curl Parallel Testing Proposal

March 14, 2023

Daniel Fandrich

1. Summary

The curl test suite comprises over 1500 tests that run sequentially in order to test curl’s functionality. These tests take a long time to run relative to compile time, hampering developer productivity. Running them in parallel would utilize the multiple cores in modern processors to dramatically reduce their time to run. Other means to speed up tests may also be possible but with less dramatic improvements.

2. Background

The curl project has a large test suite which at the time of writing comprises 1585 test cases and cover most of its protocols and features. The test cases are run with a test harness that runs them sequentially from start to finish. There are various helper processes, such as protocol servers, started by the test harness to which curl sends requests, but there is no other parallelism involved in test runs.

The test harness runs at most one test server of each type, but sometimes different servers work together to fulfill the requirements of a particular test (such as stunnel + the http server to test https). Even if running several tests simultaneously were currently possible, no more than one test requiring a particular server could run at a time.

3. Baseline Timing

A baseline time to run tests was performed on a 6 core AMD Phenom processor running at 2.8 GHz, a fairly old but multicore processor, running an x86_64 Linux 5.15.88 kernel. curl was configured with:

```
./configure --with-ssl --with-libssh2 --with-nghttp2 --with-libidn2 --
enable-debug --enable-websockets
```

resulting in the following curl -V output:

```
curl 8.0.0-DEV (x86_64-pc-linux-gnu) libcurl/8.0.0-DEV OpenSSL/1.1.1q
zlib/1.2.12
brotli/1.0.9 zstd/1.4.8 libidn2/2.3.0 libpsl/0.21.1 (+libidn2/2.3.0)
libssh2/1.10.1_DEV nghttp2/1.42.0
Release-Date: [unreleased]
Protocols: dict file ftp ftps gopher gophers http https imap imaps
ldap ldaps mqtt pop3 pop3s rtsp scp sftp smb smbs smtp smtps telnet
```

```
tftp ws wss
Features: alt-svc AsynchDNS brotli Debug HSTS HTTP2 HTTPS-proxy IDN
IPv6 Largefile libz NTLM NTLM_WB PSL SSL threadsafe TLS-SRP
TrackMemory UnixSockets zstd
```

The curl version tested was from git in March 2023, a few weeks before the 8.0.0 release.

The tests were first compiled, then run once to warm the caches, then run again with (without Valgrind) with:

```
./runtests.pl '-a -n -rf !server\ sha256\ key\ check'
```

(the keyword disables two SSH tests that are incompatible with the test machine's OpenSSH installation). The result was 1531 tests (of 1585) run in 10:54 wall clock minutes (654 seconds), but only 131 seconds of CPU time was used (in about a 16:10 ratio of user time to system time, based on other test runs). The time between starting the command seeing the message that the first test is being run is about 1.5 seconds, which is test suite startup time that won't easily be parallelized.

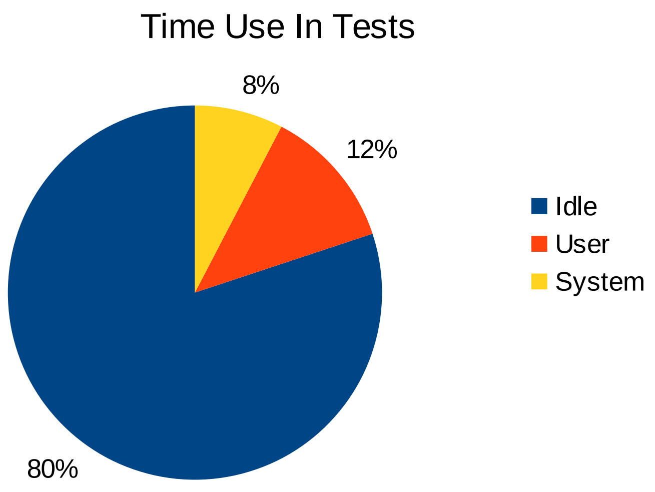

Figure 1: Time Use In Tests

The difference between the wall clock time and the CPU time is the time curl and the test suite spent waiting on test timeouts and I/O. A typical test run might spend:

188 seconds starting and verifying test harness servers.
17 seconds awaiting server logs lock removal.
5 seconds reading definitions and doing test preparations.
22 seconds verifying test results.

Much of the first two times is likely CPU idle time.

Running the test suite under systemd-run shows that the memory usage is about 13 MiB with one server started and reaches a peak of 128 MiB by the end, which includes all the servers that were started to service all tests.

The 1514 successfully executed tests had these statistics on their run times:

> Min: 0.014 sec.
> Max: 10.079 sec.
> Median: 0.10 sec.
> Average: 0.29 sec.
> Standard deviation: 0.84 sec.
> 95th percentile: 1.12 sec.

A brief and not statistically robust survey of about a dozen recent CI test runs on all 5 CI systems used by curl, on builds with differing build options that ran at least 1350 tests show they take between 327 and 1092 seconds to run with an average of 518 seconds). These differences may be due to the differing speeds of the servers used by these CI systems or the amount of over-subscription the CI systems apply to their free tier of CI services, or that slower tests were skipped on some builds.

An attempt to determine the amount of disk I/O instigated by the test suite using vmstat was stymied because the I/O rate was so low it was impossible to distinguish from background system I/O. This makes sense, since the entire tests/ subdirectory only contains 14 MiB of files, and there are only 17 tests that use `%repeat` to artificially increase the size of test data and never by much more than 1 MiB. The tests are therefore CPU bound and not I/O bound and I/O throughput of a system should make no discernible difference in run time.

Running the test suite but providing a keyword that does not exist in any test file causes all the files to be parsed but no tests to be run. This takes about 24 seconds, which is nearly 100% CPU usage. This works out to about 15 msec per test.

4. Potential Speedups

Parallel Tests

This has the potential for the greatest decrease in test execution time. Since only 20% of time is spent in CPU computations, running other tests during idle times could reduce execution time by 80% with only a single core available. If the CPU load could be spread evenly over multiple cores on a multicore system, the wall clock time would drop even further, by a factor of the number of cores. On a modern server CPU with 24 cores, this means the time to run the test suite could theoretically be brought down to 131/24 = 5.5 seconds. This is a lower bound that isn’t actually achievable, though. Some of the reasons are:

- It’s not possible to keep 24 CPUs at 100% utilization for the entire time, which is what it would take to achieve that feat. There will inevitably be time spent waiting for I/O.
- It won’t be possible to evenly distribute the load over all processors without over-subscription, which causes overhead.
- Some tests actually test timeouts that last longer than that. For example, test 303 uses an 8 second timeout, so that test will never run in less than that wall clock time.
- Coordinating all those tests to run in parallel will cause extra overhead that will add at least some CPU time.
- It takes CPU time to start and verify each new test server, and many additional test servers will be required to be started to run tests in parallel. Since 91% of tests require at least one server, nearly all parallel test threads will need to start at least one server, which could increase the total server startup overhead by two orders of magnitude.
- It currently takes significant CPU time (24 seconds) just to parse all the tests once in the test server. This parsing is in the critical path and will limit the fastest that the test can run. The parsing done by a parallel test coordinator will be more simple than is performed now, but if it stays on the critical path, parsing will need to be further optimized or it will need to be parallelized independently of test execution.

Still, if we assume only 50% effectiveness in distributing load to account for extra overhead, it should be achievable to bring the total time down to double the ideal numbers, or about 10 seconds (deliberately slow tests notwithstanding). For comparison, this is faster than the three slowest single tests take to run (tests 1112 and 1117) which take about 24 seconds each, due to deliberately-added delays. A modern dual core machine (such as some CI runners), has a much more efficient processor that’s probably 3 times faster per core than the one used for the benchmarks above. Under such conditions on an unloaded machine, a total test time of 131/2/3/0.50=44 seconds on CI builds is reasonably achievable.

By implementing parallel tests and oversubscribing test runners to CPU cores, other delays introduced in tests that are due to blocking waits and not CPU spinning, don’t really matter much any more. If too many tests have blocking waits, then we can simply introduce more test runners to keep the CPU cores full.

Parallel Server Startup

Many tests require more than one server be started before the test can run. The most number of servers is 4 for a single test, and 5.5% of tests need more than one. The servers are currently started and verified serially. One of the 4 server tests (2002) takes 3.2 seconds to start and verify the servers, then only 0.10 seconds to run the tests. If servers could be started in parallel instead of serially, this could cut down several seconds for each of these tests. However, 94.5% of tests wouldn’t benefit from this and parallelizing the entire test suite will allow other tests to run while the servers are starting and being verified, so this change isn’t worth pursuing in this project.

Server Logs Lock Timeout

Tests frequently pause with “server logs lock timeout” messages in some environments, which introduces a 2 second delay. The reason for these lock timeouts should be investigated, but the waiting does not use much CPU. However, parallelizing the entire test suite will allow other tests to run while the logs lock timeout occurs, so this change isn’t worth pursuing in this project.

Slow PUT Tests

Some tests take ~1 second to run (most take <0.1 sec) for no obvious reason. For example, in the first 99 tests these are inexplicably slow ones: `10 33 58 60 80 88 98`. They all use PUT, however, so there may be something slowing such tests down in the test suite. The test suite detailed timings show that the curl command itself runs for the full 1 second, so it may be waiting for a slow server to respond.

Solved: Some further investigation showed that curl was sending 100-continue and hitting its 1 second timeout for most of its PUT tests. I’ve submitted PR #10740 that fixes curl and drops 1 second off 30 tests, saving 30 seconds per test run.

Other slow tests

If it were possible to reduce the time of the 5% slowest tests (76 tests) down to the median time, it would save 221 seconds on the total time run, or 33%. While this would be a nice improvement if possible, it would not be game changing. It would likely also be impossible to do, since most of the slowest tests are deliberately slow due to test timeouts and can’t be sped up without spoiling the test. Profiling the test suite may show inefficient use of CPU in the test suite (or curl) that could be ameliorated, but a quick run with a profiler didn’t show anything obvious. Additionally, parallelizing the entire test suite will allow other tests to run while slow blocked tests are running, so this change isn’t worth pursuing in this project.

Miscellaneous

By profiling the test code, both the test harness and servers, other opportunities for speeding up tests may be found.

I’ve added one of `timeout`, `SLOWDOWN` and `DELAY` keywords to most tests that use timeouts or slowdowns in their design to allow them to be skipped for gathering statistics or skip for faster test runs.

5. Parallel Testing Design Sketch

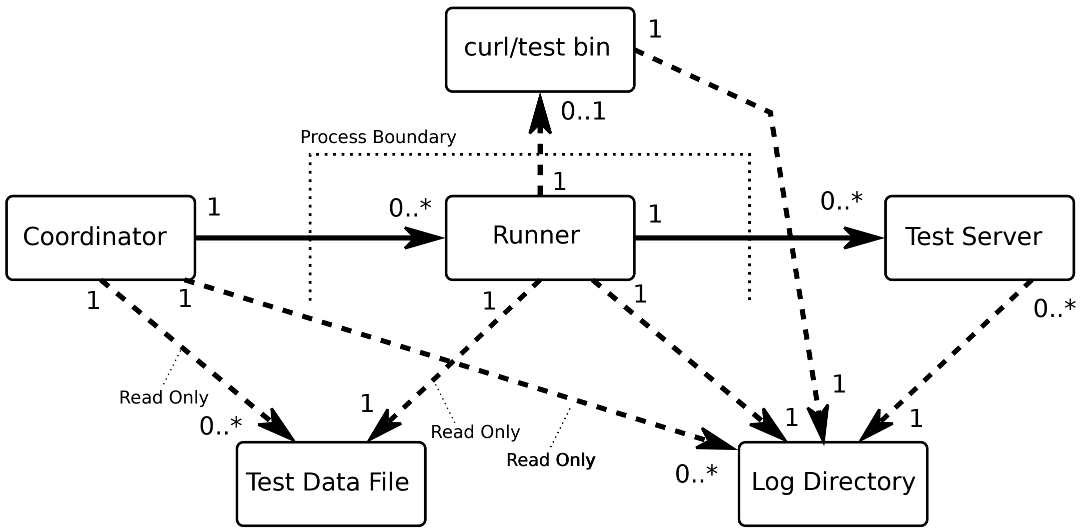

Figure 2: Parallel Testing Design

- There will be one process per test runner (“Runner”). There will be more than one runner per CPU to guarantee over-subscription and full use of CPU resources, probably on the order of 10 to 1.
- A central test coordinator (“Coordinator”) will communicate with each test runner and pass it a new test to run once the previous test results are returned. It will be advantageous to pass new tests to runners that have previously run tests using the same protocols, as their servers will already be running.
- The coordinator will be responsible for displaying all normal output to avoid output from different tests from being interleaved on the display. The coordinator will perform diffs of output and log dumps as requested by the user. The coordinator will not assign a new test to a runner until the output of the previous test has been written, so that the runner may delete the files before starting the next test. This does not include `−d` and `−v` output, which will be output directly to the display by the test runners. To avoid confusion, parallelism should be disabled by the user when either of these options is chosen.

  It might end up better for the runner to create an output file with diffs and everything and have the coordinator just dump the file instead of performing a diff. If the both the runner and coordinator need to to exactly the same protocol output diffs for all relevant sections, it doesn’t make sense to duplicate this work. It appears that the runner will exit with an error leaving check-expected and check-generated available, so the coordinator probably just needs to output the diff of that file and dump all the others, including an error message generated by the runner that contains “1130: protocol FAILED:” without actually needing to know exactly what failed. This will depend on how many kinds of output a runner could create; I’ve seen about three in the wild, that would all work with this scheme.

- The coordinator will send this non-static-global data to each test runner to start each test: test number
- The test runner will send this data back to the coordinator when the test is complete: test result PASS/FAIL, test failure message (e.g. "10: protocol FAILED:", "RUN: failed to start the HTTP/2 server", "lib1515.exe returned 125, when expecting 0\nexit FAILED"), test diffs (probably implied by presence of check-expected and check-generated files), test file names to log with -p (maybe not: just dump everything in the runner's log dir), test timings, which test server failed to start (to skip future tests that need the server).
- Each runner is responsible for starting any servers and creating any files that the test requires. It is also responsible for stopping servers at program end and deleting log files as necessary once the controller gives the all-clear after test completion.
- Each test server must be able to run alongside other test servers of the same protocol. There must not be any file or port conflicts. This includes server command files, log files, test input files, etc. This is done primarily by providing each test runner with its own log directory since test servers have recently been changed to use arbitrary port numbers.
- Each test runner will create its own unique directory under `tests/log/` in which to place files for its servers and tests. Test servers will be changed to use this directory as its "root" instead of `tests/log/`. There are 85 tests that will need to change to use a substitution variable instead of hard-coding `/log/`; almost all could be fixed by changing the `%PWD`, `%FILE_PWD`, and `%SSH_PWD` substitution values instead.
- The test substitution variables like `%FTPPORT` will be substituted by each runner since they will no longer be globally identical.
- Tests 1112 and 1117 should be revisited to reduce the delays they introduce while still adequately testing the code. These two tests take around 24 seconds each to run, which will likely be the limiting factor in how fast the test suite will be able to complete on the most powerful CPUs.

6. Parallel Tests Implementation Considerations

- The amount of parallelism must be tunable by the user. The `runtests.pl -j N` option will work like make's option to run this number of jobs in parallel. By default, it will be X times the number of CPUs (possibly X=10). Specifying `-j 1` will run tests serially, as before.
- Test output should remain substantially the same as currently, although tests won't necessarily be run in the same order. This means that the test number and test name must be immediately followed by that test's results and not another test's. In `-p` mode, the test logs must be displayed after the test number and name, and not interspersed with other tests that might complete while the log is being output.

- Portability is a concern. Tests must still be able to be run on Windows, and ideally, all systems on which is runs now. This may mean allowing a degraded fallback which does not rely on a multithreaded or multiprocessing test server.
- The number of additional Perl modules needed in an updated test server must be kept to a minimum to support parallel tests. Ideally, no new modules would be required but the system will fall back to serial tests if some modules aren't available. This may require vendoring one or more modules with a compatible license into the curl source tree.
- Total RAM usage is a consideration. However, more powerful CPUs with more cores typically also have more RAM available.   With 24 cores, 10 runners per core and worst-case RAM usage of 128 MiB per runner, that would total 30 GiB, which could be expected to be available on such a powerful machine. However, much of that RAM would be text shared between processes, and smart dispatch algorithms mean that most runners wouldn't need to start all the servers, saving even more RAM, so in practise, considerably less than that amount would be needed. A nearly best-case scenario with each of 120 runners running a single server and using 13 MiB means a total requirement of only 3.0 GiB of RAM.
- The pool of servers would best be managed per runner, since each server will be given a directory to look for its files and that directory will be unique per runner.
- If the user starts running a single slow test, there will be absolutely no test output until the test completes. This is because the coordinator must wait for that test's output first, to avoid interleaving output of different tests. This will likely happen also if `-j 1` is given.
- It seems that the coordinator is in a better position than the runner to decide if a particular test should be run or skipped. The determination requires first parsing the test file, and for efficient scheduling of tests the test runner needs to parse the file already to look at required servers, so the coordinator might as well make that determination.
- On shutdown, all test servers must be killed. This will need to be delegated to the entity that started them, likely the runners.
- Tests will not necessarily complete in numerical order. The coordinator could be made to produce test results in numerical order by buffering all output until the next sequential test is complete, but this could result in buffering absolutely all output if test 1 ends up being the slowest one in the test suite. Out of order test results should not be a problem except possibly to humans, and even humans may have a problem noticing when 150 test results per second scroll by on their screens. The test summary at the end of the test run will remain as it is now.

7. Special Test Modes

Valgrind and torture tests are special test modes the implications of which are not being considered at this time. Conceptually, there isn't a problem in running these in parallel as well (and they could certainly use speeding up), but they are lesser used and will wait to be supported until regular tests have been parallelized. As for gdb-enabled tests, they make no sense to try to parallelize.

8. Test Scheduling

Choosing which tests to run in a parallel dispatcher takes careful thought. Several approaches come to mind:

- Partition the 1585 tests into 240 batches and hand each batch to a single runner.
    - Pros: Simple.
    - Cons: Tests don’t all run in the same time, so some runners will finish first and be starved. Some runners may need to start many servers to satisfy all their tests
- Hand the tests for protocols that only ever need that protocol’s server running (and no others) to runners that serve just that protocol, and all tests to a single runner that runs all the other servers. This would currently partition the mqtt, gopher, dict and smb protocols into separate runners.
    - Pros: Can be implemented without making the test suite fully parallelized. It is good for early testing as a proof of concept.
    - Cons: Will not keep the runners very full. Will break down if a new test is added that uses one of these special protocol and another one (this test should be caught an the test server should fail if it occurs). Will not speed up tests much because of the limited parallelism.
- Have uncoordinated runners assign themselves one test at a time as each completes another test.
    - Pros: Keeps runners busy at all times.Possibly simpler since no central coordinator is needed.
    - Con: Need synchronization between runners to avoid accidentally running the same test twice, such as with locking scheme to grab tests from a pool. Runners would be duplicating effort in parsing the same test files to find an appropriate one. Runners won’t have the ability to skip inappropriate tests for them since there’s no guarantee another runner will pick it up. Test output must also be synchronized between runners or output from different tests will be interleaved.
- Have runners assigned one test at a time by a coordinator as each completes another test called Coordinator-Runner.
    - Pros: Keeps runners busy at all times, no wasted effort in multiple runners parsing the same test files at the same time. The coordinator can spawn a new runner as necessary instead of pre-spawning them all. Test files are parsed efficiently; the coordinator parses the next test data file and passes it to the next free runner before moving on. A coordinator can assign tests to runners based on the global needs of the test infrastructure.
    - Con: Test files need to be parsed by the coordinator, then parsed once again once it’s assigned to a runner.

Following are more detailed descriptions of two possible implementations of a test coordinator.

Simple Coordinator

This scheduler consists of a coordinator process that scans test data files and decides which runner shall execute each one. The simple scheduler would just read the next test data file, determine if it to be executed or skipped, and if not skipped, hand it to the next runner that comes available. This scheduler will result in high RAM usage and higher overhead as many duplicated servers would be running for different runners.

Optimizing Coordinator

This is the most promising scheduling architecture.

This scheduler also uses a coordinator process, but keeps track of which servers each runner had to run for previous tests. It would read ahead the next N test data files, where N might be something like 30 (or possibly many more). As each runner came ready, the coordinator would search the next N tests to find one that the runner is best capable of handing, hopefully a test that uses servers that the runner already has started. Any runner can handle any test, but some runners are slower because they will need to start a new server to handle the test, and with each extra server started comes extra needed startup time and RAM.

Since some tests naturally take a long wall clock time to run, it would be advantageous for the scheduler to run such tests first, so that others can run while they wait. It's suboptimal to run 1584 tests in 10 seconds then start the final, 1585th test, and have it nearly double the total test time. Since the scheduler parses the test files to determine which to run next, it could use a new test keyword or feature there as a hint to run those tests first.

A possible test dispatching implementation might be built around a pseudocode loop like this:

```
while tests_left_to_run()
    push_parsed_test(parse_next_test_if_any())
    while runner_is_ready() and parsed_test_available()
        start_runner(pop_best_parsed_test())
```

This will parse one test and placed it into a stack of tests ready to be run. If a runner becomes ready, the "best" of the already-parsed tests will be handed to it. If no runner is available, the coordinator will continue to parse tests instead of blocking waiting for a runner to become free, to avoid a parsing delay once a runner does become available.

As discussed earlier, if test parsing remains in the critical path, it may need to be performed in multiple threads to keep ahead of the test runners executing them.

9. Implementation Plan

Each of these steps will result in at least one git commit, which can be submitted immediately upon completion, independent of further changes. However, some steps may make more sense to submit together with following ones.

1. Refactor runtests.pl to separate the test selection path from the test execution path. This will enable later refactoring into controller and runner processes.
2. Allow all servers to optionally run out of a subdirectory in tests/log/. This may involve changing or creating new template substitution variables that will require at least some test data files to be updated.
3. Refactor out runner code from coordinator code. Put runner code into runner.pm so it's clear what code will be running in a separate process. Runner code will include the test server and log file lifecycle management.
4. Create and use structures for passing data to and returning data from a runner, suitable for use in an IPC mechanism.
5. Create one separate runner process and have it perform its current duties in that process. Use an IPC mechanism between the coordinator and runner.
6. Create five separate runner processes. Use a simple test scheduling algorithm that sends tests needing a gopher server only to the second runner, a mqtt server to the third runner, dict to a fourth runner, smb to a fifth, and all other tests to the first. These protocols are chosen because currently no tests require any of those servers in conjunction with any others, so they can most easily be run in parallel without conflict. This step should not be checked in independently because of the possibility for confusion about the restrictions of this limited parallelism.
7. Ensure there are no further restrictions to having multiple servers of the same type running under the control of different runners.
8. Create a new test scheduling algorithm that better utilizes the five runners, namely a simple scheduler that passes the next available test to each runner as it comes free.
9. Create N separate runner processes instead of just 5, where N is not too large because of the RAM requirements if the simple scheduler.
10. Create an optimizing scheduling algorithm that tries to reduce RAM usage and startup time by choosing runners based on which servers they have already started. Bump the default N as high as practical.