Solidity Compiler Tools Design Document

Killian Rutherford (KRR2125)
Raphael Norwitz (RSN2117)
Jiayang Li (JL4305)

1. Problem

Compiling large solidity files, particularly large solidity libraries with multiple file, can be quite time consuming with the current compiler (some cases taking several 10's of seconds). The tool we developed would enable users and developers of the solidity compiler [1] to profile pieces of code/functions within the compiler to see which areas were consuming a lot of time and acting as bottlenecks.

2. FTime-Report Tool Design

(See Tutorial for Introduction and User Manual for in detail about every element)

All our code is located on our github page https://github.com/raphael-s-norwitz/solidity [2].

Our initial design option was to produce a lightweight solution that would allow flexibility on the user/developer side in terms of what code to profile. This initial design is produced on the dev_tool branch. These flags would not alter any core compiler functionality, but provide a function call trace and time readings for each function call in the call stack.

Hence we opted for a stack-based approach for simplicity and efficiency, with TimeNode classes to hold information of begin and end times when the user pushes/pops onto the stack, names, and vector of TimeNode representing its children. We referred to gcc's implementation of --ftime-report [3] when designing our tool.

Chrono::high_resolution_clock was utilised for time measurements given it's accuracy and what we learnt in class.

Initially we wanted to have the time profile code be implemented as a base class which other classes and functions might be able to inherit from to print out information, but this was too challenging/not possible.

We decided that if a developer wants to benchmark a function or segment of code he/she will have to #include <libsolidity/interface/ftime.h> in the file; add t_stack.push(string) at the beginning of the function/code, and t_stack.pop() at the end. This adds flexibility on the user's behalf, as he/she can decide what is to be tested, how often within a function/class to test, and give appropriate string descriptions as to what is being tested.

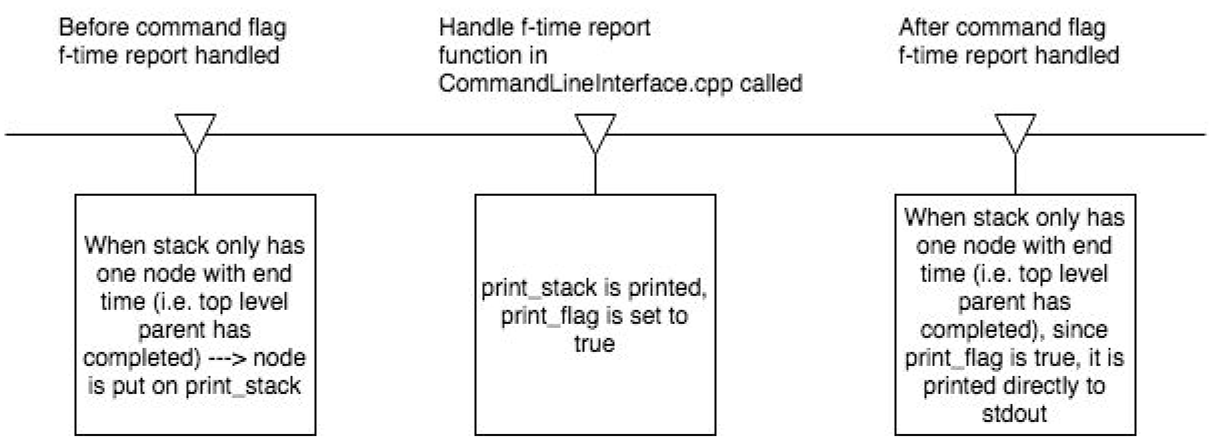
TimeNodeStack holds these TimeNodes; so that whenever a user pushes to this stack given a string argument, a new TimeNode is created with name equal to argument, begin time at the time the user pushes, and put on a stack represented as a vector.

When a user pops, if the stack has multiple elements, the top of the stack is popped (end time of this node is set) and this TimeNode is added to the stack's new top item's children list. This makes sense from a design perspective as no pointers are necessary for a TimeNode to point to its parent. If the tool is used correctly (no errors are thrown), and if the stack has more than one element, it means that the user has nested push statements (whether in the same function call or in functions called by higher-level functions).

If a user pops when the stack has only one element, it means the end of the recursion in terms of pops and pushes for the highest level call has occurred, and this can be either printed if it happened after the compiler handled the --ftime-report flag, or added to the print_stack so that the compiler can decide whether to print the report when it checks the value of the --ftime-report flag.

Having the user write t_stack.push(string) at the beginning and pop() at the end allows flexibility on the user's part in terms of being able to profile multiple times and multiple positions within the same function block [2].

libsolidity/interface/FTime.h defines a global TimeNodeStack, which the user can push to/pop from. It is necessary to have a global variable if the user wishes to profile code within the compiler that gets executed before the flag --ftime-report is handled. This comes hand in hand with the print_stack mentioned before. Since the developer can choose to benchmark code executed before or after the --ftime-report flag check, in order for the flag to work properly (i.e. data can only be printed out if the flag is specified), another print_stack needs to be created to store everything that is benchmarked before the run-time argument is treated. Then, another flag is turned on to print everything that is benchmarked after the run-time command line argument parsing code has been executed. (See diagram below for clarification).



However, users can also define multiple t_stacks if they want (this came in handy in testing), and these can become very useful when linked with TimeNodeWrappers (discussed below).

We also added a printString function to print the print_stack to a stringstream rather than dumping to stdout; to enable test cases to be handled much more easily.

We decided to only measure in microseconds because we expect that any performance hit in the solidity compiler can be measured on a reasonable scale in microseconds. Also, we hoped to build a handy tool for users and therefore wish to eliminate any unnecessary the parameters from our API.

We also incorporate the tree optional flag “ftime-report-no-tree” to give users more flexibility. If they do not want to print out in tree form, they can specify the flag and have the plain form.

We noticed that in many functions where there may be several early returns, we needed to put pop() before each of them. This can be time-consuming and error-prone since a user may not notice where an early return happens. Hence we decided to incorporate RAII into the tool to handle resources on the stack more intelligently (found on dev_tool_wrapper branch).

We created a TimeNodeWrapper class which would minimize the number of pop() calls needed to profile a complicated block. Using RAII principles learnt in class, TimeNodeWrapper object is instantiated with a given_stack and a given time node name. In the constructor of TimeNodeWrapper we call push on the given_stack, and set TimeNodeWrapper’s private stack the same as the given_stack so that we can refer to given_stack when destructing the object. The TimeNodeWrapper also has a boolean member called popped that is false by default. If the user manually pops by calling TimeNodeWrapper’s pop() API, the popped flag will be set true. When the object is being destructed, the destructor will check the value of popped and it will only pop from the given_stack if the boolean value if false. We believe this design offers users greater flexibility when profiling function blocks in the solidity compiler code [2].

Further, the destructor pops from the stack, it will check whether the name returned from the pop operation matches the object’s name. If the names don’t match, it will throw an error. We are working under the assumption that objects are destructed in the reverse order of construction; as one would expect from how memory stack allocation works. This allows the user to define multiple TimeNodeWrapper objects within a block of code or function in order to profile different sections (such as loops for instance). We decided that the constructor to a TimeNodeWrapper must take the stak and a string. We thought about writing another constructor which does not require a string argument to name the Node, and would instead take the function name within which it was called. However, this would make it harder to define several TimeNodeWrappers within the same function block, (an extra count variable to keep track may have had to be added); and if two functions in different namespaces had the same name, it would be hard to keep track of which was which. We hence thought it best in terms of simplicity and flexibility to keep the string argument in the constructor [2].

3. Optimisation within Compiler Source Code

In the beginning, we noticed that running the test script was taking a lot longer on some machines, we investigated, running on Mac vs. linux, but turned out to be optimisation time differences whether building with make or ./scripts/build.sh. The latter built the 'Release' version solc while the former built the 'Debug' version [1].

After creation of version 1 of the tool, we began to use it to see how we could identify the slow passes of the compiling process and possibly optimise on various things.

Looking at section 3.a of the User Manual, we can see the results from further testing the optimise() functions within the compiler. We see that the applyMethods loop is taking the most time. (code seen on dev_optimise_profile branch).

The bottleneck seems to be that each of these methods (PushPop, OpPop, etc..) is applied sequentially. We attempted to write out the applyMethods iteratively, calling each Method instead of recursively calling a variadic template function, this did not seem to improve the timing.

Further attempts to take out the template metaprogramming were unsuccessful. Their approach of using static functions avoids may constructor calls, while complicated, seems to be the faster than any naive approach we can come up with.

We also considered calling these variety of methods asynchronous, but we realized it would be complicated and involve several locks most likely since the optimserState would be altered, and actually these methods may need to be called in a specific order.

For results in investigations into the analyse pass of the compiler, see the User Manual section 3b. Attempts to shave off time to refine the source code were unsuccessful.

4. Parallelisation

At the moment, solidity lacks any kind of parallelism, and this seemed like the most promising way to get speedups.

As discussed before, once the ./scripts/build.sh script is run, and the solidity executable is produced (solc); we compile a solidity source file (.sol file) by running solc Coin.sol for example, along with a variety of flags (bin, optimize, ignore-missing), including path to produce the binary file (-o dir). The solc executable can take in a list of multiple source code files (and all need to be together if dependencies amongst files exist).

There are two obvious courses to reduce compile-time through parallelisation when dealing with multiple file libraries.

1. Higher-level parallelisation: splitting files across different processors, running compiler executables on each processor.
2. Lower-level parallelisation: parallelization within the compiler source code

One would hypothesise that the first method above would only be beneficial for compiling large numbers of solidity files. From above tool measurements, there is a fixed time cost for the compiler to be instantiated, and thus by doing so on several processors, a certain number of files will be necessary to offset this instantiation cost and make splitting compiling tasks among several processors worthwhile.

Further, it became clear from the onset that option number 1 would only be possible for solidity files which did not have mixed dependencies (if random splitting of the files was applied). In the best case, each processor would have to be self contained sets of dependencies, but this may be hard to identify without some sort of parsing through source code to see which files are included where (probably too hard to do non-manually).

If files are unrelated and have redefinition of contracts, it will pose issues and potential errors when compiling them together.

A lot of libraries which have inter-dependencies among files would hence make it difficult to apply the first optimisation technique.

Initially, a simple shell script was written to split the compilation across different processors.

The number of files given a certain number of processors on the machine could be calculated optimally using UNIX commands. Simple division and modulo remainder can split everything optimally [2].

E.g. given 10 solidity files, A.sol - J.sol, and 4 processors, we would compare running:

solc A.sol B.sol C.sol D.sol E.sol F.sol G.sol H.sol I.sol J.sol
(on 1 processor)
Vs.

CPU1: solc A.sol B.sol C.sol
CPU2: solc D.sol E.sol F.sol
CPU3: solc G.sol H.sol
CPU4: solc I.sol J.sol

Initial tests of with vs. without parallelisation showed some potential for improvement. Although not taking into account dependencies correctly in some cases (i.e. checking for correct compilation), we saw before parallelisation on a google instance with n1-standard-4 (4 vCPUs, 15 GB memory), over 10 runs an average compilation time of 4.81 seconds with stdev 0.014, whilst with using parallelisation- 4.23 average with stdev 0.05 seconds, giving an approximate 12% speedup. However, we noticed that using a bash script was not the best idea for portability across systems and the potential overhead it introduced. Additionally, no checks were done to make sure everything was compiled correctly [2].

Hence the switch was made to use async in a cpp file instead to document and measure everything properly (dev_optimise branch).

Using system() to call the compiler with relevant flags, and thread library, all files were split in a vector according to CPU and run on different threads using async (with the launch::async flag set), and get used to obtain the future (stored within a vector of futures).

Although there are issues with using system() such as portability (https://stackoverflow.com/questions/8086071/are-system-calls-evil) [4] , we needed to be able to call the solc executable, and we assumed since the solc executable is portable, we were within right to use system().

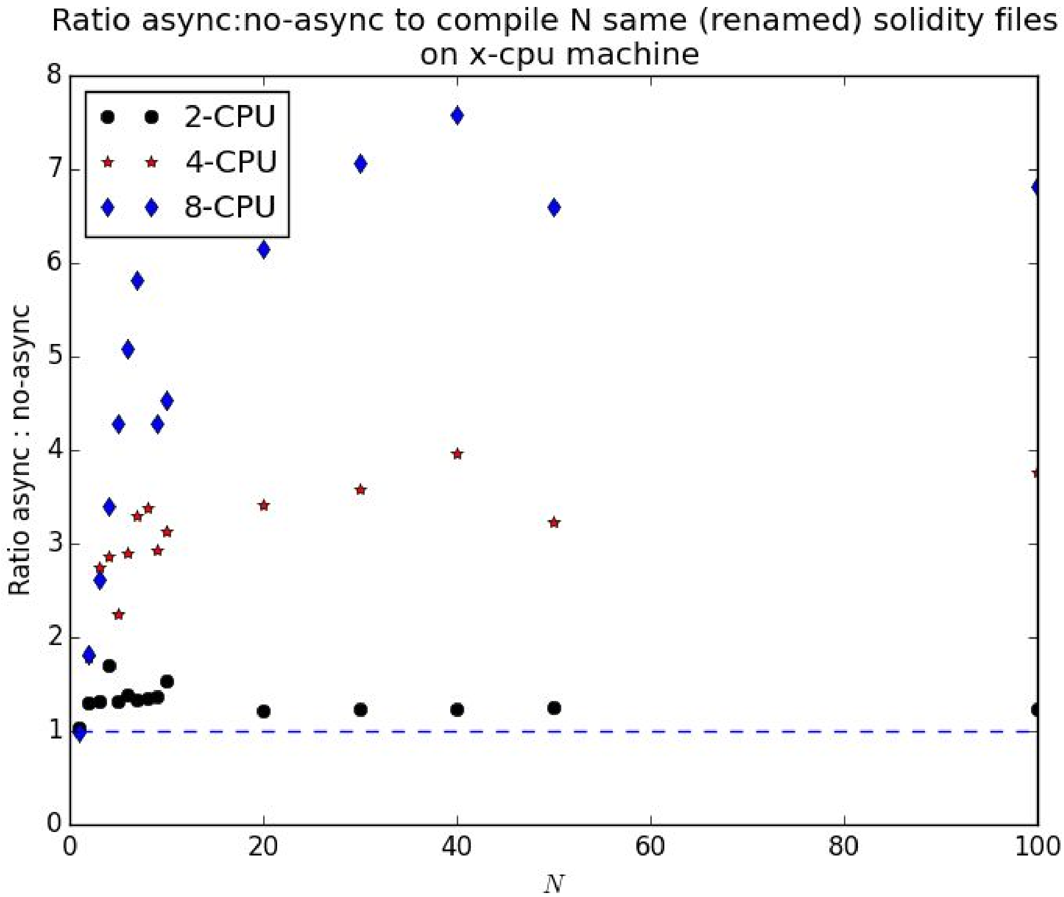
We ran into dependency issues with the test libraries provided by the solidity community, hence to determine whether parallelisation was helping, we needed to decide upon a dataset in which we know all files could be compiled independent of one another.

It may be possible to figure out which source files need to be run together given their dependencies, but for larger libraries (unless some automated tool existed), manual selection and reading header files would become very impractical. As mentioned in further work at the end, it may be possible to build a topological sort of the files in terms of when to compile them, by checking in the source files which ones import which.

The first thing we thought of was to copy a file numerous times within a library, however this would give compilation/dependency errors; hence we moved on to selecting a random contract, and copying it a certain number of times, while altering the name of the contract classes and functions to prevent the compiler from optimizing and predicting / cache hits.

This was done with BlindAuction.sol file, which contains both a BlindAuction contract and a purchase contract. (see https://solidity.readthedocs.io/en/v0.4.23/solidity-by-example.html for the files) [5].

Time ratios were averaged over 3 runs and results over different number of google instance n1-standard vCPU are shown. (n1-standard-2 (2 vCPUs, 7.5 GB memory) ; n1-standard-4 (4 vCPUs, 15 GB memory) ; n1-standard-8 (8 vCPUs, 30 GB memory)



We can see pretty quickly the benefits of parallelisation and having multiple CPUs available. As soon as N goes beyond 2, we see ratio increases, most likely due to the fact that this particular solidity file is large enough so that compiling all of it offsets the fixed cost of starting another executable on another processor. For larger number of files we see a larger separation and approximate doubling of ratio with doubling of number of CPUs.

We also wanted to test out parallelisation on a more realistic data set (i.e. not just copies of a specific file) to see what could be a potential “real-world” speedup to benchmark and to avoid any thread taking advantage of cache memory. We were pointed by the solidity development team towards https://github.com/allenday/github-solidity-all [6], which contains all solidity files on github in one place.

Rather than selecting x number of files and manually checking if they could be compiled/split across different processors, we automated this with C++ code and the system() function call.

Initially, we decided to run through different files in the github solidity all repo + the test files provided by the solidity community. We would compile the first file, using try-catch to catch any exceptions from the compilation. If any errors were caught the file would be discarded and the next one tried. We would keep on adding files this way until we obtained the number of files wanted.

We quickly realized that system() was not throwing exceptions, but would return an integer code instead; so our first check to see if the files were being compiled together correctly (before attempting to parallelise) would be to check if the return code was successful (0), then add the file to the list, otherwise discard and move on to the next file.

However, even only checking for the return value was not enough, as in some cases a correct return value was returned, but the compiler did not actually compile (specific error was the respective binary file was not generated as a compiler version was not specified in a particular solidity file). Hence, the check for correctness would have to be whether a correct binary file was produced (which was achieved through the --bin flag). With these, we were able to obtain 100 files from the general solidity files on github [6], which were able to compile independently and move forward with parallelising.

100 files:
8894 lines; 33,362 words; 294,515 characters

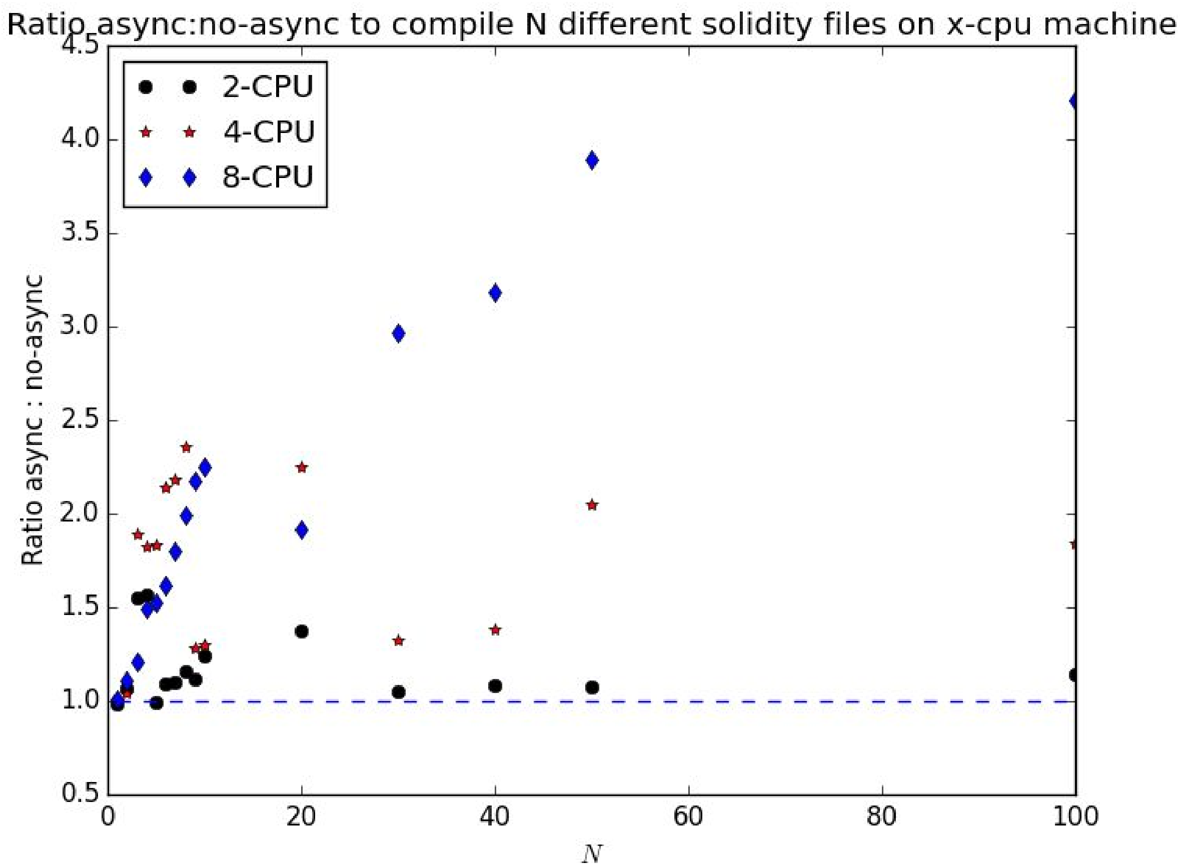
```
  73   187   1955 ./test_files/test_a7dbaa67d524dff15145be4b36f6d41f7b245e8221fc4004e05677268dd33a52.sol
  24    78    454 ./test_files/test_f1ddb3ec5142f65be5e1e26a6a476aa8f06d2c83e9647968d2f27041a9b4c980.sol
  73   187   1955 ./test_files/test_ce630e9d43f8716989b99d88bf00b32fb101a429574ad3ed92df0cfb64f979a2.sol
  53   197   1106 ./test_files/test_cf7f78c987171edefc69321edf6e5b9f8ec2e4b0dbfd0c0dd43367a034c9f16c.sol
  13    73    700 ./test_files/test_8da06f832f837a554e151371368750f2b39a2f749073e7261d7ad02dc61a10ee.sol
  78   302   2521 ./test_files/test_beb72a1741f79c88d654d63439099a12e8ae8ea39330f8a3a31ca4bfac78ca59.sol
  14    53    445 ./test_files/test_a9cf1d9073a8a58ca6044a1720a93a69020ea80fab3f5169192630590707d593.sol
 361  1336  12385 ./test_files/test_41145dd32e305cda546cada00805fe647a7db9dcadee99ed4a107f075144b59b.sol
   9    23    201 ./test_files/test_84025419be75907457e68902da17c2ba380ebb86aeda73a1701501d03bcb319e.sol
  98   370   2341 ./test_files/test_f05adf65bd6a71de586500a74b8e07a9f4a70625e90c55e93ebb5e400ddfdb41.sol
   6    27    158 ./test_files/test_2d4cd83f17573d726a08d119313fefaaae9ba47f8439e6247e06057e295452ba.sol
  55   178   1859 ./test_files/test_ced95337821dd3febb160aedcc50ad760eb75971924657e14858606aaf1fdf29.sol
  31    53    506 ./test_files/test_eff0a9670ebf709ee0b054bd3dd7fde373314e4fea9348803eccab5cccd3dd80.sol
 102   450   3487 ./test_files/test_f05dd9baccc2604127cf72d2038b5c42fac0a832d44b210f129f138cb400f2b5.sol
  13    35    281 ./test_files/test_e2fd2d93f8741884e1b9d0808d0724a13c9082da5e247ed6f8f46e26dad60719.sol
 281   856   8502 ./test_files/test_180ac4cf9e021cdd42c556aa203e951aab4a8a0fb52a7b436fa9b5977f7e998a.sol
 109   500   3923 ./test_files/test_729c48d31247252239590ba80a77aa63be3ef9feeddc1aef92ba095f07371033.sol
 370  1102  11515 ./test_files/test_c4c8cf07bc07d54727a24fa9ab84e7353637274d213406ba663bf1848ad696e6.sol
  19    45    500 ./test_files/test_27548d90ec169921a0c7ca596b266088059e2bb91976225ecab57be131e002a7.sol
  23    82    701 ./test_files/test_caf1d08f8255ca913796af6a5dcf3198b250ea8b864c7293cb815ade0841b0f7.sol
   7    20    101 ./test_files/test_1a92c9e1ad65fb0e2a67ed5f7e71cb5f2f640e701f5ca69c56abc3b175c6a111.sol
  79   195   2367 ./test_files/test_ebfcda9f83b927375ead8de9a0bc57dc23a9326cd204ea24797b62d80b887b99.sol
   9    23    202 ./test_files/test_aa49ec7f336c6fea1204a2a4152c63dbd56ddeca0c1a348bbd51783dd671ed0e.sol
  15    79    747 ./test_files/test_c5eeede822366b11933471f25adbbe999e1c3e993ea881af9ca7e63c79b90df9.sol
  26    84    596 ./test_files/test_cc48ccd774362b86a68ac59b272eaf32501e4aa87c93d5a38c6df55964cc854a.sol
 800  4066  32437 ./test_files/test_b32af89dda8f16ada02ed5a3dd800f3fe17230bc845c1483d0f4a2ed7faa78aa.sol
 708  2238  24511 ./test_files/test_b1d17622bfd063291ef15adf99fa8860ac01f994ac631fada2660be73152d1ae.sol
  15    54    392 ./test_files/test_a9158c87554cd9f67dee4e450f01b3335cd86eb91ee546b5855d48a574d12800.sol
  60   229   1719 ./test_files/test_6258bfb64990fa7fe594dda3a0eafcf12cd3d9113eaf5a9e5d07589cc0d47b2e.sol
```

```
592   2984  24208 ./test_files/test_881cbfc5553c77da2fbeee2086dc100069d058980405477e0327da491b0ff320.sol
 32     99    877 ./test_files/test_f707594dfb6630bf2bc3412854dc6dcb7be1aa82889431e073086ceefefc1b3f.sol
 42     92    704 ./test_files/test_55d6d7a52cac81df9056bc530b703b7aa6a9d6f319ab1aea3bb3920cb9975cba.sol
 27     56    504 ./test_files/test_94cd7a46e80fbfb80989bc9b2d712e7f3606683f23762e534ee4b558085597e6.sol
 20     35    339 ./test_files/test_a389cb0ed307d0fdf0496684d8b0acf01a8be2603f32fb4f528a0ef146328cd1.sol
 18     51    398 ./test_files/test_d23f90ac6b6d7c5654023b79bbd39281fbba741dfee8bb6ec3385ab30f542530.sol
  6     19    146 ./test_files/test_0e08e1aa8ac483d1d2683286aac69dc0229ad0ea5955fceea03ba27e27e14e94.sol
 17     32    270 ./test_files/test_2e67c6ba377ef76a380cb76d53ff5b254d123c649825f27b2c7dc2b7bdd73a30.sol
 23     48    450 ./test_files/test_b0eac34cd208612102c3aa9d228bc9cc5b8101313bbeb59f692955f0b60fd719.sol
 90    335   2959 ./test_files/test_c240ac4f115b4790eedc59a555d6c75640a9ec635ee94b9a252292bed07117ca.sol
172    429   4370 ./test_files/test_27b57c491d76d89275c92bfe517c7c466f9c5baaa420d585b0780ebe88aa5032.sol
 11     20    174 ./test_files/test_53f3679909545e5cb34d838087dd16ffd75f8907cdf352168b2b2dbb08862f1d.sol
260    946  10439 ./test_files/test_00fef28dbf32dabeeddda75bed666550761cb4a58fb49bdb2779bc5c96fe6fb7.sol
 53    213   1707 ./test_files/test_24b5acc4187857057a3b3acc67e8d375c1e99f85227d028ebe3143a24e02ca5e.sol
 20     35    381 ./test_files/test_2600be00203e59b5ea26b972e18058bd3e3e3b51a20b9caf77b295466a96cfcd.sol
 91    343   3394 ./test_files/test_bae4d162fd7d404dbbb62da7c6f00cacac4e95ccb283328f485146141728376e.sol
 23     52    515 ./test_files/test_1073beba092f03f5d3425f42d7b5b1df41f37f9949cb339126a12105d97df320.sol
  2      4     15 ./test_files/test_69dbaef3b2304c096a355e9871d6bf70e654b22b97c617210bb6ccd270b874c5.sol
220    702   5220 ./test_files/test_de24c86b1a6b9fc3158132ba43f6710091fa1d6637dc335514d84f7bd0973307.sol
 56    133   1800 ./test_files/test_cc279795bcb1be51ca7379cbd096b82301f3cd0c6c80b05a6644b90d86130776.sol
 23     49    493 ./test_files/test_d1ba9cc38ae66d7c7b6f2248262205cbe5d76452608f92e0428c2174c7a59376.sol
 99    516   3949 ./test_files/test_c5c14d4ce17800de3fa979a48306518a82c35acb8240e426c586d447c2db1d65.sol
  8     32    463 ./test_files/test_dcda0ee37c687155107d55c5ea7c64a1b30ec8329beb139fd1315f215938be06.sol
  9     20    140 ./test_files/test_30b8209f11035d7ab60d8335e970f58fc7a592c2780497e9a3d07e3a08b3c277.sol
 17     56    631 ./test_files/test_ee27b9d841c89e58b670db6d7c2096b0cebd621e4224293caa2ac3074ea678de.sol
 31    125    866 ./test_files/test_b2a1e02d34d25a72ad5a005795d9305be4dd6009b6c31122fbcb8e0963536489.sol
 24     49    495 ./test_files/test_079bcf654c9fb9acd811faf181ffa8dc2921a059dd430135fc474e8f427422d9.sol
 20     56    582 ./test_files/test_b727e7cdc55f3caef68c6cba823f9047752573090ece5d5a412946c12ed9e53e.sol
 21     67    483 ./test_files/test_d09447013d256b523b67b951ccb5a5719b20e38f9e47a6a10bf28d1e5de31104.sol
 83    183   1978 ./test_files/test_75e04c1d769f47687a3b6db1f040944cff9408be0d5cc8e42038f2aca05f5af0.sol
 26     45    407 ./test_files/test_a1bcc24b41baa69febd36c222d0104c562c6dda12e1175d10168cf3e3aedb73d.sol
364   1098  11381 ./test_files/test_1dd8c290ab7230fc1267d8ee3856f6578d3e2235502aacad51f7a21f1594408e.sol
152    678   5653 ./test_files/test_391ed7631d9a1183c6d6293defce1925053bc0f056e7650a762122003e31a16d.sol
269    773   9115 ./test_files/test_7166048567b924db185cd3804f51f6f3969af314255f8494c8cf8338e06288cf.sol
  4     12    107 ./test_files/test_5f913b422c3a08a1d37d6264c43d5faaa361cd2ff14669ee0b7231d618660c7a.sol
103    274   2196 ./test_files/test_ff599bc711526922add29d87d049be740bf4638a3cbadda297e462d721f85a92.sol
 91    311   2476 ./test_files/test_543ac5f8897d242b076187b3d9a169b1dac4538f4229240f8819bc30b2952f49.sol
 41    205   1441 ./test_files/test_af75ba8c2005cdbd2e31bf9cf0e6e510fc64c1f9bd2a621347d537f14caeffa0.sol
 46    118    645 ./test_files/test_40533309e2d270c161b9d328f331d028883c319de13cce6dff878dcb9b9480e2.sol
 13     33    321 ./test_files/test_14c7a213c9ee60b1e241b74e196892fd0675f46a8f4bcd3b27acbeea0bf7eca6.sol
713   2968  24859 ./test_files/test_d83f6fd57215fe8c796d4a8f8c08375b206fee9e9da061fb4d2861bc7feb3fa3.sol
 45    100    861 ./test_files/test_145d369f014d451f55bc43acc7eb38d49423046447e14212f61984ba5daa7683.sol
366   1853  14384 ./test_files/test_f11a1bc2f65010799a1a82fc9ca3324b51667841eebf3e95129f54b7ede9e315.sol
 17     53    408 ./test_files/test_80e6a9f1bc2db5c8cec14e462b886ba214c2fe09c748d720a764a6a59575cc3e.sol
 13     53    341 ./test_files/test_94d476d127829e60e66b0ef454ebe732efcca46da3f27f4d0b54c995cb6bd31a.sol
 23     49    493 ./test_files/test_8313e2d3233bffe2cdaac2213160eef056d25a6ad69dfb4aa8c93a78ed6276fd.sol
  0      3     14 ./test_files/test_bc432a88a5ecee79a418334fedd1a53e35ad068d12ac8ee7a59605365b8ace15.sol
 15     29    255 ./test_files/test_d4d2ec889a6558c5961ce550e9fddfc5863818adf4f493895f11c36d52d2627c.sol
 14     76    720 ./test_files/test_fdca83ed60188aa46b3160cdccd28b118b9e56cf92dda047aa993325323393f3.sol
108    427   3920 ./test_files/test_3991bb702569121efda01c9ad3df8b8089a6e5fa17e0f4a30a4976e2858debae.sol
 49    218   1215 ./test_files/test_ac816db87fad29b87c7da4a9c22da62b39d1f1fb808c06b68476ab0e63dc72db.sol
 17     47    369 ./test_files/test_d17c2e430477347046df44382dfc766e1e0efd508cc320a34f91272cfe6ddecc.sol
140    792   6440 ./test_files/test_475316a006533f57931d2ec6efa3e6be030242a63989d77dcfdc8efb4fa60c49.sol
 55    217   1803 ./test_files/test_c76bbfdfbde7969a1103bccc14d042a6f3389948fa42d0c1eea3201e24dceb9a.sol
  4     18    121 ./test_files/test_4fc6fc1064eb624d65550ee72eeb43edbaf8b41966f9b657e99dbe2067376eeb.sol
 18     76    647 ./test_files/test_eec0e7b013ac7adf7ecb18ff453bf187ef4f6978ff7293c79a840f38db77a243.sol
 13     22    162 ./test_files/test_b3e5b2195f09b605e67c8016dabd8167ea904074debe0b1151869d245689d17c.sol
```

```
 144   541   4602 ./test_files/test_d5ba0ba7d5a54e4c4af70c653eff6732f591bc7a0dd0c31c8d277731412d0569.sol
  23    39    299 ./test_files/test_87b1ce31b14175f8a7f66bb5879edd12ff4fbca420ac83a8ebdee48d11f82de1.sol
  13    33    268 ./test_files/test_27329c49aa09c3a886e4ddd8002ca61485f5f53624d353fb0a6d4bc22b911cfa.sol
  25    82    668 ./test_files/test_e6889d6b48abf30a71a3cbb79a0e496521e2c11e7c2d342358f5b481f19bfe2d.sol
  57   191   2466 ./test_files/test_0c1ea4a45c5036b405bfdc915e0a4f859c280eb53c18247f53b1bdc8f5999e46.sol
  21    64    621 ./test_files/test_f4eaa60c512f93cd976c80ecc6a65ea57b3bef16de3ac37af926f650f0b7d0b3.sol
  73   308   2071 ./test_files/test_25e04cd2828edeee0dec5fe14f5da908705ab7a82c510bb25c880adca7cafa62.sol
  55   337   2508 ./test_files/test_b3fff7e6a3bff46da8a8493ebac5edc8dd9665ec8f4fe4f5953caab3f92f25b0.sol
  14    27    225 ./test_files/test_8bf105fee7e33f095ec43ebfda22d3a2a133f72fd4ebb84beab8dda0cf6a26bf.sol
   8    14     95 ./test_files/test_cecddb74312958562b83418eb10f2df74f9be70c76688eed28c00c3729f9bafe.sol
   9    39    260 ./test_files/test_a201fac7f454b4b98880d79141bf8f23a53430a6620c4a3c4b300154ba0be338.sol
  55   130   1429 ./test_files/test_336feb9082de244fa180ce831404e8426e445dc61b3e15ce42ef054130fff292.sol
  27    38    351 ./test_files/test_518fc11b394e60cf0f3ea7e5c397526dbf87e7b6c299af7f14ef3f838f0f94ee.sol
  17    68    381 ./test_files/test_f805a68f0675fb683edb12a6bfa19d777a1b75c6d4bcc6e6502b41db3b511cc8.sol
8894 33362 294515 total
```

Finally, to test if the parallelising compilation produced the same binaries as the original solidity compiler, we ran a python script to check if the binaries compiled on 1 CPU vs. multi were exactly the same. In all of our tests, if two files generated by the above 2 cases had the same file name, they always matched. However, after 50 files we noticed in some cases where the amount of generated binaries differed (perhaps due to having several contracts within one file being compiled to different binaries). Nevertheless, the total sizes of binary files generated in the two cases were similar, so we thought it was still fair to compare compilation times in this setting [2].

We stopped after collecting 100 files due to the large amount of time needed to check if all these files can be compiled without error, and for reasons like the asynchronous compiling process and the original compiling process each generated a tiny amount of binaries that the other didn't produce.



Results shown above demonstrate expected speedups with 4/8 core google cloud instances, particularly over a large number of files. However, these ratios were smaller than the ones we derived when doing the compiling tests on identical files, which is most likely due to large singular files taking over in one specific processor. These large files probably explain occasional dips in ratio seen, as well as discrepancies for smaller N compared to the previous graph.

We have developed a parallelisation tool inspired by the result of the asynchronous test, see tutorial/manual for further details.

Nonetheless, as discussed above and in the tutorial, there are several limitations to using this tool, mainly dependency issues.

In order to combat this, we attempted briefly to look at the second parallelisation approach, which is parallelising within the compiler itself. We specifically focused on the following part of code in CompilerStack.cpp:

```
bool CompilerStack::compile()
{
	t_stack.push("CompilerStack::compile");
	if (m_stackState < AnalysisSuccessful)
		if (!parseAndAnalyze()) {
			t_stack.pop();
			return false;
		}

	map<ContractDefinition const*, eth::Assembly const*> compiledContracts;
	for (Source const* source: m_sourceOrder)
		for (ASTPointer<ASTNode> const& node: source->ast->nodes())
			if (auto contract = dynamic_cast<ContractDefinition const*>(node.get()))
				if (isRequestedContract(*contract))
					compileContract(*contract, compiledContracts);
	this->link();
	m_stackState = CompilationSuccessful;
	t_stack.pop();
	return true;
}
```

Trying to apply async to CompileContract() caused several issues as one cannot apply async on member functions. Using bind was attempted, but unfortunately did not produce good results.
We concluded that parallelisation at lower level would require much more involved and complicated processes; perhaps not achievable (or wanted) within the time frame we had [1].

5. Parallelisation Tool

(See Manual for in detail about every element - located on dev_parallel_tool)

The asynchronous test demonstrated the potential to speed up the compiling process by parallelising it Therefore, we decided to build a parallelisation tool.

Since some solidity files use the “import” statements, they will have some assumptions about the current directory of the OS when they are being compiled. Therefore, in the original design, we would like to use the C++ filesystem library to set the current path for different tasks in different threads. However, we soon discovered that the parallelisation tool would give a non deterministic output, given the above design. A compiling task may fail in one run, but it may succeed in producing binaries in another run. This is due to that all threads from a process share the same current_path. Thus, it is not possible to have a consistent current_path when compiling in parallel. To tackle this issue we decided to keep the current path as the top level directory, and required users to use absolute path/ relative path instead of imposing any assumption on the current directory. For example, “import x” (this would assume x is in the top level directory) should be changed to “import ./x” (this would assume x is in the same directory as the file being compiled, which is desired). It solved the problem.

Another thing we need to tackle is the dependency issue in some solidity libraries. We have found that some solidity files that involved operations like name base inheritance, need to be compiled together to generate the binary files. Therefore, it will break the parallelisation process if we separate these files and assign them to different tasks. We address this issue by asking users to create a subdirectory under the top level directory for each batch of interdependent solidity files. For files that are independent and can be compiled in any order, the users can put them in the top level directory. Our helper function distribute_tasks() will handle the above two cases by checking its boolean argument isRoot when traversing the directories [2].

Three are three major functions that support the tool.

```
/* A bare-bones function to compile solidity source files */

int compile(string compiler, string flags, string files,

std::experimental::filesystem::path current);


/* A helper function to distribute sources to different tasks correctly */

void distribute_tasks(string& path, string& extension,

vector<tuple<std::experimental::filesystem::path, vector<string>>>& pool, bool isRoot);


/* A helper function to evenly distribute independent sources (only exist in the

top-level directory according to our assumption) among CPUs */

void parallel_compile(string& compiler, string& flags, vector<string>& sources,

vector<future<int>> &vec_res);
```

And we call them in main() in the following way:

```
vector<tuple<std::experimental::filesystem::path, vector<string>>> pool;

distribute_tasks(path, extension, pool, true);

for (auto & p : pool) {

        auto[path, sources] = p;

        if (path == root_path)

                parallel_compile(compiler, flags, sources, vec_res);

        else
```

```
                /* ... */
                vec_res.push_back(async(std::launch::async, compile, compiler, flags,
tasks, path));
        }
        for_each(vec_res.begin(), vec_res.end(), [](future<int> &res){ res.get(); });
```

We first call distribute_tasks() to populate the pool vector that stores a tuple of <directory, sol files> for different threads' compiling tasks. Then we iterate over the vector, and if the directory in the tuple is the top level directory, we will call parallel_compile() to evenly distribute them to different CPUs. Otherwise we will assume all files in the directory need to be compiled together and we will assign a new thread that compiles all of them.

The parallelisation tool takes 4 arguments, which are path to the solidity compiler executable, flags for the compiler, top level directory for sources and extension for solidity files. Once the parallelised task finishes, the binaries generated will be in the "bin" directories in top level directories or each subdirectory [2].

6. Future Work/Version 1.2:

- Utilize topological sort algorithm to develop parallelisation tool further, searching for imports within files to build a sorted dependence tree structure of sorts, giving less constraints than assuming pure independence
- Report more types of statistics:
    - Compiler memory usage
    - Space on the blockchain
- Use tool to investigate the rest of the codebase
- Modify tool to support concurrency
- Extending parallelization to other single-threaded compilers

7. References

[1] "ethereum/solidity - GitHub." https://github.com/ethereum/solidity. Accessed 26 Apr. 2018.
[2] "raphael-s-norwitz (Raphael Norwitz) · GitHub." https://github.com/raphael-s-norwitz. Accessed 26 Apr. 2018.
[3] "gcc/timevar.c at master · gcc-mirror/gcc · GitHub." https://github.com/gcc-mirror/gcc/blob/master/gcc/timevar.c. Accessed 26 Apr. 2018.
[4] "c++ - Are system() calls evil? - Stack Overflow." https://stackoverflow.com/questions/8086071/are-system-calls-evil/8086130. Accessed 26 Apr. 2018.

[5] "Solidity by Example — Solidity 0.4.21 documentation - Read the Docs." http://solidity.readthedocs.io/en/v0.4.21/solidity-by-example.html. Accessed 26 Apr. 2018.
[6] "GitHub - allenday/github-solidity-all: All *.sol files from github in a ...." https://github.com/allenday/github-solidity-all. Accessed 26 Apr. 2018.

All code taken from our github repository: github.com/raphael-s-norwitz/solidity