Parallelizing HPCpp code

Agustín Mista
mista@chalmers.se

Introduction

HPCpp [3] is a fork of the Haskell Program Coverage tool (HPC) [1] that can process multiple coverage files at once, combining them to produce fine-grain reports with detailed information about which property of the test suite produced each tick in the code during testing.

In the future, I plan to extend the ideas behind HPCpp in two possible directions: i) creating dynamic reports (inspired by QuickCheck-CI), and ii) automated test suite reduction based on coverage. However, for these ideas to work in practice, I need not only suitable representation for the coverage information (that now comes from multiple inputs), but also a basic set of fast operations to manipulate it. This report explores the process of implementing and improving the performance of such representation and operations using parallelism and playing with different data structures.[1].

Setup

Case studies With the hope of having somewhat representative results, I picked three case studies to be benchmarked in this report:

- **bst**: an implementation of binary search trees by John Hughes. Contains 79 individial coverage (tix) files adding up to 354kb of data.
- **primitive**: the test suite of the package `packages-primitive`, which implements some of GHC's primitive memory-related operations [4]. Contains 86 individual tix files adding up to 2.4Mb of data.
- **xmonad**: the test suite of the package `xmonad`, which implements the popular windows manager for Linux [5]. Contains 119 individual tix files adding up to 1.88Mb of data.

Each case study contains as many tix files as QuickCheck properties are defined in their corresponding test suites. They were generated after running each test suite using either a patched version of `tasty-quickcheck` [2] (**bst** and **primitive**), or a manually patched test suite (**xmonad**).

[1] All the code and results I present on this report can be found at `https://github.com/agustinmista/hpcpp/tree/master/bench`

Platform I used Criterion to implement every benchmark shown in this report. More specifically, the times shown correspond to the mean ones reported by Criterion. To run them, I used a desktop machine with a 6 cores (12 threads) CPU (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz) and 32gb of RAM.

The GHC version used to compile the code is 8.10.2, with the `-O2`, `-threaded` and `-feager-blackholing` flags. As for the RTS flags, I set the stack, heap and nursery size to 200Mb (`-K200m`, `-H200m`, and `-A200m`, respectively). All the sequential implementations were benchmarked using a single core (`-N1`), whereas the parallel ones were bencharked varying the number of cores from 1 to 6.

Sequential implementation and variants

In this section I briefly describe the sequential implementation of the code I used as an initial baseline, along with two variants using different data structures to store different parts the data.

The tix files generated by HPC use a simple representation: a list of tix modules, where each tix module contains a list of coverage ticks (of type `[Integer]`) and some metadata:

```
data Tix = Tix [TixModule]
data TixModule = TixModule String Hash Int [Integer]
```

In my implementation, I want to be able to distinguish which property produced each tick, so instead of using `Integer`s, I use a mapping from property names to ticks (`TicksCount`), the rest of the data is simply a mirror from that of the original tix implementation:

```
data MTix = MTix [MTixModule]
data MTixModule = MTixModule String Hash Int [TicksCount]
type TicksCount = [(String, Integer)]
```

Using these data types, I am interested in four main operations: *read*, *merge*, *project* and *expression coverage*, which I describe in detail below.

Reading The first operation takes care of reading and parsing the input tix files. For this, I rely on the original HPC implementation (`readTix`), and wrap it with a transformation from `Tix` to `MTix` that creates a singleton tick count for each tick on the original tix file:

```
readMTixs :: [FilePath] -> IO [MTix]
readMTixs = mapM readMTix

readMTix :: FilePath -> IO MTix
readMTix file = do
  mtix <- readTix file
  case mtix of
    Nothing -> error $ "error reading tix file from: " ++ file
    Just a -> return (tixToMTix (takeBaseName file) a)
```

```
tixToMTix :: String -> Tix -> MTix
tixToMTix prop (Tix xs) =
  MTix (toMTixModule <$> xs)
  where
    toMTixModule (TixModule n h i ticks) =
      MTixModule n h i (tick prop <$> ticks)
```

Merging The second operation I consider in this implementation takes care of merging the singleton `MTix` values I read using `readMTixs` into a single `MTix` that encompasses all the information. For this, I can simply fold the inputs using a merging operation. This operation zips the singleton tick information using monoid concatenation (`<>`) across every module:

```
mergeMTixs :: [MTix] -> MTix
mergeMTixs [] = error "mergeMTixs: empty input"
mergeMTixs xs = foldr1 mergeMTix xs

mergeMTix :: MTix -> MTix -> MTix
mergeMTix (MTix ts1) (MTix ts2) =
  MTix (zipWith mergeMTixModule ts1 ts2)

mergeMTixModule :: MTixModule -> MTixModule -> MTixModule
mergeMTixModule (MTixModule n1 h1 i1 tks1) (MTixModule n2 h2 i2 tks2)
  | n1 == n2 && h1 == h2 && i1 == i2 =
      MTixModule n1 h1 i1 (zipWith (<>) tks1 tks2)
  | otherwise =
      error "mismatched modules!"
```

Projection The next operation I implemented serves the purpose of projecting specifing properties from an `MTix` value, which can come in handy when implementing a test suite reduction algorithm in the future. This operation takes a list of properties to project and simply filters them on every tick count:

```
projectMTix :: [String] -> MTix -> MTix
projectMTix props (MTix ts) = MTix (projectMTixModule props <$> ts)

projectMTixModule :: [String] -> MTixModule -> MTixModule
projectMTixModule props (MTixModule n h i tks) =
  MTixModule n h i (projectTicksCount props <$> tks)

projectTicksCount :: [String] -> TicksCount -> TicksCount
projectTicksCount props = filter ((`elem` props) . fst)
```

Expression coverage The last operation I consider here takes care of calculating simple code coverage statistics. For simplicity, I only care about expression coverage here, but the implementation for the rest of the criteria (alternatives, top-level definitions, boolean conditions, etc.) supported by HPC should be analogous. This operation is a bit more involved than the rest, as the long list of ticks contained in the original tix files represents several kinds of syntactic elements in the code (corresponding to the different criteria mentioned above),

and here we only care about those that represent expressions. To distinguish them, we need to use a secondary set of files produced by HPC, known as *mix* files. These files contain the *kind* of tick associated to each tick value within a tix module.

Here I assume that these auxiliary files are already parsed and are given as an input to our algorithm, using a mapping from module names to their corresponding mix information. Then, our algorithm looks up for those ticks that represent expressions in the code, and transform each one them into a tuple *(covered, total)* that can be folded across different modules in order to calculate expression coverage statistics in a per-module basis:

```
expCoverMTix :: [(String, Mix)] -> MTix -> [(String, Double)]
expCoverMTix mixs (MTix mods) = do
  expCoverMTixModule mixs <$> mods

expCoverMTixModule :: [(String, Mix)] -> MTixModule -> (String, Double)
expCoverMTixModule mixs (MTixModule name hash size tix) = do
  (name, (fromIntegral cov * 100) / fromIntegral tot)
  where
    (cov, tot) = foldr expCover (0,0) (zip mix tix)
    Just (Mix _ _ _ _ mix)  = lookup (takeFileName name) mixs
    mixName (Mix m _ _ _ _) = takeFileName m

expCover :: (MixEntry, TicksCount)
         -> (Integer, Integer) -> (Integer, Integer)
expCover ((_, ExpBox _), ticks) (cov, tot)
  | ticksTotal ticks > 0  = (cov+1, tot+1)
  | otherwise             = (cov,   tot+1)
expCover _ (cov, tot)     = (cov,   tot)
```

Implementation variants

The initial implementation uses lists everywhere, which are usually a concern when it comes to performance. For this reason, I decided to implement two different variants of the implementation above, changing some data structures here and there:

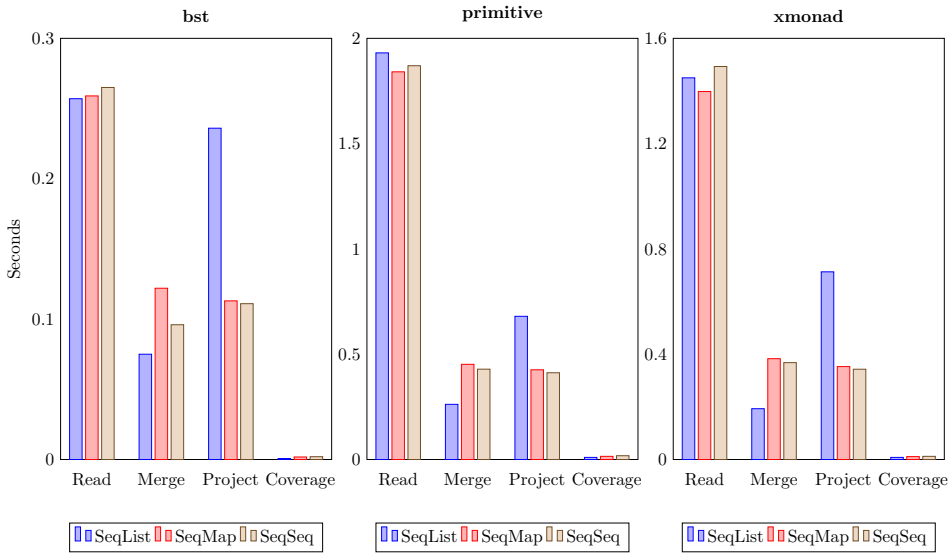
- **SeqList**: the sequential implementation using lists shown above.
- **SeqMap**: like **SeqList**, except that individual tick counts are implemented using finite maps (implemented using balanced binary trees [6]), as:

  ```
  type TicksCount = Map String Integer
  ```

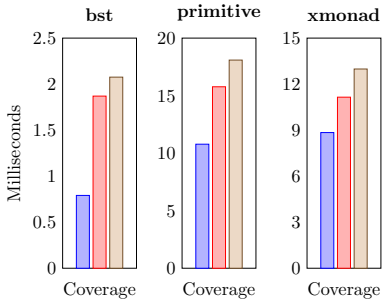
- **SeqSeq**: like **SeqMap**, except that modules use sequences of tick counts (implemented via 2-3 trees [7]) instead of lists:

  ```
  data MTixModule = MTixModule String Hash Int (Seq TicksCount)
  ```

To compare these variants, I first ran each operation on each case study, varying the input size from one to the full set of inputs. The results there shown that the execution time *always grows linearly over the number of inputs*, so I decided to only consider from now on: the full set of inputs for the read, merge, and expression coverage operations, and half of the inputs for the projection operation (so I do not end up with the same input value as output). With these considerations, the times required to perform each operation are as shown below:



And since the expression coverage operation takes a considerably shorter time on average, below is a "zoomed" version of the times required by this operation (notice the Y scale is in milliseconds!):



At simple sight, it seems that the larger improvement comes from using finite maps for tick counts when performing projections. The rest of the operations show some slowdown when using data structures other than lists, due to better deforestation support and optimization rules for lists perhaps? Using sequences slightly helps reducing times when merging, and projecting, but adds a little overhead when reading and measuring coverage.

After these observations, I decided to continue the benchmarks using the **SeqSeq** variant as the baseline for the sequential implementation when comparing with other parallel ones (implemented as variants of **SeqSeq**).

Parallel implementations

It is finally time to start speeding things up using more than one CPU core at a time. In this section I present different parallel variants of each of the four operations described so far.

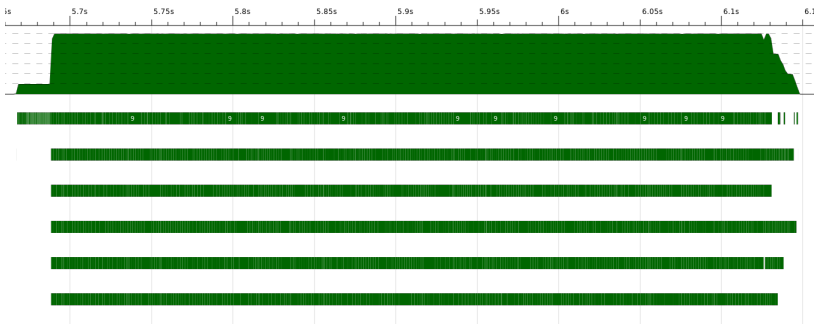
An important thing to notice is that all the parallel variants I present in this section are implemented using parallel strategies from the `parallel` package [8] by Simon Marlow. I did not find noticeable differences when trying to implement them using the `Par` monad, so I saw no point in comparing them here.

Parallel Read

To parallelize this operation I simply make sure that each element of the output is evaluated in parallel on a different spark:

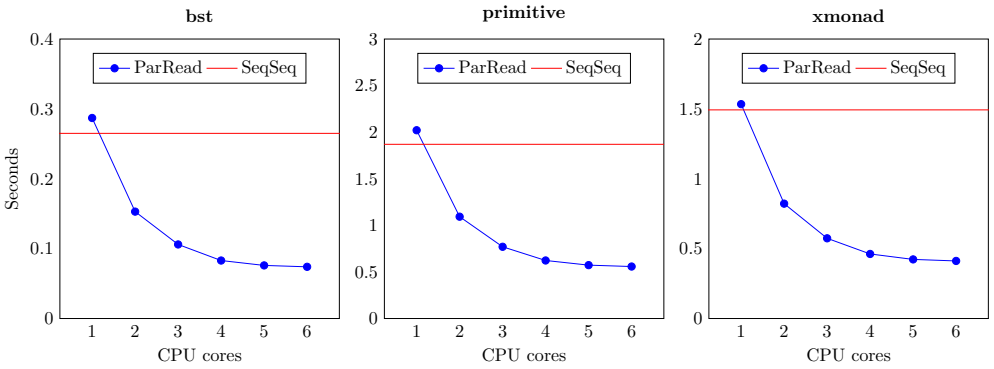
```
readMTixs :: [FilePath] -> IO [MTix]
readMTixs files = do
  mtixs <- mapM readMTix files
  return (mtixs `using` parList rdeepseq)
```

Acording to ThreadScope, this simple tweak produces a nice green block of CPU usage when ran in parallel using 6 cores for the **primitive** case study (the rest of the ThreadScope figures on this report were obtained using the same case study and number of cores):



I suspect that the single threaded work that happens at the beginning of this operation could be a consequence of listing the directory where inputs are located using the `System.Directory.listDirectory` function.

Benchmarks I benchmarked this operation in function of the number of CPU cores used on each case study:



These results indicate that it is possible to obtain a performance increase of around 3.5x when using 6 CPU cores. With this result in place, I decided to move onto the next operation, since obtaining an ideal ∼6x speedup will be very unlikely to happen due to the added overhead. Also, I consider than spliting the input in a very small number of chunks will hardly make things better, as we only have a handful of inputs (a couple of hundreds at most), which should not incur into a noticeable overhead in terms of sparked computations. By the contrary, it could harm the performance if some chunk contains inputs larger than the average.

Parallel Merge

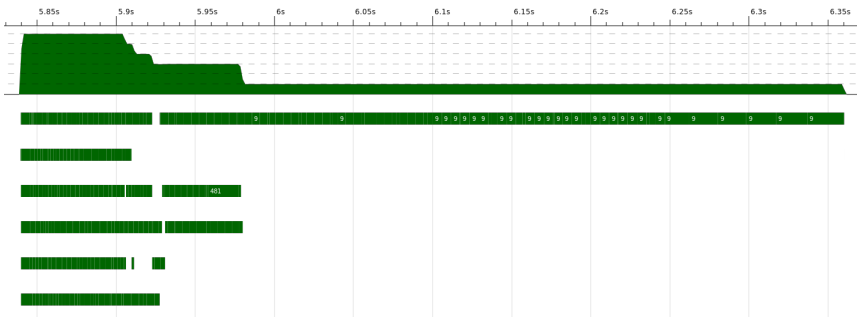
Parallelizing this operation was quite problematic, as every parallel implementation I tried ended up being much slower than the sequential one. After dozens of combinations of implementations, evaluation strategies, compilation flags, RTS flags, parallel frameworks, I could not find a single one that could show a noticeable speedup versus the sequential baseline. By the contrary, none of them seemed to be able to be even on par to the baseline.

At the light of these sad results, I decided to carry on and report three implementations using different parallel granularity and analyse what could be happening.

ParMerge1 In this first implementation, I decided to parallelize the `mergeMTix` operation using a bottom-up reduction approach, exploiting the associativity of this binary operation:

```
mergeMTixs :: [MTix] -> MTix
mergeMTixs [] = error "mergeMTixs: empty input"
mergeMTixs xs = parFold1 rdeepseq mergeMTix xs

parFold1 :: Strategy a -> (a -> a -> a) -> [a] -> a
parFold1 s f [x] = x
parFold1 s f xs = parFold1 s f (reduce1 f xs `using` parList s)

reduce1 :: (a -> a -> a) -> [a] -> [a]
reduce1 _ [] = []
reduce1 _ [x] = [x]
reduce1 f (x:y:ys) = f x y : reduce1 f ys
```

Inspecting the ThreadScope output, we can see that some parallelism is happening during the computation:



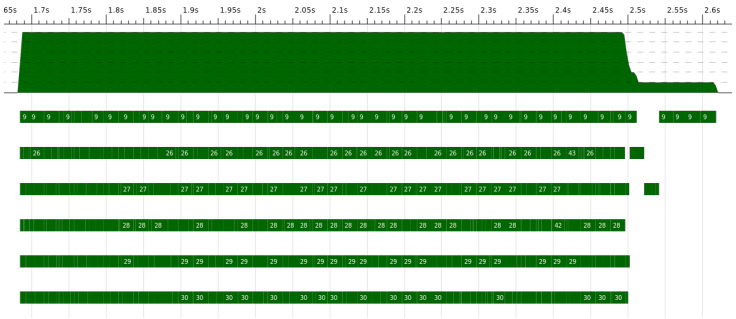
The shape of the CPU activity is easy to interpret: the first levels in the bottom-up reduction strategy contain at least a few pairs of inputs that can be sparked in parallel. However, as we approach the last levels, we only have a handful of pairs of larger values to merge, thus some CPUs will eventually start idling. In the last level, we only have two very large values to merge, and this operation will be sparked in a single CPU, taking most of the computation time.

In principle, this result would suggest that we have the wrong parallel granularity in this implementation, though it should not be much slower than the sequential version.

ParMerge2 In the second variant of this operation, I moved the paralelism to the module level, processing the fold operation sequentially, and zipping `MTixModule`s in parallel via a parallel version of the `zipWith` combinator:

```
mergeMTixs :: [MTix] -> MTix
mergeMTixs [] = error "mergeMTixs: empty input"
mergeMTixs xs = foldr1 mergeMTix xs

mergeMTix :: MTix -> MTix -> MTix
mergeMTix (MTix ts1) (MTix ts2) =
  MTix (parZipWith rdeepseq mergeMTixModule ts1 ts2)

parZipWith :: Strategy a -> (a -> a -> a) -> [a] -> [a] -> [a]
parZipWith s f xs ys = zipWith f xs ys `using` parList s
```

This resulted in better CPU usage acording to ThreadScope, altought we can still see a tail of uneven work:

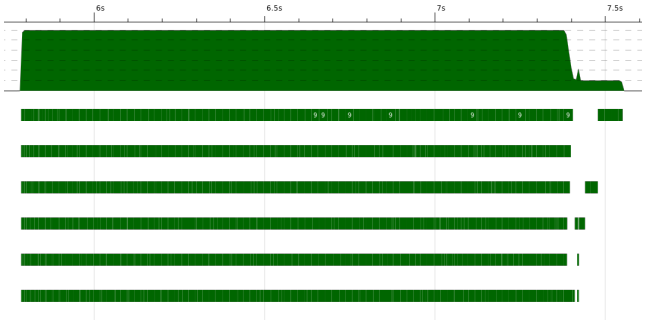


However, this implementation is much slower than the previous unbalanced one! (Benchmark results are at the end of this subsection.)

ParMerge3 In a final attempt to parallelize this operation, I reduced the granularity to chunks of tick entries on each module. This is done similarly as before, except that I explicitly split the sequences of tick entries of each module into the number of CPUs available, with a minimum of 500 elements to ensure a reasonably large task on each spark:

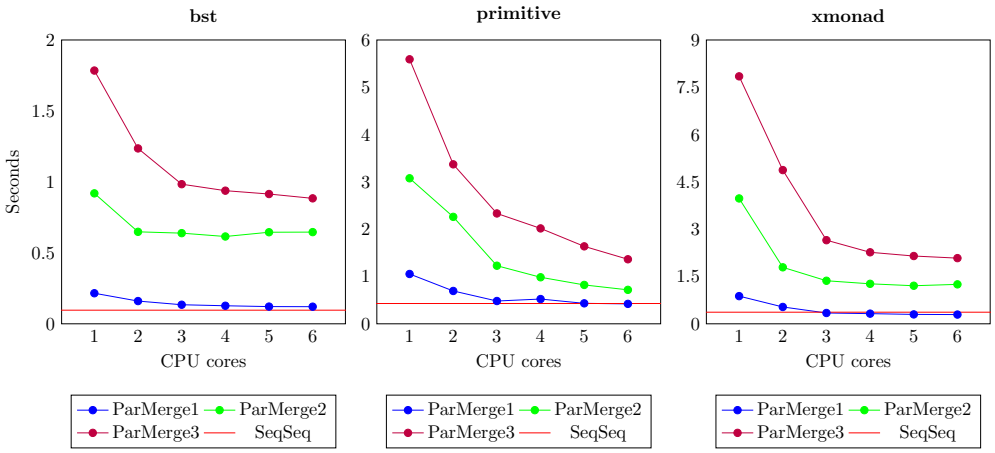
```
mergeMTixs :: [MTix] -> MTix
mergeMTixs [] = error "mergeMTixs: empty input"
mergeMTixs xs = foldr1 mergeMTix xs

mergeMTix :: MTix -> MTix -> MTix
mergeMTix (MTix ts1) (MTix ts2) =
  MTix (parZipWith rdeepseq mergeMTixModule ts1 ts2)

mergeMTixModule :: MTixModule -> MTixModule -> MTixModule
mergeMTixModule (MTixModule n1 h1 i1 tks1) (MTixModule n2 h2 i2 tks2)
  | n1 == n2 && h1 == h2 && i1 == i2 =
      let n = max 500 (i1 `div` numCapabilities)
      in  MTixModule n1 h1 i1 (parSeqZipWithChunk rdeepseq n (<>) tks1 tks2)
  | otherwise =
      error "mismatched modules!"

parZipWith :: Strategy a -> (a -> a -> a) -> [a] -> [a] -> [a]
parZipWith s f xs ys = zipWith f xs ys `using` parList s

parSeqZipWithChunk :: Strategy (Seq a) -> Int -> (a -> a -> a)
                   -> Seq a -> Seq a -> Seq a
parSeqZipWithChunk s n f xs ys =
  concatSeqs (fmap zipChunk chunks `using` parTraversable s)
  where
    concatSeqs = foldr (Seq.><) Seq.empty
    zipChunk = fmap (\(x, y) -> f x y)
    chunks = Seq.chunksOf n (Seq.zip xs ys)
```

The ThreadScope output for this implementation shows a similar result as before, with a (slightly smaller) tail of uneven work:



More bad news, however. This implementation is even slower than the previous one! Also, given that the tail of uneven work has not dissapeared completely, I doubt that the one on the previous implementation was caused due to a too coarse granularity.

Benchmarks The benchmarks of the different variants I implemented for this operation show a strange pattern: it seems that adding more CPU cores to the formula helps making them faster, but the whole result is almost an order of magnitude slower:



We can also notice that as we reduce the granularity of the parallel tasks, the performance degrades considerably. This effect would be reasonable if our code is spawning too many small parallel sparks which contribute to the overhead. However, the faster parallel implementation, **ParMerge1**, is the one that clearly seemed it could benefit from a finer task granularity.

The reason behind the huge slowdowns of every implementation I tried remains a mistery for me to this very day. My best guess at what it going on is that the parallel evaluation strategy is too strict, reducing subterms to normal form repeatedly. However, replacing `rdeepseq` with different strategies resulted in computations that seem to be running in a single core according to ThreadScope (no parallelism whatsoever). Another reason could be perhaps that the benchmark itself is for some reason adding up the time required to reduce the output to normal form. Hopefully, a reader trained in the art of parallelizing

Haskell code will point out what I did wrong.

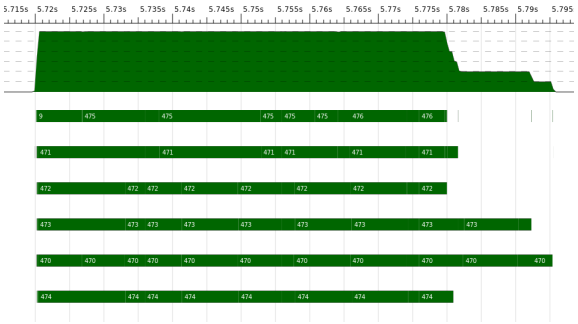
Parallel Projection

For the third operation, I implemented two variants of the projection operation, which differ on the parallel task granularity.

ParProject1 For this first attempt, I replaced the sequential map of `projectMTixModule props` for its parallel counterpart, provided by the `parallel` package:

```
projectMTix :: [String] -> MTix -> MTix
projectMTix props (MTix ts) =
  MTix (parMap rdeepseq (projectMTixModule props) ts)
```

This simple solution produced a good amount of parallel CPU usage when inspected using ThreadScope:



Although I was not very happy about the tail of uneven parallel work at the end of the computation, which was the motivation for reducing the task granularity in the second variant.

ParProject2 This variant works just like **ParProject1**, except that it also divides the sequence of tick entries on each module in chunks that are processed in parallel:

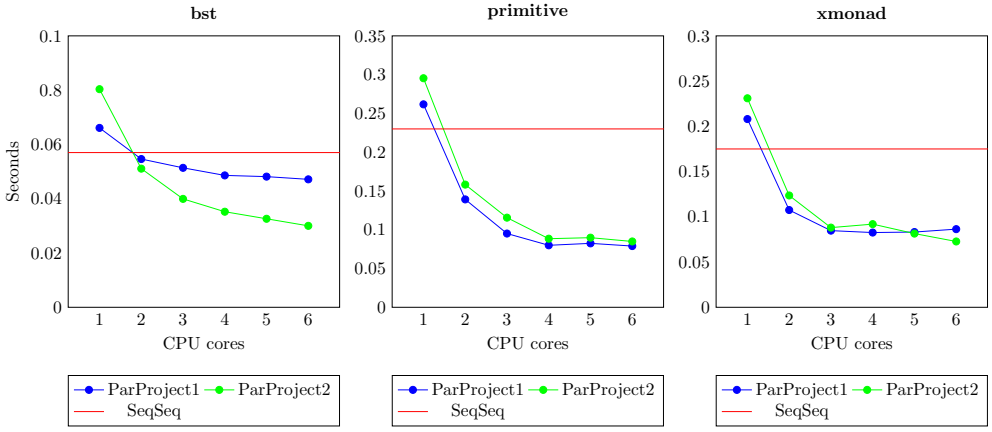
```
projectMTix :: [String] -> MTix -> MTix
projectMTix props (MTix ts) =
  MTix (parMap rdeepseq (projectMTixModule props) ts)

projectMTixModule :: [String] -> MTixModule -> MTixModule
projectMTixModule props (MTixModule n h i tks) =
  MTixModule n h i (parSeqMapChunk rdeepseq 50 (projectTicksCount props) tks)

parSeqMapChunk :: Strategy (Seq b) -> Int -> (a -> b) -> Seq a -> Seq b
parSeqMapChunk s n f xs =
  concatSeqs (fmap (fmap f) chunks `using` parTraversable s)
  where
    concatSeqs = foldr (Seq.><) Seq.empty
    chunks = Seq.chunksOf n xs
```

The function `parSeqMapChunk` divides a sequence into chunks, evaluates them in parallel, to finally concatenate them. After manually tunning the chunk size, I found that processing chunks of 50 elements works well in most cases.

The ThreadScope output for this variant is as below:

This result shows that this parallel computation now distributes work more evenly across cores during its whole lifetime.

Benchmarks The benchmarks of both variants in terms of number of cores for each case study are as shown below:



These results give us a couple of hints:

- If our `MTix` contains very few modules like the **bst** case study (only two modules), it is crucial to divide the inner sequences into chunks in order to use all the available CPUs.
- When our `MTix` data contains a larger number of modules, the **ParProject2** variant carries a little overhead associated with the "chunkification" of the tasks. Otherwise they behave quite similarly.

With these considerations in mind, I would lean to use the **ParProject2** variant, as I think it would give good results not only in large but also small codebases. Concretely, this variant achieves around 2x∼3x of speedup.

Parallel Coverage

For the last operation I consider in this report, I also implemented two variants with different level of parallel granularity.

ParCoverage1 Just like for the case of **ParProject1**, this first variant maps the coverage operation over each module in parallel:

```
expCoverMTix :: [(String, Mix)] -> MTix -> [(String, Double)]
expCoverMTix mixs (MTix mods) = do
  parMap rdeepseq (expCoverMTixModule mixs) mods
```

And as expected, we can see a (small) tail of uneven parallel work when we inspect this computation using ThreadScope:

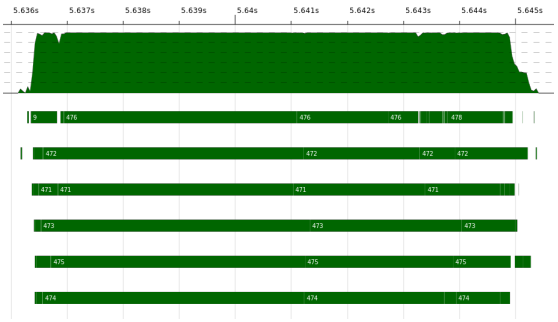
To solve this issue, the next variant also reduces the parallel granularity of the computation.

ParCoverage2 As before, this implementation also splits the sequence of tick entries of each module into chunks. However, this computation also needs to combine the coverage results from each chunk into a single one. Thus, it is easy to see how this operation can be implemented in terms of a parallel MapReduce combintator:

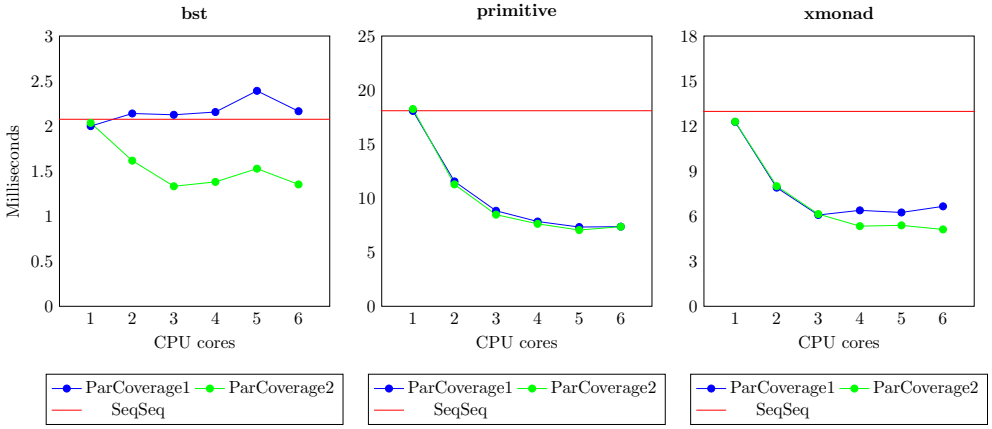
```
expCoverMTix :: [(String, Mix)] -> MTix -> [(String, Double)]
expCoverMTix mixs (MTix mods) = do
  parMap rdeepseq (expCoverMTixModule mixs) mods

expCoverMTixModule :: [(String, Mix)] -> MTixModule -> (String, Double)
expCoverMTixModule mixs (MTixModule name hash size tix) = do
  (name, (fromIntegral cov * 100) / fromIntegral tot)
  where
    (cov, tot) = mapReduceChunks rdeepseq 200 mapper reducer pairs
    reducer = foldr (\(c1, t1) (c2, t2) -> (c1+c2, t1+t2)) (0,0)
    mapper = foldr expCover (0,0)
    pairs = Seq.zip (Seq.fromList mix) tix
    Just (Mix _ _ _ _ mix)  = lookup (takeFileName name) mixs
    mixName (Mix m _ _ _ _) = takeFileName m
```

```
mapReduceChunks :: Strategy b -> Int
                -> (Seq a -> b)
                -> (Seq b -> c)
                -> Seq a -> c
mapReduceChunks s n mapper reducer xs =
  reducer (fmap mapper chunks `using` parTraversable s)
  where
    chunks = Seq.chunksOf n xs
```

After some manual tunning, I found that using chunks of 200 elements works reasonably good on every case study. If we inspect the ThreadScope output, we can see that the parallel work is now more evenly distributed across cores:



Benchmarks The final set of benchmarks presented in this report compare the performance of the different expression coverage implementations:



These results are similar to those obtained before for the projection operation, except that here the most granular approach seems to outperform the less granular one on every case study, and especially on the small **bst** one. Concretely, **ParCoverage2** obtained speedups of between 1.5x to 2.5x.

Conclusions

Altought in theory it seems rather easy, achieving a good level of parallelism in Haskell can be quite intricate, as it requires a large variety of case studies to test with, and a good eye for finding the best parallel granularity.

Some operations like map and map-reduce are relatively straightforward to implement and to tweak. However, others like transpositions of data, e.g., the merge operation here, are more tricky to reason about, especially if we consider Haskell's lazy evaluation, and the contradicting fact that we are sometimes forced to fully evaluate things in order to get better parallelism.

However, even if I only managed to obtain a $\sim$0.75x speedup per core the best scenario I tested (and almost none in the worst), the results I found throughout this project motivate me to continue digging on the future ideas behind it, namely test-suite reduction and better dynamic coverage reports.

References

[1] Gill, Andy, and Colin Runciman. "Haskell program coverage." Proceedings of the ACM SIGPLAN Workshop on Haskell. 2007.

[2] `https://github.com/agustinmista/tasty-quickcheck-hpc`

[3] `https://github.com/agustinmista/hpcpp`

[4] `http://hackage.haskell.org/package/primitive`

[5] `http://hackage.haskell.org/package/xmonad`

[6] `https://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Strict.html`

[7] `https://hackage.haskell.org/package/containers-0.6.3.1/docs/Data-Sequence.html`

[8] Marlow, S., Maier, P., Loidl, H. W., Aswad, M. K., and Trinder, P. (2010). Seq no more: better strategies for parallel Haskell. ACM Sigplan Notices, 45(11), 91-102.