CS 1632 – Deliverable 4:

Performance Testing

Jeremy Kato – github.com/jeremykato

Danielle Moonah - github.com/tivieta

github.com/jeremykato/D4

Summary

The most difficult part of this deliverable was mostly in execution of making the code run as fast as possible. It took a solid amount of experimentation and thinking through before our group found the ways to get a maximum speedup out of the program, as some techniques (multi-threading, OpenCL) brought unique problems to the code (race conditions, overhead, etc).

We considered and worked around all of the following cases:

- Wrong number of parameters provided
- File provided not being found
- Invalid number of separators (vertical bars) in a line
- Block numbers being wrong (ie, starting at 0 and increasing by 1 each line)
- Block numbers being invalid (unable to be parsed as an integer)
- Block hashes being invalid (unable to be parsed as a hexadecimal integer)
- Block hashes not matching the prior block's hash
- Having zero transactions in a block
- Transaction lists being in an invalid format.
- Transaction address in an incorrect format
- Invalid transaction amount (anything besides a positive integer)
- Having an address with a negative balance at the end of a block
- Invalid formatting on block time
- Block time being negative
- Block time being earlier than the previous block
- Block's listed hash not matching its actual hash value

In our initial program, the verifier program used the default exponentiation function (a ** b) in its hashes, which for the large values required in the hash function, was incredibly slow. Sure enough, the flamegraph revealed that the vast majority of the program's execution was spent performing the hash of each character over and over again.

To optimize our program, we took the following actions:

- Modular Exponentiation: Ruby has a built in modular exponent function (a.pow(b, mod)) that is much, much faster than performing the hash function and then taking the modulus.

- Caching Hash Values: To avoid calculating the same hashes over and over again, we added a hash map to the verifier's engine. Each time it went to calculate the hash value for a given UTF-8 character, it first checked if it was already in the hash map, and if so, it used that value instead. If not, it calculated the value and added it into the map.
- Parallel Processing: Finally, to squeeze extra performance out of the program, we looked for ways that tasks could be done in parallel. As a first note, we were going to use Ruby's threading implementation, but then we learned of the GIL and cried sadly. Fortunately, another library we found, Parallel, allowed us to access parallel threading in Ruby. We were originally going to make threads that each verified their own blocks in the blockchain, but this introduced some interesting problems.

  The main things that have to be verified include block numbers, timestamps, transaction validations, and the hashes on each block (both the current and previous block's hash). The problem is that transactions are extremely difficult to split into parallel, because each thread needs the results of the prior line before it can go on. It might be possible to gather the net difference of each block's transactions, and have a thread sweep through to verify them in the end, but this is rather costly for a number of reasons (ruins locality/ CPU cache performance, requires waiting for each block to be verified in order before it can proceed).

  Instead, we opted to make the tasks of verifying each block a separate thread. Our program first loads the entire blockchain into memory, and then splits into four threads.

- Thread 0 verifies the block's separators and block number
- Thread 1 verifies the block's timestamp's format and value
- Thread 2 verifies the block's transactions (format and leaving a positive value in each address)
- Thread 3 verifies the hash values on each block

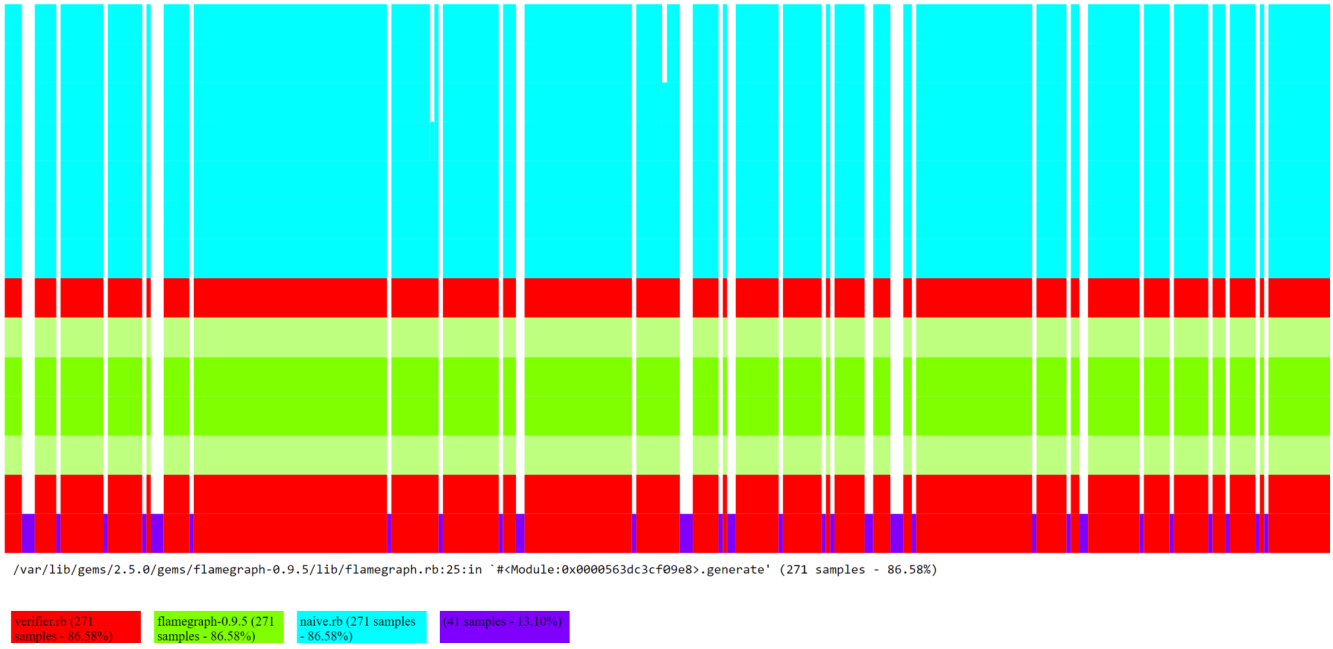
  By doing this, we now have task parallelism that doesn't create dependencies on one another – instead, the program's execution time is merely whichever task set takes the longest, and we avoid most of the overhead associated with creating new processes or tasks.

  With this complete, the program executed so fast that flamegraph was having trouble *collecting enough samples* to give us more data on what to optimize – a pleasant problem to have.

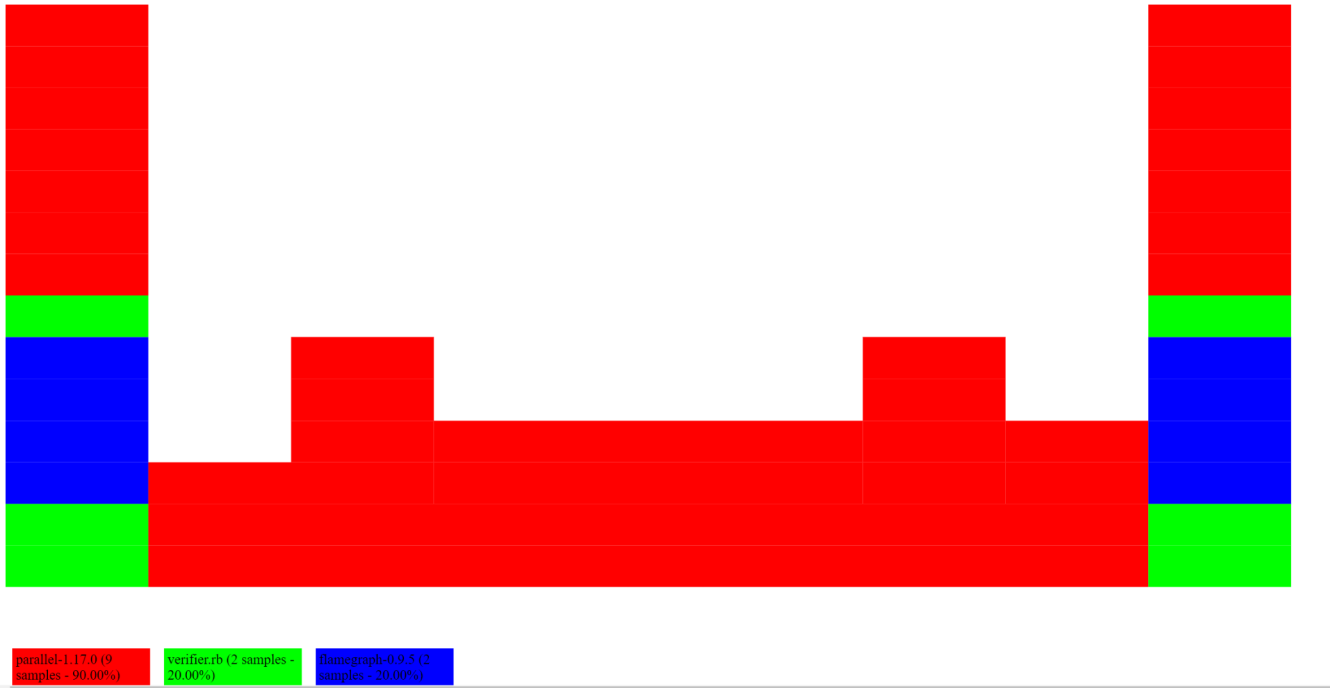
Data Summary

	Initial Implementation	**Optimized Implementation**
Test 1	30.79 sec	0.231 sec
Test 2	29.59 sec	0.221 sec
Test 3	31.16 sec	0.241 sec
Test 4	30.87 sec	0.235 sec
Test 5	30.23 sec	0.220 sec
Median	**30.87 sec**	**0.231 sec**
Average	**30.53 sec**	**0.230 sec**

Flamegraphs



(Original Implementation: sample.txt)



(Optimized Implementation: long.txt)