QuickerCheck

Implementing and Evaluating a Parallel Run-Time for QuickCheck

Robert Krook
krookr@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Nicholas Smallbone
nicsma@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Bo Joel Svensson
bo.joel.svensson@gmail.com
Lind Art & Technology
Stockholm, Sweden

Koen Claessen
koen@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

This paper introduces a new parallel run-time for *QuickCheck*, a Haskell library and EDSL for specifying and randomly testing properties of programs. The new run-time can run multiple *tests* for a single property in parallel, using the available cores. Moreover, if a counterexample is found, the run-time can also *shrink* the test case in parallel, implementing a parallel search for a locally minimal counterexample.

Our experimental results show a 3–9× speed-up for testing QuickCheck properties on a variety of heavy-weight benchmark problems. We also evaluate two different shrinking strategies; *deterministic shrinking*, which guarantees to produce the same minimal test case as standard sequential shrinking, and *greedy shrinking*, which does not have this guarantee but still produces a locally minimal test case, and is faster in practice.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent algorithms**; **Shared memory algorithms**; • **Theory of computation** → **Shared memory algorithms**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

property-based testing, quickcheck, testing, parallel functional programming, haskell

ACM Reference Format:
Robert Krook, Nicholas Smallbone, Bo Joel Svensson, and Koen Claessen. 2023. QuickerCheck: Implementing and Evaluating a Parallel Run-Time for QuickCheck. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652561.3652570

IFL 2023, August 29–31, 2023, Braga, Portugal

ACM ISBN 979-8-4007-1631-7/23/08.
https://doi.org/10.1145/3652561.3652570

1 INTRODUCTION

QuickCheck [5] is a widely known Haskell tool for property-based random testing of programs. First, the programmer writes a property of the program under test that they expect to always hold. Then, to check the property, QuickCheck generates a number of random test cases to exercise the property. If the property always held, the check is reported as successful. If a test case makes the property fail, a process called *shrinking* is invoked, which consists of a greedy search for a (locally) minimal failing test case.

For example, we may be testing an implementation of System F [6], where we have the following types and functions:

```
type Expr -- expressions
type Type -- types

reduce :: Expr -> Maybe Expr
typeOf :: Expr -> Type
```

The types `Expr` and `Type` stand for expressions and types of expressions in System F. The function `reduce` takes one evaluation step, if possible. The function `typeOf` computes the type of an expression. *Subject reduction* is a property that says that evaluation of expressions does not cause their type to change. This can be expressed as a QuickCheck property as follows:

```
prop_Preservation :: Expr -> Property
prop_Preservation e =
  isJust r ==> typeOf e == typeOf (fromJust r)
 where
  r = reduce e
```

Here, the operator `==>` specifies a *precondition*: only tests satisfying `isJust r` are of interest.

To run QuickCheck, the user must also supply an `Arbitrary` instance describing how to generate random *well-typed* `Exprs`[1] (a non-trivial task studied in e.g. [7]). QuickCheck will then generate a configurable amount of random expressions, which by default is 100, and evaluate the property for them. In fact, QuickCheck will typically evaluate the property even more times, because:

- QuickCheck discards any test case not satisfying the precondition `isJust r`, and continues until it has executed 100 tests satisfying the precondition.

[1]There is no requirement by QuickCheck itself that the generator has to generate well-formed terms. This is primarily required to meaningfully exercise the property in question.

- If a test case fails (for example if the function `reduce` contains a bug), shrinking searches for a smaller counterexample by executing the property on many smaller test cases.

All this happens *sequentially* at the moment in QuickCheck. If the evaluation of the property takes a long time, QuickChecking it (and possibly shrinking the counterexample) will take an even longer time. This is time often spent waiting by the programmer, perhaps wondering why their computer is roaring like a spaceship while only one core is in use. The contribution of this paper is to propose and practically evaluate a way of performing both the testing phase as well as the shrinking phase of QuickCheck in parallel[2].

Note that our work aims to reduce waiting time for the programmer while checking a *single property*. There exist frameworks (for example tasty [4]) that allow testing of multiple properties and unit tests in parallel or even distributed on a cluster. These are typically used in regression tests or continuous integration. Our work can not only speed up testing in these settings but also during active development, where programmers typically run QuickCheck on a single property and wait for the result.

2 WHAT ARE THE CHALLENGES?

Even though running each test is supposed to be independent, and as such testing a property 100 times should be a so-called *embarrassingly parallel* task, in practice parallelizing testing is not so easy. For one, individual tests may interact with each other, but luckily in Haskell, we often get the independence guarantees we need from pure (or at least thread-safe) code. In this paper, we assume that the property itself is thread-safe.

But the biggest problem is that *QuickCheck's algorithm is inherently sequential*. This is not at all obvious at first glance. The problem comes from two features in QuickCheck – adjustment of test size, and shrinking. As we will see, these features introduce a data dependency: the test case that we should try next depends on the result of the *previous* test. Addressing these dependencies was one of the main challenges in parallelizing QuickCheck.

Test size. Many times it is enough to generate a small test input to falsify a property. QuickCheck tries to generate smaller inputs early on, and gradually increases the size as more and more tests are passed. This is achieved by QuickCheck supplying the test-data generator with a *size*. The generators are free to disregard the size completely but may use it if they wish to. As an example, the default generator for lists uses the size as an upper bound of the length of the generated list. The size of the first test is always 0, while the default upper bound is 100. If the user specifies that 100 tests should be executed, QuickCheck will make sure that the generator has been provided with all sizes between 0 and 99. The authors point out that so far everything discussed is easily parallelisable.

However, properties in QuickCheck can not only succeed or fail, but also *discard*, which means that a pre-condition in the property was not fulfilled. A discarded test case is not counted towards the total number of successful tests[3]. So, the distribution of sizes during testing only depends on the total number of *successful* tests so far, not on the total number of tests in general. This introduces a small but significant data dependency preventing parallel evaluations of tests; when a worker runs a test it must know the appropriate size of the test case to run, and the appropriate size depends on whether the previous tests were successful or not. This dependency needs to be dealt with somehow in the parallelization.

Shrinking. When a failing test case is found, QuickCheck searches for a smaller failing test case by applying a process called *shrinking* [10]. The goal of shrinking is to produce a locally minimal failing test case. Shrinking first produces a list of *shrink candidates*, variants of the test case that have been reduced in size in a variety of ways. This list is traversed from left to right until we find a new failing test case. Shrinking is then applied recursively on the new failing test case until the current failing test case cannot be reduced anymore. Shrinking does not backtrack in search of a globally minimal counterexample, but only promises to yield a local minimum.

To use shrinking, the user must define a *shrink function*. For a type `T`, this is a function `shrink :: T -> [T]` which, given a test case, produces a list of shrink candidates, i.e. smaller or simpler test cases to try. For example, suppose that we are testing System F again, and the type of expressions is defined as follows:

```
data Expr
  = Var String      -- variable
  | App Expr Expr   -- application
  | ...             -- other constructors
```

Then we can define a shrink function as follows:

```
shrink :: Expr -> [Expr]
shrink (Var _) = []
shrink (App t u) =
  concat [ [ t, u ]
         , [ App t' u | t' <- shrink t ]
         , [ App t u' | u' <- shrink u ]
         ]
shrink ( ... ) = ...  -- other constructors
```

A `Var` can not be shrunk further, so we return an empty list of candidates. A function application `App t u`, however, can be shrunk further. We can remove the `App` constructor and one of the subexpressions, leaving just `t` or `u`, which if successful may shrink the expression considerably. We can also keep the `App` constructor but shrink the subexpressions.

Note that QuickCheck *always tries the shrink candidates in the order they appear in the list, from left to right.* Hence it is common to return the *greedy* candidates first, those that remove large parts of the value, as we do in this case. Ordering the shrink list appropriately can greatly improve the speed of shrinking.

We propose to parallelize shrinking in two ways:

(1) *Greedy shrinking* evaluates as many shrinking candidates in parallel as possible, and as soon as a candidate fails, it recursively continues with that candidate. It may be that a

[2]The implementation can be found at https://github.com/Rewbert/quickcheck. The authors intend to eventually merge this work into mainline QuickCheck.

[3]The reason for this is that if the precondition of a property is more likely to succeed for small test data sizes, we still want to make sure that we exercise the property on larger sizes.

candidate *earlier* in the shrink list (corresponding to a more aggressive shrink step) would also have failed if we had waited, and in that case, we may perform a smaller shrink step than necessary.

(2) *Deterministic shrinking* speculatively evaluates test cases in the search before we know we will need to, but always makes the same choices as in the sequential case. That is, when a shrink candidate fails, it waits until it knows that *no earlier candidate fails*

In our evaluation, greedy shrinking is usually faster than deterministic shrinking.

3 QUICKERCHECK

We present QuickerCheck via two examples. We point out that the QuickCheck API for writing generators, shrinkers, and properties remains unchanged, and only the internal evaluation of a property is modified.

System F. In Section 1, we saw the property `prop_Preservation :: Expr -> Property` for testing subject reduction in System F. To test this property with *sequential* QuickCheck we run:

```
> quickCheck prop_Preservation
+++ OK! Passed 100 tests.
```

As the property is pure, it is safe to test in parallel using QuickerCheck. To do so, we must compile the code with the `-threaded` and `-rtsopts` flags and pass in the `-N` option to the run-time system, to enable parallelism in GHC. Then all we have to do is invoke `quickCheckPar` instead of `quickCheck`.

The output (assuming all tests passed) is

```
> quickCheckPar prop_Preservation
+++ OK! Passed 100 tests.
  tester 0: 50
  tester 1: 50
```

The lines `tester 0: 50` and `tester 1: 50` show that two threads were used (we happened to limit GHC to using two cores) and that they each executed 50 test cases. What is not visible in the output is that, since the tests were distributed among two cores, QuickerCheck ran close to twice as fast.

Compiler testing. A function that is not necessarily embarrassingly parallel is one that is effectful. To test a compiler it is necessary to perform IO actions, such as invoking the compiler under test or executing the compiled binary. Testing compilers is non-trivial, but a well-studied approach is *metamorphic testing* [3]. In this approach, assuming a function of type `Program -> IO Output` that compiles and runs the program, we define a function `mutateProgram :: Program -> Program` that mutates the program in some way, and then specify how the output should change in response by a function `mutateOutput :: Output -> Output`. Mathematically, the property that should hold:

```
-- compileAndRun :: Program -> IO Output
compileAndRun (mutateProgram p) =
  fmap mutateOutput (compileAndRun p)
```

In practice, we also need to perform various housekeeping tasks such as writing the program source to a file and cleaning up output files, so a more realistic property is:

```
prop_metamorphic :: Program -> Property
prop_metamorphic program = ioProperty $ do
  writeFile "p.c" (render program)
  writeFile "q.c" (render $ mutateInput program)
  output1 <- compileAndRun "p.c"
  output2 <- compileAndRun "q.c"
  mapM removeFile ["p.c", "q.c", "p.exe", "q.exe"]
  return (mutateOutput output1 == output2)
```

The property executes both the original and modified programs after having first written them to the file system. The file system is cleaned up, after which the outputs are compared. The output of the unmodified program is modified to reflect the change described by the metamorphic relation.

Unfortunately, running this property with `quickCheckPar` will produce *extremely strange* test failures. The reason is that the property, while innocent-looking, is not thread-safe. There is an implicitly shared resource, the file system: if multiple instances of the property execute in parallel, they will all write to the *same* files `p.c` and `q.c`. This leads to obvious race conditions. There are different ways to modify the property such that there are no race conditions, one of which is to let the property create a temporary directory to which intermediary files are written.

```
-- create a fresh temporary directory based on a baseline name
-- withSystemTempDirectory :: String -> (FilePath -> IO a) -> IO a

prop_metamorphic :: Program -> Property
prop_metamorphic program = ioProperty $ do
  withSystemTempDirectory "compiler_output" $ \dir -> do
    -- rest of property, now using dir as a
    -- scratch space for temporary files
```

If we disregard other implicitly shared resources such as CPU caches, RAM, bandwidth, etc, this property can now be evaluated in parallel by using `quickCheckPar`.

In general, using QuickerCheck requires three steps. (1) Make sure that the property is thread-safe (only for properties doing I/O). (2) Compile the program with threading options. (3) Run `quickCheckPar` instead of `quickCheck`.

4 QUICKERCHECK DESIGN AND IMPLEMENTATION

The extensions to QuickCheck described in this paper are designed such that as few observable behaviors as possible are changed. Some design choices of QuickCheck do not lend themselves nicely to parallelism, and QuickerCheck tries to make reasonable compromises where possible. One notable case of this is the way QuickCheck computes sizes for a test case. The size is derived from the number of tests that have passed so far, and the number of tests that have been discarded since the last passing test. This means that we can not compute the size of a test until we have observed the outcome of all tests that came before. This sounds sub-optimal for parallelization; below, we explain what QuickerCheck does to address this.

Testing. The test loop in ordinary, sequential QuickCheck is a recursive function that maintains a state containing e.g. the count of how many tests were executed so far, how many were discarded

due to a failed pre-condition, etc. It also holds the random seed used to generate the test case. It generates and executes one test at a time, adjusting the size of the test case whenever a test succeeds, but not when a test is discarded. Once a test fails, the test loop terminates and a shrinking routine is invoked.

The parallel test loop is implemented by running concurrent instances of the sequential test loop. The main thread spawns concurrent *testers* that evaluate one test after another, and then goes to sleep until the testers report that all tests have been executed, too many tests were discarded, or a counterexample was found. The test loop maintains a state that is updated after every test, recording how many tests have been passed so far, the next random seed, and many other things. In order to facilitate multiple, concurrent testers, some of the state has been moved into `MVars`. As an example, the integer representing the number of tests a particular thread has yet to run resides in an `MVar`, enabling other threads to read it if they wish to steal work from that thread.

Communication between threads occurs as little as possible in order to not incur synchronization costs. When testing is initiated, the number of tests to run is divided equally between the testers, and only when one thread has exhausted its budget of tests will it inspect the budgets of the concurrent testers. If work-stealing is enabled, a thread may then decrement the counter of a sibling tester and run another test on its own. Each tester has its own random seed that it splits before running a test, as sharing a seed between all testers would incur synchronization overheads. Additionally, each tester computes the sizes to use for test cases based on their individual counters for how many tests they have passed, and how many tests they have discarded since the last passing test. In an effort to explore the same set of sizes as in sequential QuickCheck, they each apply a *stride*: If we have k threads, then thread number i uses sizes $i, i+k, i+2k, i+3k, \ldots$. This is illustrated in figure 1. A compromise is made when a thread steals a test from a sibling tester, in which case the local next size is used. This reduces synchronization costs, as the thread that ran the test doesn't need to report the result back to the other thread. With this approach, we explore the same set of sizes as sequential QuickCheck, except when work stealing happens.

As an alternative to strides, we have also implemented a strategy that divides the set of sizes into contiguous segments for each of the testers, by applying an offset to the size computation. There is a risk, however, that test cases generated by e.g. smaller sizes will run faster than test cases generated with larger sizes. This would lead to the concurrent testers finishing their given workloads at different times. Computing sizes with an offset is implemented and can be chosen by configuring the arguments to `quickCheckWith`[4], but the default behavior is to use a stride.

When a thread finds a counterexample it wakes up the main thread by writing the used seed and size to an `MVar`. The main thread will then terminate the remaining testers before it shrinks the counterexample, by delivering asynchronous exceptions. This is very abrupt, with the exceptions delivered at the next allocation point.

[4]The function `quickCheckWith` is a variant of `quickCheck` that accepts a configuration parameter where default behavior can be overridden.

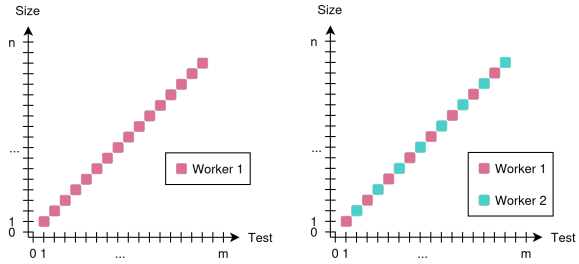


Figure 1: An illustration of how size grows as more and more tests are passed, and to which worker they are assigned. In order to get a fair workload for the concurrent testers a stride is applied when computing sizes.

graceful. When a property is aborted as violently as this, by raising an asynchronous exception, there is a risk that there will be artifacts left from a test. If a property e.g. creates a new file on the file system that is normally deleted at the end, an interruption by an asynchronous exception may make the file erroneously persist.

```
prop :: Input -> Property
prop ip = ioProperty $ do
  run $ writeFile "temp.txt" (show ip)
  -- do some work
  run $ deleteFile "temp.txt" -- we may never execute this
```

To address this, we introduce a combinator `graceful` that takes an `IO` action and a handler. The handler will run if QuickCheck makes the choice to terminate evaluation of the property.

```
-- graceful :: IO a -> IO () -> PropertyM IO a

prop :: Input -> Property
prop ip = ioProperty $ do
  run $ writeFile "temp.txt" (show ip)
  graceful
    (do -- do some work
        deleteFile "temp.txt")
    ( deleteFile "temp.txt")
```

The handler is implemented by intercepting the asynchronous exception before the worker is restarted and running the handler before rethrowing the exception. *graceful* can only capture a specific exception thrown internally by QuickCheck. We choose to implement this dedicated operator like this rather than relying on existing bracket functionality, as both user code and QuickCheck might already have code in place to deal with exceptions.

`graceful` can be used not only for shrinking but also for testing. When one tester finds a counterexample the concurrent testers will be aborted. This combinator will make sure that cleanup occurs then as well.

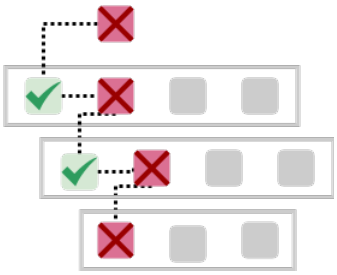
Shrinking. The existing shrink loop continually evaluates the head of the candidate list until a new counterexample is found, at which point the loop recurses, or until the list is empty, at which point shrinking is terminated. This is illustrated in figure 2a. The design of the new loop is very similar.

Rather than a single thread traversing the candidate list one element at a time, the parallel shrink loop spawns concurrent worker threads that cooperate and traverse the same list, now residing in an `MVar`. If any of the concurrent workers finds a new counterexample, they will update the shared list of candidates and signal to their sibling workers that they should stop evaluating their current candidate and instead pick a new one from the new list.

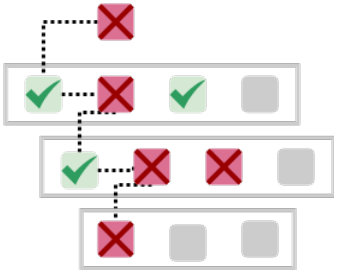
The behavior of this shrink loop might return a non-deterministic result. Whereas the previous loop will always find the first counterexample in the candidate list, the parallel loop might find a counterexample other than the first one. To emulate the deterministic behavior, the new loop can choose to only signal a restart to those concurrent workers that are evaluating candidates that appeared after the current one in the candidate list, and tell them to speculatively start shrinking the new counterexample. The other workers will keep evaluating their current candidates, and if one of them turns out to be a counterexample, the current progress will be discarded, and shrinking will continue with the new counterexample. In this case, we might do some unnecessary work, but we will get the same deterministic result. Figure 2b illustrates this and how this approach may make us evaluate candidates that we don't need.

Another alternative is that when any worker has found a counterexample, all concurrent workers are restarted and told to start shrinking the new counterexample, regardless if this was the first counterexample or not. This might lead to a non-deterministic result, as the path down the rose tree of shrink candidates is not the leftmost one, as illustrated in figure 2c. Restarting a worker is done by raising an asynchronous exception in the worker. The worker will catch this exception and enter the shrink-loop anew, and begin to search through the new list of candidates.

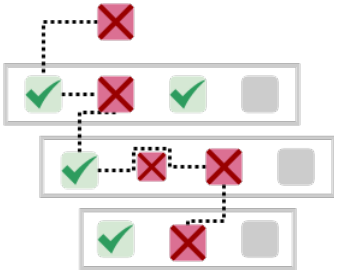
Repeatedly accessing a shared resource may incur overhead costs. If two workers attempt to modify a shared resource at the same time, one will have to wait for the other. As the list of candidate counterexamples is shared between workers, if candidates are evaluated very fast, it is likely that using more threads will slow down shrinking.



(a) Illustration of the existing QuickCheck shrink-loop. It guarantees to always return the same locally minimal counterexample.



(b) The new deterministic shrink-loop promises to find the same local minimum every time, but it may speculatively evaluate other candidates in its search for the final counterexample.



(c) The greedy shrink-loop does not guarantee to find the same local minimum, potentially returning a different final counterexample.

Figure 2: The three figures above illustrate how the search for a minimized counterexample happened. The dotted line represents the final path to the local minimum, green boxes are candidate counterexamples that turned out to not be new counterexamples, and red boxes are counterexamples that still falsified the property. Grey boxes are candidate counterexamples that were never evaluated.

5 EVALUATION

We evaluate QuickerCheck to answer the following four questions

- **Question 1**: Is the sequential performance of the new implementation comparable with QuickCheck?
- **Question 2**: How does the parallel run-time scale as we add more cores?
- **Question 3**: Can we find bugs faster by using more cores?
- **Question 4**: Can we shrink counterexamples faster by using more cores?
- **Question 5**: Does the choice of shrinking algorithm affect the quality of shrunk counterexamples?

To answer these questions we run properties and collect information. We will refer to such properties as benchmarks, and the benchmarks we use are described in the following subsection.

5.1 Benchmarks

We perform all our evaluations using six distinct benchmarks. While the first benchmark *constant* is artificial, the other benchmarks are intended to represent a diverse set of testing tasks. *compiler testing* and *compressid* are effectful tasks making use of IO facilities, while the other tasks are pure.

constant. The benchmark named *constant* is not one that anyone would write organically, but its inclusion as a benchmark in this set has a very specific purpose. The underlying property is

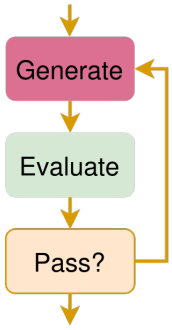


Figure 3: A high-level description of the internal testing loop. The loop begins by generating input and then invoking the property. After this, the loop inspects the outcome before it either reports having found a counterexample, or loops back to repeat all steps. The bottom box and all arrows are part of the internal testing loop, while the top two boxes are defined by the user.

```
1 prop_constant :: () -> Bool
2 prop_constant () = True
```

The cost in execution time of running a test consists of three parts – generating input, running the property, and the machinery of the internal testing loop. This is illustrated in figure 3. As QuickerCheck only changes the workings of the testing loop, we want to measure the change in cost of just the testing loop. The above property minimizes the execution time of both the generation of test data and evaluation of the property. Generation and evaluation are constant time as there are no random choices to make during generation and evaluation of the property is trivial. Measured changes in the execution speed of QuickCheck vs QuickerCheck on this benchmark should primarily be a result of the different testing loops.

compiler testing. The underlying property of the *compiler testing* benchmark asserts that a compiler for an imperative language generates correct output. The property is stated as a metamorphic relation as described in section 3.

In practice, the property does significantly more work than the other benchmarks. It generates a type-correct imperative program and produces several executables that are invoked to assert the correctness. The generated programs may include non-terminating loops, so the property might require some time to execute. Such loops are eventually broken by the property itself after consuming too many resources. During evaluation, the property will spend a significant amount of time in external processes.

compressid. This benchmark composes the two Unix commands *gzip* and *gunzip* and verifies that the composition behaves as the identity function. It generates an arbitrary string and invokes `gzip`, passes the compressed result to `gunzip`, and asserts that the final output is identical to the input.

This benchmark comes in three flavors – one is a naive implementation (*naive*) that writes intermediary values directly to the file system. Since the file system is a shared resource a property like this will experience race conditions if multiple threads are used. We test two alternative implementations that make the property thread-safe in different ways. The first (*tmpfs*) generates fresh directories for each concurrent worker to write such files to, and the second (*nofs*) uses pipes to pass values around, never using the file system.

verse. This property asserts the confluence of the rewrite system for the Verse Core Calculus [2]. A rewrite system is confluent if, regardless of which rewrite rules are applied in each step, the result is always the same, single, normal form.

The property generates an arbitrary term and applies two arbitrary sequences of rewrite rules. If the two resulting normal forms are different, the rewrite system is not confluent and the property is falsified.

system f. The *system f* benchmark is a pure property that generates arbitrary lambda terms and asserts the subject reduction property, described in section 1, which states that the type of a term should not change after performing one reduction of said term. The code was taken from Etna, an evaluation platform for Property-based testing frameworks[8].

twee. Twee [9] is a high-performance theorem prover for equational logic written in Haskell. A key component is the *term index*, a data structure for finding equations matching a given term. The *twee* benchmark is a pure property stating that, after any sequence of update operations on a term index, the data structure's invariant is preserved.

5.2 Results and Discussion

Evaluation is done using an Intel I7-10700 8-core CPU with turboboost turned off. The evaluation system is equipped with 64GB of 2933MT/s RAM.

We use GHC to compile and execute Haskell code, using the compile-time flags `-threaded`, `-feager-blackholing`, and `-rtsopts`. We don't try to mitigate garbage collection costs by increasing the nursery size or try to improve the performance in any other way, as we believe most people use QuickCheck without doing this. All invocations of QuickCheck are made with the `chatty` flag set to `False` as printing would otherwise affect the results. In appendix A it is illustrated how `chatty` affects experimentation.

Is the sequential performance of the new implementation comparable with QuickCheck? We answer this by executing each benchmark several times both with QuickCheck and QuickerCheck, using only one core. We compute the median execution times and compare them. The results are presented in figure 4.

Something that immediately stands out is the huge overhead experienced by the *constant* benchmark. This benchmark is intended to act as a worst-case property and illustrate precisely what the overhead of the new testing loop is. The results indicate that, in the worst case, QuickerCheck will incur a penalty of 70%.

The other benchmarks all perform some actual workload and experience much more modest changes in performance. The *system f* property, just like the *constant* property, is very fast. By running many more tests it interacts much more with the new testing loop, incurring more of the new costs. This shows up by QuickerCheck

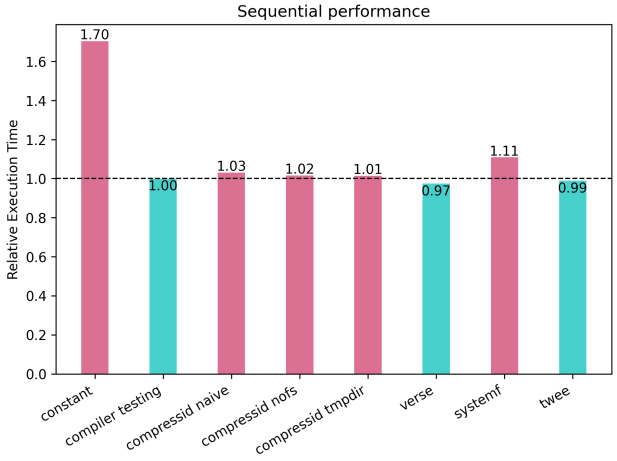


Figure 4: The performance of sequential QuickerCheck compared to that of QuickCheck. A number of 1 means that there was no difference in performance, whereas a number less than 1 indicates that QuickerCheck was faster than QuickCheck (e.g. 0.5 shows that QuickerCheck finished in half the time). A number greater than 1 indicates that QuickerCheck was slower than QuickCheck.

requiring 11% more execution time to finish the same workload. Some of the workloads experienced no change at all or even got slightly faster.

Not accounting for the *constant* benchmark, it appears that there is no major change in performance by using sequential QuickerCheck instead of QuickCheck.

How does the parallel run-time scale as we add more cores? Each of the benchmarks is executed several times for each core configuration, the median execution time is computed and the speedup relative to the sequential running time is computed. The results are presented in figure 5.

We first observe that many of the benchmarks scale very well until the point where we exhaust the number of physical cores. The highest speedup was achieved by the *compiler testing* benchmark which got more than eight times faster. The *verse* property is not far behind.

The *twee* benchmark initially scales very well but starts to lose momentum when we approach the limit of physical resources. When hyper-threading is active performance slowly but surely degrades. The *twee* benchmark is very data-intensive and frequently moves data around. While two hyper-threads appear to the operating system as two CPUs, they are actually two logical threads that share hardware components required to execute machine instructions, such as caches and the system bus. One potential explanation for this degradation is that the different testers affect the cache in unfavorable ways.

One noticeable difference between e.g. the *compiler testing* and *system f* benchmark is that the *compiler testing* property is significantly slower. The property may spend over a second evaluating a single test while the *system f* benchmark may run thousands of

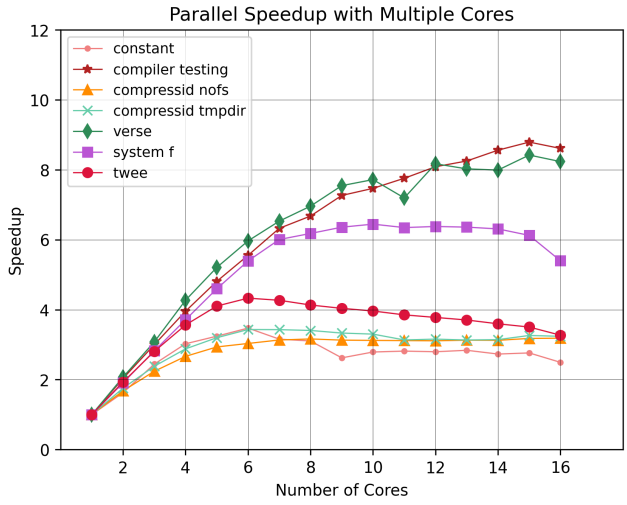


Figure 5: The acquired speedup relative to the sequential execution time when running tests.

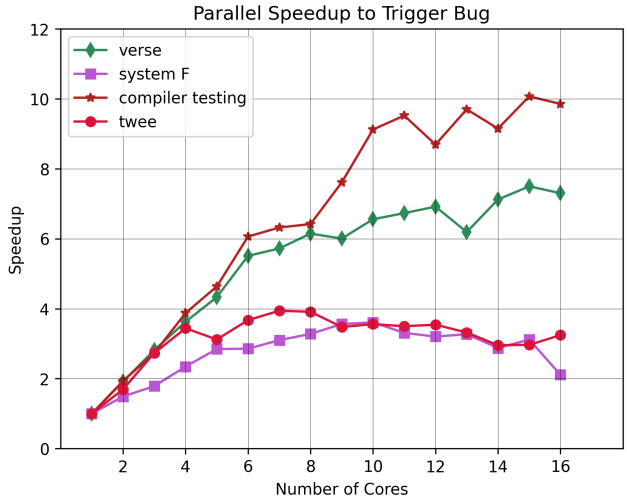


Figure 6: The acquired speedup relative to the sequential execution time when searching for a planted bug.

tests in the same time frame. The results seem to indicate that the more time a property spends inside the body of the property, the greater the potential speedup.

Can we find bugs faster by using more cores? To evaluate this we plant a bug in 4 of the 6 benchmarks and let QuickerCheck run until it finds the bug. This is repeated 300 times after which the median execution time is computed. Figure 6 illustrates the speedup acquired relative to the sequential execution time.

We first note that the *system f* benchmark doesn't reach as high of a speedup as when we are running tests without a bug enabled. While we can't say with certainty what to attribute this to, we have a pretty good guess of what is happening. The shape of the

curve is the same, except that it is pushed down towards lower multiples. There are some new costs associated with starting up the parallel test loop, and when we run tests without a bug enabled the benchmark is allowed to run for a few seconds, running hundreds of thousands of tests. The cost of starting up the test loop is amortized over all these tests, while when a bug is enabled there are many fewer tests. The bug was found after roughly 200 tests, running for just a couple of milliseconds.

The overall shape of the *twee* benchmark is the same, but not reaching quite as high of a speedup as when just running tests. The *compiler testing* benchmark acquires a 10x increase in performance, outperforming all other evaluated benchmarks. We believe this speedup is higher than that achieved in figure 5 because many concurrently running tests are aborted when a counterexample is found. When evaluating speedup for question two, every test that we began evaluating was expected to finish, whereas when we evaluated question three, we terminated concurrent testers when one of them found a counterexample. We will thus do slightly less work. We observe the inverse behavior in the *system f* property, where the concurrent testers have time to run many additional tests before they are terminated by a tester who found a counterexample.

Can we shrink counterexamples faster by using more cores? We generate 200 random seeds that we know trigger bugs, such that we can replay them to deterministically see the same counterexamples. We replay the seeds and measure how long it takes to shrink them, varying the number of cores and the choice of strategy (deterministic or greedy shrinking). We have done this for the three benchmarks *compiler testing*, *twee*, and *verse*. Because it is impractical to show all the results, we have picked some subsets of data that we find representative of the overall results.

The *compiler testing* results are presented in figure 7, 8, and 9. Figures 7 and 8 illustrate the relationship between sequential and parallel execution time, using two cores. The red dots got slower when two cores were used, whereas the blue dots achieved a speedup. The further from the line a point lies, the more extreme the achieved effect is. From the two figures, we can see that the greedy algorithm appears to benefit more experiments and that the achieved effects are greater. The blue dots in figure 7 appear to tangent a line. This line traces the execution time that is twice as fast as the sequential one and illustrates the upper bound defined by Amdahl's law[1]. The results in figure 8 show some experiments crossing this boundary, which is explained by the greedy algorithm being able to return a completely different counterexample.

As more and more cores are added, the results indicate that more and more experiments got slower, while the remaining ones that achieved a speedup achieved a much greater speedup.

To try and answer which counterexamples may benefit from parallel shrinking, we plot the *efficiency* of the shrunk counterexamples. The efficiency of a single counterexample is defined as the fraction of evaluated candidates that successfully shrunk the counterexample, and as such is a number between 0 and 1. It is clear that if the efficiency is one, there is nothing to be gained from parallelism as shrinking becomes a sequential search. As an example, the total number of evaluated candidates in figures 2a, 2b, and 2c are 5, 7, and 8 respectively. In all 3 cases the number of successful shrinks was 3, so the efficiencies are 0.6, 0.42, and 0.375.

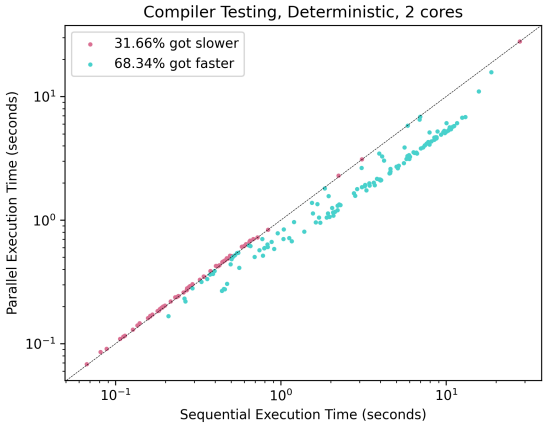


Figure 7: The speedups acquired when using two cores to shrink the *compiler testing* tests, using the deterministic algorithm.

Figure 8: The speedups acquired when using two cores to shrink the *compiler testing* tests, using the greedy algorithm.

Figure 9 shows that there is a clear trend of counterexamples with a good efficiency not benefiting from parallel shrinking. If there was not that much extra work to be done from the beginning, the existence of more cores does not offer any substantial performance improvements.

The results observed from *twee* (figures 10, 11, and 12) tell a different story. The *twee* property finishes shrinking in a couple of milliseconds, and using more cores quickly makes all observed counterexamples shrink slower. The efficiency appears to make no difference and we believe that the overhead of the parallel search overshadows any benefits of using more cores. The advantage of having more cores at one's disposal appears to mainly be beneficial in cases where execution will require a non-trivial amount of time.

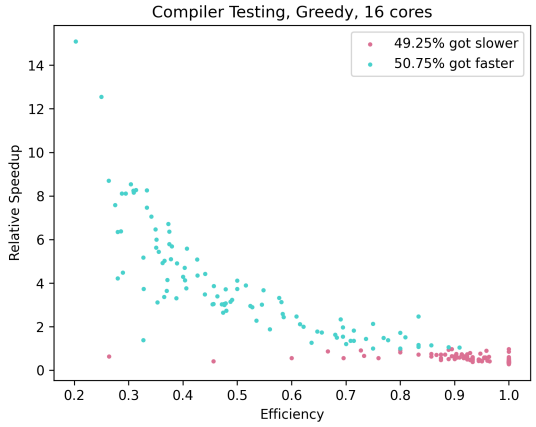


Figure 9: This figure illustrates that the closer the efficiency is to one, the higher the probability that the test will get slower when shrinking. It also appears that the relative speedup is higher the lower the efficiency.

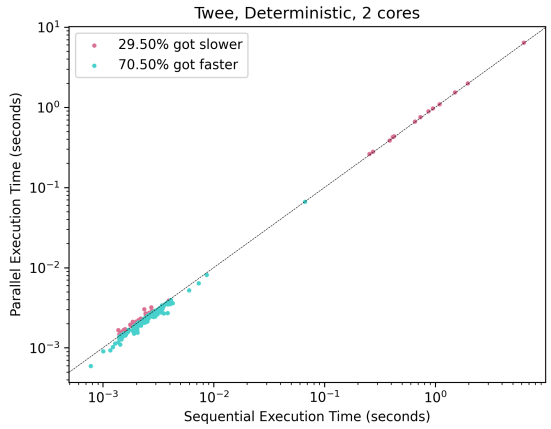


Figure 10: The results indicate that most tests finished shrinking very fast when only two cores was used.

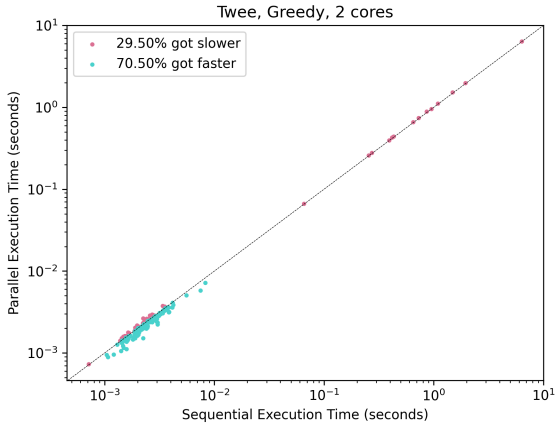


Figure 11: The greedy algorithm appears to perform roughly the same as the deterministic one, with the exception of some tests that did indeed shrink faster.

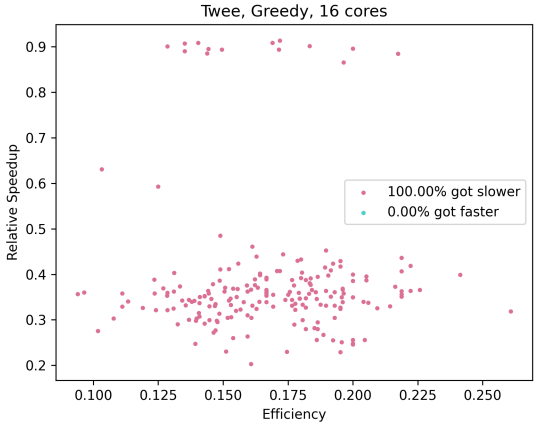


Figure 12: While the efficiency turned out to be an excellent indicator for whether a test got faster or not for the *compiler testing* benchmark, the same can not be said for *twee*. Using 16 cores and the greedy algorithm, all tests got slower and there was quite a spread of efficiencies. The overall efficiency appears to be much lower, but there is still nothing to be gained by additional cores.

The *verse* benchmark, much like the *compiler testing* one, achieves a noticeable speedup for the majority of candidates. This is illustrated in figures 13 and 14. The efficiency of the shrinker is depicted in figure 15, and shows that there is a slight trend towards candidates with a lower efficiency being more likely to benefit from multiple cores. This benchmark shrinks quite rapidly, and as we add more cores, more and more candidates become slower, with the final number at 16 cores showing that roughly half of the candidates experienced a slowdown.

Does the choice of shrinking algorithm affect the quality of shrunk counterexamples? The deterministic shrinking algorithm will always yield the same locally minimal counterexample, while the greedy algorithm may return another local minimum, of a potentially different size. We are interested in finding out whether the distribution of sizes of shrunk counterexamples is different for the two algorithms. We evaluate this on two benchmarks, *compiler testing* and *verse*. We collect 300 seeds from the *compiler testing* benchmark and 500 from the *verse* benchmark. These seeds immediately falsify the property, allowing us to shrink them and record the size using both algorithms. We define the size of a counterexample as the number of constructors in it. We point out that both algorithms produce identical results when only one core is used.

To compare the results from the two algorithms, we model the measured sizes as negative binomial distributions. Whereas we

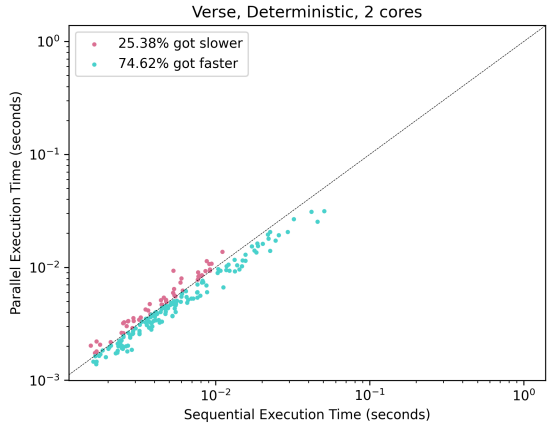


Figure 13: The speedups acquired with two cores using the deterministic algorithm, for the *verse* benchmark.

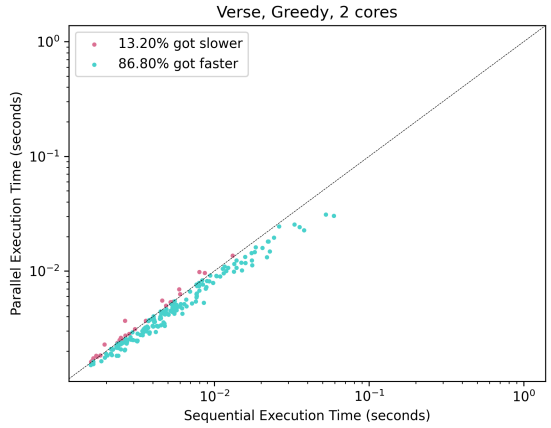


Figure 14: The speedups acquired with two cores using the greedy algorithm, for the *verse* benchmark. It can be observed that the number of candidates that achieved a speedup increased, compared to using the deterministic algorithm.

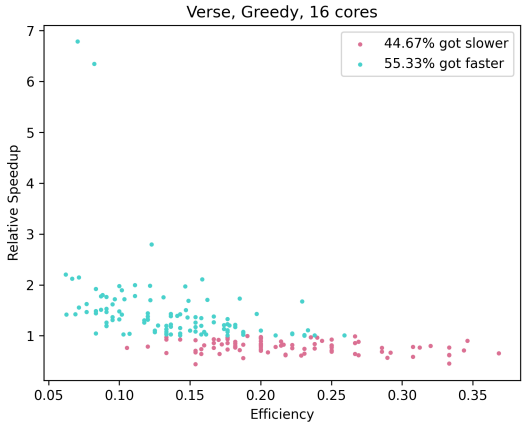


Figure 15: The efficiency of the *verse* shrinker shows that there is a slight trend of lower efficiency indicating that there is a speedup to have by using more cores.

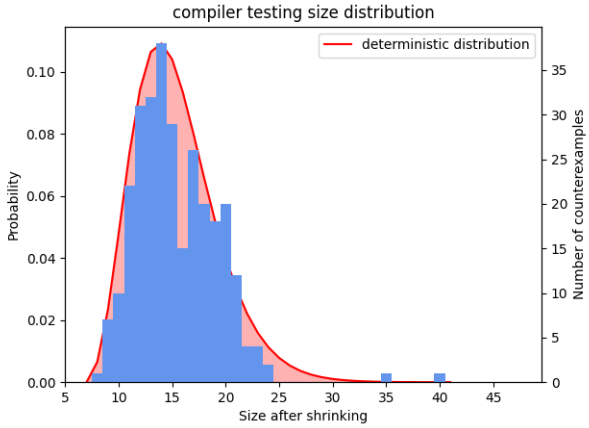


Figure 16: The measured sizes are rendered as a histogram, together with a model that represents the distribution from which they were drawn.

only have one baseline model (the deterministic algorithm), we have 16 models representing the greedy algorithm (one for each core configuration). The authors note that in the single-core case, the two algorithms are identical. Figure 16 illustrates both the measured sizes of the deterministic algorithm, as well as the model representing them.

We compare the models representing the greedy algorithm to the baseline model by computing the entropy between them. Figures 17 and 18 illustrate the baseline model and the greedy model with the highest relative entropy. In both measured benchmarks the difference is very small. While the *verse* benchmark shows little to no difference at all, the *compiler testing* benchmark has a small but noticeable difference. This difference is not large enough to say whether the distributions are different or not. The results indicate that the choice of algorithm does not impact the quality of shrunk counterexamples at all.

6 RELATED WORK

QuickCheck, having proven itself an extremely useful framework for testing software, has been re-implemented in many programming languages. It appears that most other implementations don't support parallel execution of properties. The only other implementation we could find that supports parallelism is *fsCheck*, a QuickCheck implementation for testing *.NET* code. The parallel run-time is not described in any paper and the documentation is sparse, but the implementation is discussed in a merge request introducing the work. The discussion indicates that they initially

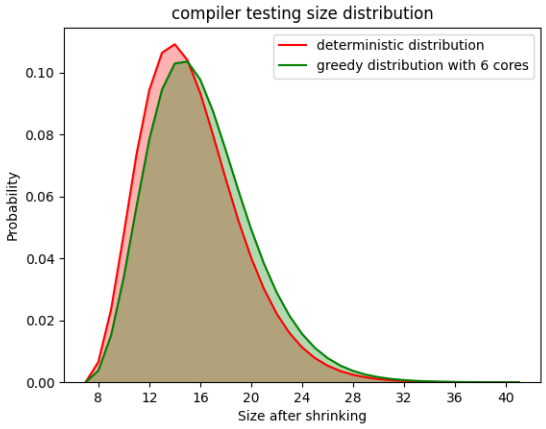


Figure 17: The figure illustrates the distribution of the baseline samples, as well as the greedy distribution with the highest relative entropy, from the *compiler testing* benchmark.

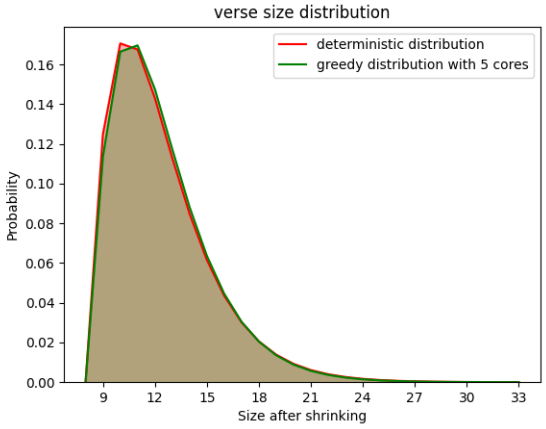


Figure 18: The figure illustrates the distribution of the baseline samples, as well as the greedy distribution with the highest relative entropy, from the *verse* benchmark. The distributions are practically the same.

used an offset to compute sizes for tests but switched to using a stride after observing an uneven workload between workers.

The largest framework for property-based testing by number of users is the Python package *Hypothesis*. They have explicitly chosen not to provide support for parallel evaluation of properties as it can not be determined beforehand whether the function being tested is thread-safe or not. In a non-pure language like Python, this might be a concern, but we believe that this concern is not as severe when it comes to Haskell code. Haskell code is usually split up into its pure parts and effectful parts, with the pure parts being embarrassingly parallel from the get-go. Effectful code can in many cases be refactored to be thread-safe, such that parallel testing may yield positive results.

The Haskell package *tasty* [4] lets the user define test suites with individual tests in a suite being of different kinds. A test suite can simultaneously include e.g. QuickCheck tests, SmallCheck tests, and unit tests. This is possible by *tasty* using different test drivers to execute the tests. *tasty* can execute individual tests in a test suite in parallel, but it will not introduce parallelism in the underlying test drivers. If a test suite contains many tests, with all but one test terminating very quickly, the majority of execution time will be sequential, waiting for the longest running test to terminate.

7 CONCLUSIONS AND FUTURE WORK

Our results show that parallel testing is beneficial. If the property being tested is slow to run the expected performance increase is high, whereas a fast property stands to gain less (but not nothing).

Thanks to the natural division of effectful and pure code in Haskell, many properties are immediately able to benefit from the parallel run-time. We found that with slight modifications to effectful properties, we could run them in a thread-safe manner.

Parallel shrinking is not as universally beneficial, but can still yield good results. For all benchmarks evaluated, individual counterexamples could go either way, either experiencing a slowdown or a speedup. We can not conclude that parallel shrinking is always beneficial. It depends on not only the property but also the specific test case. As more cores are added, some counterexamples will get significantly faster, while the likelihood of your counterexample shrinking slower increases. There seems to be a good compromise around using multiple cores, but a lower number. The greedy algorithm appears to offer a greater speedup than the deterministic one, without compromising on the quality of the counterexamples.

While the work presented in this paper represents a considerable engineering effort, there are still many lines of future work to pursue. While implementing the work described in this paper, it became clear that the ad-hoc way of computing sizes in QuickCheck does not lend itself nicely to parallelism. It imposes a sequential ordering to test cases and is tricky to distribute over multiple cores. While we have implemented a best-effort attempt to maintain the previous behavior, it is not a perfect imitation. The authors would like to implement and evaluate several different ways of computing sizes and reach some conclusions about which strategies are most efficient.

While the greedy algorithm is allowed to search for the fastest path to a counterexample, there may well be more efficient algorithms still. There is still a bias towards finding earlier paths. It would be interesting to see how a random walk would perform.

Currently, the user must explicitly request parallel QuickCheck by using `quickCheckPar` instead of `quickCheck`. This choice was made because properties involving I/O can not always be safely executed in parallel. It would be possible to instead have QuickCheck automatically execute tests in parallel when it is safe to do so. For example, pure properties (not using *ioProperty*) can always be parallelized. Properties doing I/O could be marked as thread-safe using a special combinator.

We are also working together with the QuickCheck maintainers towards merging this line of work into mainline QuickCheck.

REFERENCES

[1] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA (AFIPS Conference Proceedings, Vol. 30)*. AFIPS / ACM / Thomson Book Company, Washington D.C., New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[2] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 203 (aug 2023), 31 pages. https://doi.org/10.1145/3607845

[3] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543

[4] Roman Cheplyaka. 2013. *tasty*. https://hackage.haskell.org/package/tasty

[5] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[6] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph. D. Dissertation.

[7] Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 91–97.

[8] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 878–894.

[9] Nicholas Smallbone. 2021. Twee: An Equational Theorem Prover. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, New York, NY, USA, 602–613. https://doi.org/10.1007/978-3-030-79876-5_35

[10] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

A THE EFFECT OF CHATTY

The `chatty` flag in QuickCheck controls whether QuickCheck should continuously print what it is doing or not. While printing is helpful in assessing the current progress, it can be a bottleneck when it comes to performance.

QuickCheck prints the current progress before every test. If you run just a few tests every second this is of no concern, but if your property is a very fast one it has a huge effect on performance. Running 10000 tests per second means that you will print 10000 times per second, which is significantly more than a human eye can observe.

QuickerCheck takes a different approach to printing. Since the new run-time is multi-threaded anyway, QuickerCheck will spawn a separate worker thread whose sole purpose is to periodically print the progress to the terminal. The duration of the period can be configured, and the default is 200 milliseconds.

While the property that now runs 10000 tests in one second would previously have printed 10000 times, QuickerCheck would only have printed 5 times. In figure 19 it can be observed how much of an effect this has on a sequential workload. The *constant* and *system f* properties run extremely fast, and we observe that with the `chatty` flag set to `True`, QuickerCheck is significantly faster. The *constant* property finished evaluating in one-fifth of the time that QuickCheck required.

As we add cores, it appears that `chatty` might make the benchmarks scale slightly worse, but not a lot, as indicated in figure 20.

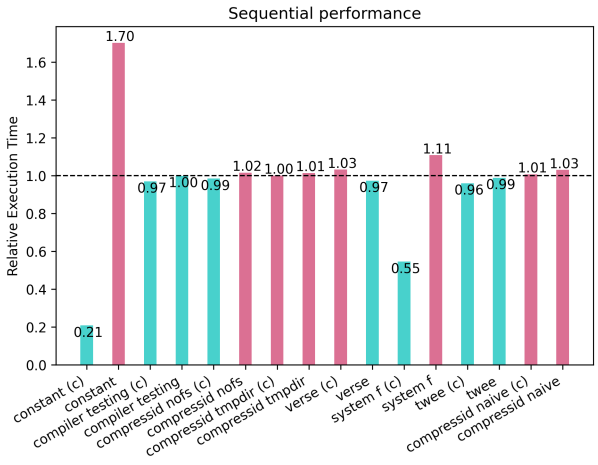


Figure 19: The sequential performance of QuickerCheck relative to QuickCheck, evaluated with the `chatty` flag turned on and off. The *(c)* suffix indicates that `Chatty` was set to `True`.

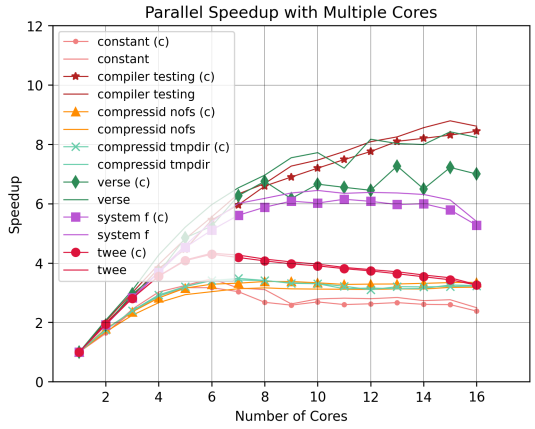


Figure 20: The speedup relative to sequential execution time, when `chatty` is enabled.