Computer Science

UDC 004.02

Oliynyk Olena

Senior Lecturer of Software Engineering Department

Kharkiv National University of Radioelectronics

Tisheninova Varvara

Student of the

Kharkiv National University of Radioelectronics

PARALLELISM AND CONCURRENCY IN HASKELL

Summary. *Comparison of computational complexity of parallel programs written on Haskell with standard modules Concurrent and Parallel.*

Key words: *parallel programming, Haskell, concurrency, threads, multithreading.*

The exponential growth in data processing and networking speeds means that parallel architecture is not just a good idea, it is necessary.

It is hard to remember complex software that relies on execution in a single thread, neglecting the ability to speed up. Today, all developers use parallelism or concurrency.

This article will discuss the effectiveness of parellism and concurrency in functional programming by the example of the Haskell programming language, and discuss the differences from the executions in other languages.

Haskell is a standardized linear functional programming language for general purposes, based on lambda calculus. The concept of the language is the idea of mathematician Haskell Carry, who wrote, "the proof is the program, and the formula to be produced is the type of program. It is in honour of H. Carry's

name was given to the language. His ideas became the basis for all functional programming languages.

Today, Haskell has become a much-demanded language in many areas. It is used for quick development of reliable, short and correct programs.

This article will describe the theoretical problem to be solved with multithreading or asynchrony. An installation guide and a link to the full code is also included.

1. CONCURRENCY AND PARALLELISM IN HASKELL

Haskell is a purely functional programming language, so it is much more interactive and interactive than other programming languages.

Haskell is often touted as the language for multi-core programming and easier parallelization than corresponding imperative programs.

Most languages do not distinguish between parallel computation and multithreading, providing a single toolkit for them. Haskell has a different set of tools and abstractions for each of these tasks.

A parallel program uses a multiplicity of computing units (processors, cores, GPUs, etc.) to speed up computation. We divide the work between simultaneously running computational units in the hope that the gains from parallelism will outweigh the extra cost of it. Accelerating computation is the only goal of parallelism.

Multithreading, by contrast, is the use of several control threads in a single program. These threads execute "concurrently" in the sense that the side effects of the threads interleave with each other. The goal of multithreaded programming is to maintain modularity when interacting simultaneously with multiple external agents (user, database, etc.). The number of processors in this case does not matter. Thus, parallelism requires multiple processors and multithreading requires side effects.

The notion of "control threads" is meaningless in the context of pure computation, because it has no side effects. Thus, multithreading only makes sense in the context of an IO monad. Moreover, multithreaded computation is fundamentally nondeterministic, because its task is to interact with external agents. Moreover, where there is nondeterminism, there is pain in debugging, testing, and support.

2. SETTING UP THE ENVIRONMENT

To start writing and run the code in the Haskell programming language, you need to install Haskell Platform and configure your IDE for it:

1. First of all, you need to download and install the Haskell Platform (http://www.haskell.org/platform/windows.html).

2. Then you need to install IDE. I recommend Sublime Text 3, which you can download on official website (http://www.sublimetext.com/3).

3. After this you need to install Haskell tools. Run in terminal:

```
cabal install cabal-install
cabal update
cabal install aeson
cabal install haskell-src-exts
cabal install ghc-mod
cabal install cmdargs
cabal install haddock
```

4. Then install hdevtools (https://github.com/mvoidex/hdevtools) and unpack it to some folder. After this go to that folder and run:

```
runhaskell Setup.hs configure --user
runhaskell Setup.hs build
runhaskell Setup.hs install
```

3. COMPARISONS OF CONCURANCY AND PARALLELISM

It is necessary to try in practice to understand the differences between multithreading and parallelism in Haskell. To do this, solve the problem of cracking the password hash using these two methods.

Formulation of the problem. Find the hash function prototype. To compare the performance of the multithreading and parallel programming

implement the solution of the corresponding problem with Control.Concurrent and Control.Parallel modules.

3.1 Bruteforce algorithm

First, we create a module which has two functions:

1. Hash converts string to sha256.
2. PasswordList generates a list of passwords of a given length, consisting of the specified letters.

The Haskell implementation is shown below:

```
module Bruteforce.Common where

import Data.Word
import Text.Printf
import Data.Digest.SHA2

passwordList :: String -> Int -> [String]
passwordList charList len =
    stream beginState
  where
    beginState = replicate len charList
    endState = replicate len [ last charList ]
    nextState ((_:[]):xs) = charList : nextState xs
    nextState ((_:ys):xs) = ys : xs
    nextState x = error $ "nextState " ++ show x
    stream st =
      let pw = map head st in
      if st == endState then [ pw ]
                        else pw : stream (nextState st)

hash :: String -> String
hash =
  concatMap (printf "%02x" :: Word8 -> String) .
    toOctets . sha256Ascii
```

3.2 Multithread implementation

Multithread implementation uses Control module and our custom Bruteforce module.

The getNumCapabilities function returns the number of operating system threads used by the runtime system. As will be shown below, this number can be set at program startup. In this case, it makes sense to use as many lightweight threads as real threads are used.

The program uses two queues. Actually, they are just lists, not queues, but I think it makes more sense to think of them as queues. The taskQueueue has the type MVar [String]. It contains tasks, which are prefixes of passwords to try out. The thread takes the prefix from the queue (the head of the list) and goes through all the passwords with that prefix. The resultQueue queue is of type MVar [[String]]. When the thread completes the task, it puts into this queue (also a list head) the list of strings it would like to display.

Finishing the main thread causes the whole program to terminate. Therefore, it must be terminated only after the child threads are finished. To stop the main thread from sitting idle, let us give it some work. This work is to clean up the resultQueue and display the results of the other threads' work. Strictly speaking, we have to do this because I/O is not thread safe in Haskell. At the same time the main thread counts how many tasks are currently left to do. When the task counter is zeroed, the main thread is terminated.

The implementation is shown below:

```
import Bruteforce.Common
import Control.Concurrent
import Control.Monad
import Control.DeepSeq

workerLoop :: MVar [String] -> MVar [ [String] ] -> String -> Int ->
                [String] -> IO ()
workerLoop taskQueue resultQueue charList pwLen hashList = do
  maybeTask <- modifyMVar taskQueue
                (\q -> return $ case q of
                                  [] -> ([], Nothing)
                                  (x:xs) -> (xs, Just x))
  case maybeTask of
    Nothing -> return ()
```

```
      Just task -> do
        let postfixList = passwordList charList $ pwLen - length task
            pwList = map (task ++) postfixList
            pwHashList = [(pw, hash pw) | pw <- pwList]
            rslt = [pw ++ ":" ++ h | (pw,h) <- pwHashList,
                                     h `elem` hashList]
        rslt `deepseq` modifyMVar_ resultQueue (\q -> return $ rslt:q)
        workerLoop taskQueue resultQueue charList pwLen hashList

mainLoop :: MVar [ [String] ] -> Int -> IO ()
mainLoop _ 0 = return ()
mainLoop resultQueue taskNumber = do
  results <- modifyMVar resultQueue (\q -> return ([], q))
  case results of
    [] -> do
      threadDelay 100000 -- 100 ms
      mainLoop resultQueue taskNumber
    _ -> do
      mapM_ (mapM_ putStrLn) results
      mainLoop resultQueue (taskNumber - length results)

main :: IO ()
main = do
  let hashList = [
        -- 1234
        "03ac674216f3e15c761ee1a5e255f067" ++
        "953623c8b388b4459e13f978d7c846f4",
        -- r2d2
        "8adce0a3431e8b11ef69e7f7765021d3" ++
        "ee0b70fff58e0480cadb4c468d78105f"
        ]
      pwLen = 4
      chunkLen = 2
      charList = ['0'..'9'] ++ ['a'..'z']
      taskList = passwordList charList chunkLen
      taskNumber = length taskList
  workerNumber <- getNumCapabilities
  taskQueue <- newMVar taskList
  resultQueue <- newMVar []
  workerNumber `replicateM_` forkIO (workerLoop taskQueue resultQueue
                                                 charList pwLen hashList)
```

3.3 Parallel implementation

A list of all possible passwords is generated on the fly; we do not store it or the hash list entirely in memory. Haskell's laziness has allowed us to achieve modularity (password generation and filtering are separated from each other; in languages with an applicative computation order, we would have to use iterators, generators).

Algorithmically partition the password space into sets of sufficiently large size, and then run the computation of the whole filterMatched in parallel for each such set. After that, we merge the results into one using concat.

First, we will generate prefixes of a certain length (the prefix length will control granularity), and to each prefix we will assign all possible residues. Passwords with a common prefix will be combined into sets, which will be processed in parallel.

The implementation is shown below:

```
import Bruteforce.Common( hash )
import Control.Monad( replicateM )
import Control.Parallel.Strategies( using, parList, rdeepseq )

type Hash = String
type Password = String
type CharList = String

-- | Sequential version of hash bruteforcing.
-- Given a list of candidate passwords, compute
-- hashes and compare to known ones. Return passwords (together with
-- its hashes) that can be found in the list of known hashes.
filterMatched :: [Hash]
              -> [Password]
              -> [(Password, Hash)]
filterMatched knownHashes candidates =
    filter (elemOf knownHashes . snd) pwHashList
  where hashes = map hash candidates
        pwHashList = zip candidates hashes
        elemOf = flip elem
```

```
-- | Parallel version of hash bruteforcing.
-- Split password space into chunks and apply
-- sequential bruteforce algorithm to each chunk
-- in parallel.
filterMatchedPar :: Int
                 -> CharList
                 -> [Hash]
                 -> [(Password, Hash)]
filterMatchedPar pwLen charList knownHashes = concat matched
  where prefLen = 1
        grouped = groupedPasswords pwLen prefLen charList
        matched = map (filterMatched knownHashes) grouped
                    `using` parList rdeepseq

-- | Generate passwords of given length, grouped by
-- common prefix (length of a prefix is provided as well).
-- This function is used for parallelization. Each spark
-- should process its own subset of password space, which
-- is generated lazily.
groupedPasswords :: Int -> Int -> CharList -> [[Password]]
groupedPasswords totalLen prefLen charList =
    map (prependPrefix postfixes) prefixes
  where prefixes = replicateM prefLen charList
        postfixes = replicateM restLen charList
        restLen = totalLen - prefLen
        prependPrefix post pre = map (pre ++) post

main :: IO ()
main = do
    let hashList = [
          -- 1234
          "03ac674216f3e15c761ee1a5e255f067" ++
          "953623c8b388b4459e13f978d7c846f4",
          -- r2d2
          "8adce0a3431e8b11ef69e7f7765021d3" ++
          "ee0b70fff58e0480cadb4c468d78105f"
          ]
        pwLen = 4
        charList = ['0'..'9'] ++ ['a'..'z']

        -- SEQUENTIAL VERSION USAGE EXAMPLE
```

```
        -- allPasswords = replicateM pwLen charList
        -- matched = filterMatched hashList allPasswords

        matched = filterMatchedPar pwLen charList hashList

    mapM_ (putStrLn . showMatch) matched

  where showMatch (pw, h) = pw ++ ":" ++ h
```

3.4 Results

After the above-described algorithm was implemented in one thread and multi threads, using two different modules.

The programs execution time was measured.

The following results were obtained:

Table 1

Bruteforce encryption execution time

One thread	Concurrency	Parallel
12.7 seconds	3.8 seconds	3.6 seconds

It can be seen from the table that the execution on one thread takes much more time than multithreads or parallel implementations.

Conclusions. After analyzing the results, we can say that parallel execution in Haskell significantly speeds up the program. With multiple threads, we lose determinism, but we gain time advantage, but with true Haskell parallelism, we gain runtime advantage without losing determinism of functions.

The implementation of parallelism and concurrency in the Haskell programming language are very different. It is very good practice to choose and use technologies depending on the task.

Full example code is available at https://github.com/Varvarya/sha256-bruteforce-haskell.

As a summary, we can say that Haskell is very well predisposed to parallel programming and is capable of solving both simple and complex problems that require fast execution.

References

1. Haskell official documentation. URL: https://www.haskell.org/documentation/ (date of the application 17.12.2021).
2. Marlow S. Parallel and concurrent programming in Haskell. Beijing: O'Reilly. 2013.