Search Party: a Haskell library for speculative parallelism in generate-and-test searches

Extended Abstract

Michael Walker
University of York
msw504@york.ac.uk

Colin Runciman
University of York
colin.runciman@york.ac.uk

Introduction

Generate-and-test computations are very common in functional programs, whether these be in the form of generating and testing statements in list comprehensions, or chained calls to `map` and `filter`. Although compilers generating sequential code are now very good at optimising this sort of thing, the opportunity for parallelism is enticing in cases where the elements of the resultant list can be computed independently.

List comprehensions are only one facet of generate-and-test problems, however. Searches which return a successful result, if one exists, from a pool of candidates can be regarded as a non-deterministic generate-and-test. Sometimes this nondeterministic result itself may be required, in which case the computation falls into the `IO` monad, but other times merely the presence or absence of a result must be known, restoring determinism.

Contributions

This work makes the following contributions:

- a new library called Search Party* for lazy parallel search, supporting both deterministic and nondeterministic searches;
- a number of illustrative examples of varying complexity, of which one is mentioned in the conclusions.

Related Work

GUM[4] introduces parallelism in Haskell with the `par` combinator, which starts evaluating its first argument in parallel and returns its second, and demonstrates a speed-up even with this primitive operation alone. In contrast, Search Party uses much more controlled parallelism, preventing a risk of a large amount of parallel computation being started at once, as is likely to happen with liberal application of `par` combinators.

The `Par` monad[2] provides for deterministic parallelism, and so might be seen as an obvious candidate to use, rather than the deterministic Search Party combinators. However, relaxing the requirement of determinism can be faster in some cases.

Perhaps nondeterminism is over-rated. Most programmers would only use a library like Search Party to parallelise list comprehensions and similar pure computations. Even here, the `Par` monad may not be suitable, as when a `Par` computation terminates, all parallelism must be complete, consider:

```
import Control.Monad.Par
parMapMaybe f = catMaybes . runPar . parMap f
```

This function loses the main flexibility of lazy lists, that the input list cannot be infinite.

The `Find` Monad

The goal of this work is to provide a parallel abstraction over generate-and-test powerful enough to handle many instances of this pattern. The interface is structured around a monad, `Find`. A `Find a` value represents a computation that, when executed, might *fail*.

The intuition behind the `Find` monad is:

- if there is a solution, we want one;
- if there are multiple solutions, we don't care which we get;
- as searching may be expensive, parallelism should be exploited as much as possible.

A `Find` computation is defined in terms of work items, which are explored in parallel. The Functor and Applicative instances preserve parallelism by deferring blocking on a result until it is actually required. The magic happens in the `<*>` operator:

```
(Find mf) <*> (Find ma) = Find $ do
  -- Begin computing 'f' and 'a' in parallel.
  f <- mf
  a <- ma

  -- Block until they are both done, or until
  -- one fails.
  successful <- blockOn [void f, void a]

  if successful
  then do
    -- Extract the results
    fres <- unsafeResult f
    ares <- unsafeResult a

    -- Return the result
    workItem' . Just $ fres ares
  else workItem' Nothing
```

*https://github.com/barrucadu/search-party

This is a similar approach to the Haxl work[3], where both sides of the `<*>` are explored in parallel for a result. Binding the value inside the `Find` begins the parallel search. As this computation is inside a `Find`, it will not be executed until the result is demanded.

The basic building-blocks for `Find` computations are `oneOf`, `success`, and `failure`, which together allow transforming a sequential search into a parallel one, by returning a successful result nondeterministically:

```
oneOf   :: Foldable t  => t (Find a) -> Find a
success :: a -> Find a
failure :: Find a
```

From these, we can derive the indexing operators `!` and `?` (which is an indexing operation using `Maybe`):

```
(!) :: Foldable t  => t a -> (a -> Bool) -> Find a
as ! p = oneOf . map p' $ toList as where
  p' a = if p a then success a else failure

(?) :: Foldable t => t a -> (a -> Maybe b) -> Find b
as ? f = oneOf . map f' $ toList as where
  f' = maybe failure success . f
```

Generalising to `Streams`

A natural progression from finding a single successful result is to find *all* such results:

```
allOf :: Foldable t => t (Find a) -> IO (Stream a)
```

A `Stream` is like a `Find` which contains multiple values. For example, we can find all perfect numbers:

```
perfect :: [Integer]
perfect = toList $ [1..] >! isperfect
  where
  isperfect n =
    sum [x | x <- [1..n - 1], n `rem` == 0] == n
```

Where the result is deterministic, a `Stream` can also be turned into a lazy list,

```
toList       :: IO (Stream a) -> [a]
unsafeToList :: Stream a -> [a]
```

The `unsafeToList` function is so called because it combines side-effectful reading with `unsafePerformIO` internally. It is only safe, then, if the `Stream` is *never used again* after passing it to `unsafeToList`. The `toList` function is safer as, rather than taking a `Stream` directly, it takes a computation which produces a `Stream`.

Regaining Determinism

Determinism can be restored by returning results in the order that they appear in the `Foldable.toList` enumeration of their container. In the case of `Find` this means returning the first successful result *in the list*, rather than the first successful result *to finish computing*. Two new basic functions are provided:

```
firstOf   :: Foldable t => t (Find a) -> Find a
orderedOf :: Foldable t => t (Find a) -> IO (Stream a)
```

Unfortunately, producing a deterministic result requires additional synchronisation, introducing additional blocking. This is why the nondeterministic functions are still provided, as they may be faster.

Version	Optimum	Actual	
-N1	0.49s (1.0x)	0.55s (0.9x)	0.9x
-N2	0.25s (1.9x)	0.35s (1.4x)	0.7x
-N4	0.14s (3.6x)	0.21s (2.3x)	0.6x
-N8	0.08s (6.4x)	0.16s (3.1x)	0.5x
-N16	0.04s (10x)	0.16s (3.1x)	0.3x
-N24	0.04s (13x)	0.18s (2.7x)	0.2x

Figure 1: Optimal speed-ups for Hutton's Countdown solver

Conclusions

We have presented a new library for parallel generate-and-test style computations, and demonstrated that it can give a significant speed-up over the sequential case. Furthermore, we have shown that it is not feasible to achieve the same flexibility as Search Party using traditional methods.

Parallelism and concurrency are generally seen as worthwhile tools to apply to large problems. Because of the difficulty of correctly using them, applying them to small problems is often simply not considered. Figure 1 shows the results of using Search Party with Hutton's Countdown solver[1]. The success of this technique on an already well tuned program is promising, and shows that this "parallelism in the small" *can* and *does* offer improved performance, and should be considered.

References

[1] G. Hutton. *Programming in Haskell*. Cambridge University Press, Jan. 2007.

[2] S. Marlow, R. Newton, and S. Peyton Jones. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82. ACM, 2011.

[3] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 325–337. ACM, 2014.

[4] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, 1996.