	-0.1758
33	-0.156

The following is a plot of all of the times of the individual files for 1, 5, 13, 25, and 33 tasks, ordered by file size:



1.4.2 Investigating mispredictions

Looking at some of the files with the most mispredictions, many of them have large blocks of base64-encoded data. This appears as PNG, GIF, or SVG images or WOFF (Web Open Font Format) files. This totally defeats the prediction algorithm, since the base64-encoded data appears within a `url()` token, and must therefore be regarded as a single, large lexical token. This base64-encoded string can span multiple chunks, defeating any ability of the parallel operation to gain speed increases. In one of the files from live.com, for example, the majority of the source is taken up by three large base64-encoded WOFF files, each of which is between 32 and 44 KB. The entire file produces only 34 tokens but is 119KB. This is a dramatic case, but 21 of the 95 tested files includes base64 data.

1.5 Possible optimizations

Given the observed performance characteristics, there are some strategies which could lead to increased performance.

- The overhead of creating tasks and synchronization is not worth it for very small files, so a purely sequential strategy could be used for small files.
- Since base64-encoded text destroys any possibility of parallel speedup, if it is detected, the program could fall back to a purely-sequential execution.
- Increase in predictor accuracy.
- Reduction of other bottlenecks. Given that, in the absence of prediction failures, there is no contention between tasks, performance should scale roughly linearly. This is especially true for large files in which the overhead is less significant. This would require a better understanding of the performance characteristics of Rust as well as a profiler to detect bottlenecks.

1.6 Conclusion

It has been shown that Rust is a useful language for safe concurrent programming and that, even in its early state, can achieve significant speedups from parallel execution on multicore machines.