Parallel Boggle

by Khyber Sen on 12/18/2019

Project for COMS 4995 Parallel Functional Programming with Stephen Edwards.

Obtaining and Viewing

To run this, you can clone it from github if you don't already have it:
`git clone git@github.com:kkysen/ParallelBoggle.git`
If you already have the source, it should also contain sample event logs (in `eventlogs/`) from different stages of development where I tried different methods of parallelism. A PDF of this README is also included in the repo.

The project proposal, in LaTeX and PDF form, are in the `proposal/` directory, which explains the basis of the project.

Compiling

To run, you first need to install a few libraries: `apt install libblas-dev liblapack-dev libgsl-dev`. This is for GNU GSL, which contains a simulated annealing library I'm using (more on this below).

I'm also using a forked version of `bytestring-trie` to access its internals. It should work seamlessly without having to do anything.

To compile, just run `stack install`, which produces `ParallelBoggle-exe` in a directory that should be on your `PATH`.

Running

To run, run `ParallelBoggle-exe <m> <n> <numTries> <numItersPerTemp> <coolingRate> <minTemp>`. `<m> <n>` is the size of the boggle board, and the rest are simulated annealing parameters.

For example, you can run `ParallelBoggle-exe 70 70 1 10 1.05 1`, which will optimize a large `70 x 70` board but only slightly optimize it.

You can also run `ParallelBoggle-exe 10 10 5 50 1.05 0.5`, which will optimize a smaller board but run more iterations and thus optimize it more.

To highly optimize a smaller board, run `ParallelBoggle-exe 4 4 10 5000 1.05 0.05`. The parallelism is much worse on this, though, as explained below, and it runs too long for ThreadScope as well.

You can add `+RTS -N<parallelism>` to control the parallelism as well.

When the program runs, intermediate "energies" are printed, and at the end, the optimized board is printed out, along with its score and iteration number.

Problems Before Parallelism

After writing the serial version of the boggle solver, which stores the dictionary in a trie and performs a depth-first search of the board using the dictionary and a bitset of the path to filter out new branches, I found that for small boards, the running time was dominated by building the trie, not actually performing the search. For a `4 x 4` board, building the trie dictionary (`data/sowpods.txt`, which is the Scrabble dictionary) took around ~1.5 seconds, but solving the board only took a few milliseconds (measured by running thousands of boards). I was trying to parallelize the boggle solving, not the trie building, so I was left with a few options:

- Drastically optimize the trie building by storing it statically in a file, which I'd then load at runtime. The trie wouldn't be built at runtime, it'd be pre-built and saved in the file in trie form, using indices instead of pointers and embedding them in the file.
  This is especially hard to do in a garbage-collected, memory-managing language like Haskell, and I'd have to write my own trie library to do this. I tried using `fixfile` for this, but it wouldn't compile.

- Run much larger boards, which still take a while.
  This is a good option, but it still wasn't great, since there aren't many words in the dictionary that are extremely long, so as the board size increases, the depth of the search maxes out. This leads to tons of short (depth), fast searches, which are a bit harder to parallelize since they still run very fast. In hindsight, perhaps I should've concentrated more on this.

- Go back to the idea of finding the highest scoring board of a given size. Amanda recommended against this, saying to just focus on the finding words in the boggle board part of the problem. But by using simulated annealing, I could force an inherently serial algorithm that repeatedly solves smaller boards. The serialness of this meant I wasn't just solving, say a 1000 random boards, which could be parallelized trivially. Using the serial simulated annealing algorithm forced me to focus the parallelism in solving each board.
  And since simulated annealing runs so many iterations, the runtime won't be dominated by building the trie, but by solving thousands of boards.

- Parallelize the trie building itself. This would have helped speed it up, although it is still many order of magnitude slower than solving small boards, so I'm not sure how much this would've helped. This might've been a better place to focus my parallelism efforts in hindsight, but I did want to focus on parallelizing the board search itself, since that was my main idea.

I chose the third option. I didn't want to focus much on the simulated annealing, however, so I found a library on hackage. It uses GSL, a C library, to do the actual simulated annealing, which is why you need to `apt install` some libraries. In hindsight, I should've just written the simulated annealing algorithm in Haskell myself, since it's simpler than I realized. This would have helped a lot with issues I had with randomness, since monadic randomness can't really cross into an FFI API not designed with monads. Furthermore, I found what seems to be a bug in `mkStdGen`. It silently and undocumented (I had to check the source code) converts an `Int` seed into an `Int32` seed, which means `Int` seeds over the `2^31 - 1` lose their randomness. This led to me spending a long time debugging why the simulated annealing wasn't changing the boards at all.

Parallelism

Note: event logs for most of these tries are included in `eventlogs/`. The `N1` are the serial version and the `N8` are the parallel versions.

I thought parallelizing the search algorithm would be fairly straightforward. I could just do some sort of a parallel map over each of the neighboring letters I explore in the search. I could limit this to just the first round (`Depth 1`), where I just search from all the letters on the board, which I thought would be enough parallelism. Or I could parallelize deeper (`All Depths`) and set a threshold beyond which I stopped parallelizing the search.

However, when I tried implementing the former idea, the parallelism actually slowed down the algorithm slightly with 8 logical cores. The latter idea (`Path Lenght 3`) fared a bit better, but not by much. The threshold I used was how long the current word path was. But the parallel version only barely improved over the serial version.

After inspecting the event logs in ThreadScope, I realized the threads weren't doing nearly enough work, even though that work was very well balanced. I assume this is because I was running small, `4 x 4`, boards. Although simulated annealing was running them tens of thousands of times, each board was solved so fast that parallelizing each board wasn't very efficient since the parallelism was too finely-grained.

When I ran larger boards, the parallelism improved, but still not by much, only by about 20%. I did have to drastically reduce the number of iterations run in the simulated annealing, since the event logs being produced were too large for ThreadScope. After inspecting ThreadScope, I saw that most of the sparks being created were wasted, i.e. fizzled or GC'd, instead of being converted.

At this point, I was using strategies to parallelize, using a `` `using` parList rdeepseq ``, since `deepseq` forced full evaluation of the found word paths, unlike `seq`. I decided to try using the Par monad instead, since that should lead to less wasted sparks, but when I used `parMap`, both on the initial neighbors and before a certain threshold, the performance was even worse, and the event logs were so huge even for shorter-running programs that ThreadScope could only read them after 10 minutes and 10 GB of RAM.

Thus, I returned to using strategies, which at least led to some, albeit not much, parallel performance improvement. To limit the number of sparks wasted, I realized that using the current length of the path was not the best proxy for how much work was left in that search branch, and thus lead to poor load balancing. Instead, I decided to use the size of the remaining dictionary at that point as a better proxy for how much work was left. In the search algorithm, after exploring a neighboring letter, the sub-trie dictionary containing that letter as a prefix is passed through to the next recursion, meaning the size of that dictionary was proportionate to the amount of possible words remaining down that branch.

However, `Trie.size` is `O(n)`, meaning adding it would likely slow down the algorithm. When I used `Trie.size` naively, I finally got better parallel performance. But this is because the parallel version got only slightly worse (than without `Trie.size`) and the serial version got twice as worse, leading to a 2x speedup (still not great). The threads were all doing much more work when viewing ThreadScope, but this was just because the work was now half in `Trie.size`, not the search itself.

I figured out a way to make `Trie.size O(1)`. I had to access the internals of the library to do this, though, so I forked `bytestring-trie` and made a modified version that just exposes the data constructor of `Trie`, so I could implement my own more internal algorithms. (This fork is included in `stack.yaml`, so it should just work out of the box with `stack`.)

I used this to convert a `Trie ()`, my original dictionary, into a `Trie Int`, where the `Int`s at each leaf represented the size of the sub-trie at that point. This made computing the size of a `Trie` `O(1)`, since either the leaf is at the root, in which case it'd be a simple lookup, or there is no leaf at the root, in which we have to iterate over the immediate branches, but there is a constant bound over the number of such branches.

When I switched to using this `O(1)` version of `Trie.size` (`Dict Size 10000 - 70x70`), the serial version went back to normal, but the parallel version didn't have a proportionate improvement. It still did better than before, but the parallelism was still only 1.5x - 2x.

However, when looking at ThreadScope, far more of the sparks were now converted, about 2/3 now, instead of barely any before. It doesn't appear that this had that significant of an effect on the runtime, unfortunately, and I think this is still because each spark isn't doing enough work, so the parallelism is too fine-grained. If I make the sparks do more work, however, the threads have more work to do sometimes, but then the load balancing becomes worse again. This tradeoff results in about the same speedup for different dictionary size thresholds.

Although I'm not sure and I might just be missing something about the parallelism, the root of this problem seems to lay in the fact that English words aren't long enough. There are a ton of English words, more than most languages, but since there aren't a ton of very long words, I can't split the parallelism that deep without the amount of work per spark going way too low. Perhaps if I used a different language, like German, that has more long words, this could be improved, but at this point it's too late to try that.

Perhaps I should've focused the parallelism in the board optimization instead, either making simulated annealing parallel or using another stochastic optimization algorithm like a parallel genetic algorithm. In these cases, the actual parallel algorithm might've been harder, but each board should take roughly similar amount of times, leading to excellent load balancing, and introducing parallelism at the board level (not in the board), would've led to coarser-grained parallelism that would've left each spark much more work to do, resulting probably in a much better parallel speedup.

Code

I'm not sure what is meant by including listings of all the code in the PDF. Although most of the core parallelism and search algorithm are in `src/Boggle.hs` in the `newWithScorer` function, there are important parts spread across modules. It seems weird to include them all in this report instead of just viewing the files themselves. Nevertheless, here are some of the important parts of the code concerning the main algorithm and parallelism (excluding simulated annealing and optimization):

`Boggle.hs`:

```
 type IJ = Int -- (Int, Int) packed into one Int
type PathElement = Int -- (Word8, IJ) packed into one Int
type Neighbors = [PathElement]
type BitSet = Integer -- BitSet used as Bits BitSet
type Path = ([PathElement], BitSet)
type PathDictElement = (PathElement, BitSet, Dict)

data FoundWord = FoundWord {
    score :: Int,
```

```haskell
    word :: ByteString,
    pathSet :: BitSet,
    path :: [IJ]
}

instance Eq FoundWord where
    a == b = word a == word b

instance Ord FoundWord where
    a `compare` b = word a `compare` word b

data Solution = Solution {
    words :: [FoundWord],
    totalScore :: Int,
    board_ :: Board
}

type Scorer = Int -> Int -- length to score

data Boggle = Boggle {
    parallel :: Bool,
    board :: Board,
    scorer :: Scorer,
    get :: IJ -> Word8,
    toNeighbors :: [IJ] -> Neighbors,
    startingPathSet :: BitSet,
    startingNeighborIndices :: [IJ],
    startingNeighbors :: Neighbors,
    neighborIndices :: IJ -> [IJ],
    neighbors :: IJ -> Neighbors,
    searchFrom :: PathDictElement -> [Path],
    searchIndices :: Dict -> BitSet -> [IJ] -> [Path],
    toFoundWord :: Path -> FoundWord,
    solve :: Lang -> Solution
}

newWithScorer :: Scorer -> Board -> Bool -> Boggle
newWithScorer scorer board runInParallel = Boggle {
    parallel = runInParallel,
    board,
    scorer,
    get,
    toNeighbors,
    startingPathSet,
    startingNeighborIndices,
    startingNeighbors = toNeighbors $ startingNeighborIndices,
    neighborIndices,
    neighbors = toNeighbors . neighborIndices,
    searchIndices,
    searchFrom,
    toFoundWord,
    solve
}
  where

    Board {board = boardArray, size = (m, n)} = board

    size = m * n

    fromIJ :: IJ -> (Int, Int)
    fromIJ = (`divMod` m)

    toIJ :: (Int, Int) -> IJ
    toIJ (!i, !j) = i * n + j

    get :: IJ -> Word8
    get ij = BS.index boardArray ij

    fromPathElement :: PathElement -> (Word8, IJ)
    fromPathElement e = (fromIntegral e .&. 0xFF, e `shiftR` 8)

    toPathElement :: (Word8, IJ) -> PathElement
    toPathElement (!c, !ij) = fromIntegral c .|. ij `shiftL` 8

    toNeighbors :: [IJ] -> Neighbors
    toNeighbors = map (\ij -> toPathElement (get ij, ij))

    startingPathSet :: BitSet
    startingPathSet = 0

    prod s t = [(a, b) | a <- s, b <- t]

    startingNeighborIndices = [0..(m - 1)] `prod` [0..(n - 1)]
        & map toIJ

    indices = [-1..1] `prod` [-1..1]
        & filter (/= (0, 0))

    neighborIndices :: IJ -> [IJ]
    neighborIndices ij = indices
        & map (\(!x, !y) -> (i + x, j + y))
        & filter (\(!i, !j) -> i >= 0 && i < m && j >= 0 && j < n)
        & map toIJ
      where
        (!i, !j) = fromIJ ij

    searchFrom :: PathDictElement -> [Path]
    searchFrom (!pathElem, !pathSet, !subDict) = ij
```

```haskell
searchFrom (!pathElem, !pathSet, !subDict) = ij
    & neighborIndices
    & searchIndices subDict pathSet
    & map (\(!path, !pathSet) -> (pathElem : path, pathSet)) -- TODO simplify
  where
    (!c, !ij) = fromPathElement pathElem

searchIndices :: Dict -> BitSet -> [IJ] -> [Path]
searchIndices subDict pathSet indices = indices
    & filter (not . (pathSet `testBit`))
    & toNeighbors
    & map searchNeighbor
    & parallelize
    & concat
  where
    -- only use parallelism when the subDict is large enough
    -- since a large subDict implies there's a lot more search work to do
    parallelize = case Dict.size subDict > 5000 of
        True -> (`using` parList rdeepseq)
        False -> id

    searchNeighbor :: PathElement -> [Path]
    searchNeighbor pathElem = currentPath ++ subPaths
      where
        (!c, !ij) = fromPathElement pathElem
        (found, maybeSubDict) = Dict.startingWith (BS.singleton c) subDict

        currentPath :: [Path]
        currentPath = found
            <&> const ([pathElem], pathSet)
            & maybeToList

        subPaths :: [Path]
        subPaths = maybeSubDict
            <&> (pathElem, pathSet `setBit` ij, )
            <&> searchFrom
            & fromMaybe []

toFoundWord :: Path -> FoundWord
toFoundWord (!combinedPath, !pathSet) = FoundWord {word, pathSet, path, score}
  where
    unPackedPath = combinedPath & map fromPathElement
    word = unPackedPath & map fst & BS.pack
    path = unPackedPath & map snd
    score = scorer $ BS.length word

solve :: Lang -> Solution
solve Lang {dict, dictSize} = Solution {words, totalScore, board_ = board}
  where
```

```haskell
        words = startingNeighborIndices
            & searchIndices dict startingPathSet
            & map toFoundWord
            & filter ((> 0) . score)
            & Set.fromList
            & Set.toList
            & sortBy cmp
        cmp = (comparing (BS.length . word)) `mappend` (comparing word)
        totalScore = words
            & map score
            & sum

new :: Board -> Bool -> Boggle
new = newWithScorer scorer
  where
    scorer n
        | n < 3 = 0
        | n == 3 = 1
        | n == 4 = 1
        | n == 5 = 2
        | n == 6 = 3
        | n == 7 = 5
        | n > 7 = 11
```

`Lang.hs`

```haskell
 withSizes :: (Int -> a -> b) -> Trie a -> (Trie b, Int)
withSizes map = f
  where
    f Empty = (Empty, 0)
    f (Arc k Nothing t) = (Arc k Nothing t', n)
      where
        (t', n) = f t
    f (Arc k (Just v) t) = (Arc k (Just $! map (n + 1) v) t', n + 1)
      where
        (t', n) = f t
    f (Branch p mask l r) = (Branch p mask l' r', m + n)
      where
        (l', m) = f l
        (r', n) = f r

ofSizes :: Trie a -> (Trie Int, Int)
ofSizes = withSizes (\n _ -> n)

type Dict = Trie Int

data Lang = Lang {
    dict :: Dict,
    dictSize :: Int,
    bytes :: ByteString
}

fromWords :: ByteString -> Lang
fromWords bytes = Lang {dict, dictSize, bytes}
  where
    (dict, dictSize) = bytes
        & CBS.split '\n'
        & map (, ())
        & Trie.fromList
        & ofSizes

fromFile :: FilePath -> IO Lang
fromFile path = mmapFileByteString path Nothing
    <&> fromWords

size :: Dict -> Int
size = f
  where
    f Empty = 0
    f (Arc _ Nothing t) = f t
    f (Arc _ (Just n) _) = n
    f (Branch _ _ l r) = (f l) + (f r)

startingWith :: ByteString -> Dict -> (Maybe (), Maybe Dict)
startingWith = Trie.lookupBy (\exists subDict -> (
        exists <&> const (),
        Just subDict & mfilter (not . Trie.null)
    ))
```