PARALLEL RUBIK'S CUBE SOLVER IN HASKELL

Yash Ashok Agarwal - ya2467
Chandrashekhar Dhulipala - cd3132

1 INTRODUCTION

The Rubik's cube is one of the most popular puzzles in the world. It is a 3D combination puzzle in which each of the six faces of the cube is colored with one of the six solid colors. The six faces are denoted as Front, Up, Right, Left, Down and Back. Each face can be rotated clockwise (U) or anti-clockwise (U'). Any sequence of moves to transform the cube from one configuration to another can be represented using this notation.

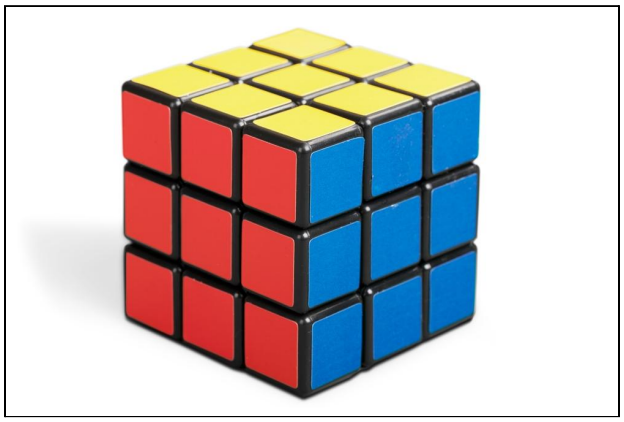

Figure1: Rubik Cube

In this project, we attempt to design and implement a parallel batch rubik's cube solver in Haskell. We use an algorithm known as the two-phase solver developed originally by Herbert Kociemba. We used a pre-existing implementation, refactored it to make it easy to parallelize and experimented with various parallel strategies. We compared the execution times and analyzed various factors that affect it. The subsequent sections describe our approach and results.

2 SEQUENTIAL ALGORITHM

2.1 Two Phase solver

Given the scrambled configuration of a cube in the form of a length-54 string (indicating the colors on each of the fifty-four faces), the two phase solver finds a solution to the configuration, which is a sequence of moves to transform the scrambled cube to a solved one. Example below -

Input : DDDFUDLRB FUFDLLLRR UBLBFDFUD ULBFRULLB RRRLBBRUB UBFFDFDRU

Output : U L B' L R2 D R U2 F U2 L2 B2 U B2 D' B2 U' R2 U L2 R2 U

This solver is implemented as a modular function in haskell, which for the rest of the report, is abstracted out as a method which takes as input one cube and outputs a string denoting the solution.

The first time this function is called on a system, it precomputes and stores some lookup tables which act as lower bounds on a heuristic function used in A* like search. This precomputation is done only once and does not factor into the subsequent runs of the program.

The actual solving procedure transpires in two phases. In the first phase, the algorithm looks for a sequence of moves to transform the scrambled cube into a particular subgroup called ‘G1’. This subgroup is a family of orientations that make use of only the moves <U,D,R2,L2,F2,B2> to go from one to the other. To search for a particular configuration in G1 reachable from the initial state, the algorithm makes use of Iterative-Deepening A* search which makes use of the heuristic function described above. After the algorithm finishes the first phase, the cube is in G1.

In the second phase, the algorithm makes use of only moves in G1 to transform the cube into a solved state. This makes use of standard mathematical properties of the G1 group and does not involve any search algorithm.

In the serial version of our code, we read the input file, which contains either 20,000 or 40,000 randomly permuted initial configurations of the cube and run the solve procedure described above sequentially on the list of cubes.

```
file <- readFile "test-40k.txt"
let cubes = lines file
     solutions = map (faceletList2 p) cubes
print(solutions)
```

This takes **863 seconds** for **20k cubes** and **1797 seconds** for **40k cubes.**

These numbers give us a fair idea of the sequential complexity of the algorithm and indicate that parallelism could help make the overall program much more efficient. The threadscope profile for the sequential program is shown below, and as expected the entire work is done by one core.

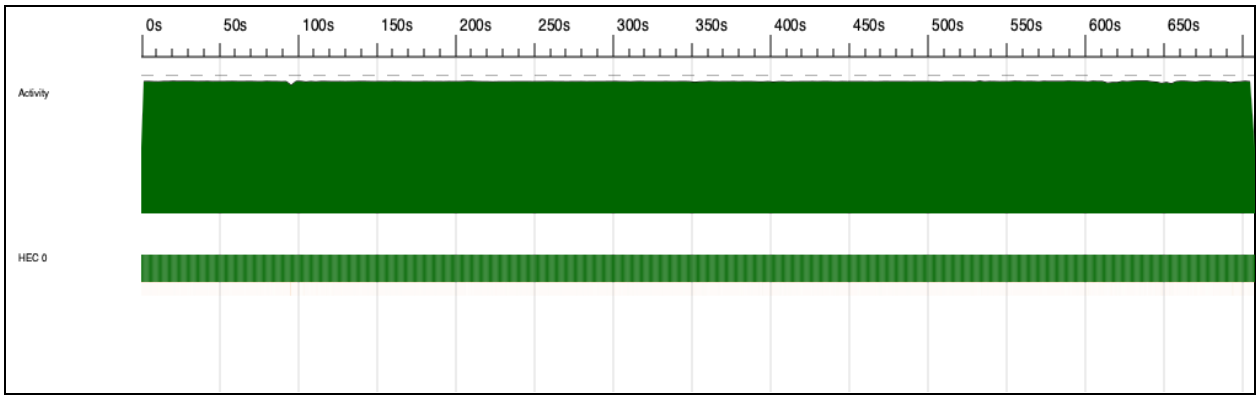

Figure 2: Threadscope profile for 20k cubes, serial execution

3 Parallel Implementation

3.1 AN ATTEMPT TO PARALLELIZE - `parMap`

Our first (probably naive) idea to parallelize this was to use the *parallel_map* operation to execute the solve procedure parallelly for all cubes in our huge input list. This scans the whole list and creates sparks (using rpar) for solving each cube in the list, which are then placed in a pool. If there are any processors available to take up new work, then they are supplied with pending sparks from the pool, which may be evaluated at any point in the future. So a spark here is similar to an incomplete result or a 'promise' of either evaluation (conversion) or failure (overflow/fizzle).

```
parallel_map::(a->b)->[a]->Eval [b]
parallel_map func [] = return []
parallel_map func (x:xs) = do
  b <- rpar (func x)
  bs <- parallel_map func xs
  return (b:bs)
```

```
solutions = runEval(parallel_map (faceletList2 p) cubes)
```

This approach is inspired by Simon Marlow's initial pointers on parallelizing a Sudoku solver. The `parallel_map` operation returns immediately after creating sparks for all the cubes, so we use the `runEval` operation from the Eval monad to actually evaluate the sparks before proceeding further. parMap in Control.Strategies has the same implementation as our paralle_map

While the performance of this on a list of 20,000 cubes is slightly promising, we observed that the performance of a parallel map on inputs as huge as 40,000 cubes caused the performance

to degrade compared to the serial version of the program. For 40,000 cubes, the trend of elapsed runtime is shown below.

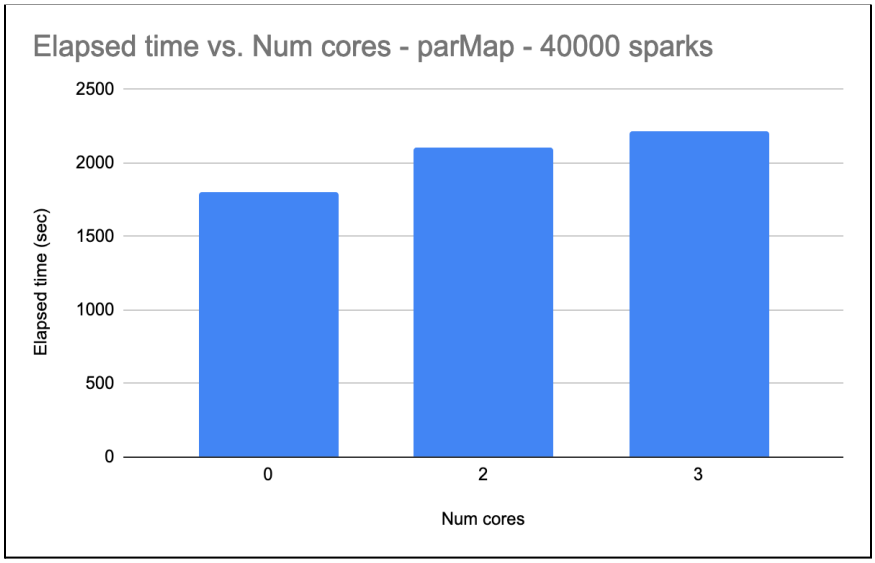

Figure 3: Elapsed time vs Number of cores on 40000 cubes
('0' cores here illustrates sequential version)

We can attribute this to the fact that an excessive number of sparks are created which eventually overflow or fizzle out.

```
SPARKS: 40000 (8193 converted, 31807 overflowed, 0 dud, 0 GC'd, 0
fizzled)

  INIT    time    0.002s  (  0.035s elapsed)
  MUT     time  1752.441s  (1868.417s elapsed)
  GC      time  440.915s  (240.919s elapsed)
  EXIT    time    0.005s  (  0.012s elapsed)
  Total   time  2193.531s  (2109.383s elapsed)
```

The above log is for 40,000 cubes run parallelly on 2 cores.

Another reason why this is not the best approach to parallelize is that it leads to a very imbalanced distribution of load between the N cores. This is evident from the threadscope profile below.

We see that HEC 0 is idle for a long time towards the tail of execution, while HEC 1 is busy evaluating sparks allotted to it.

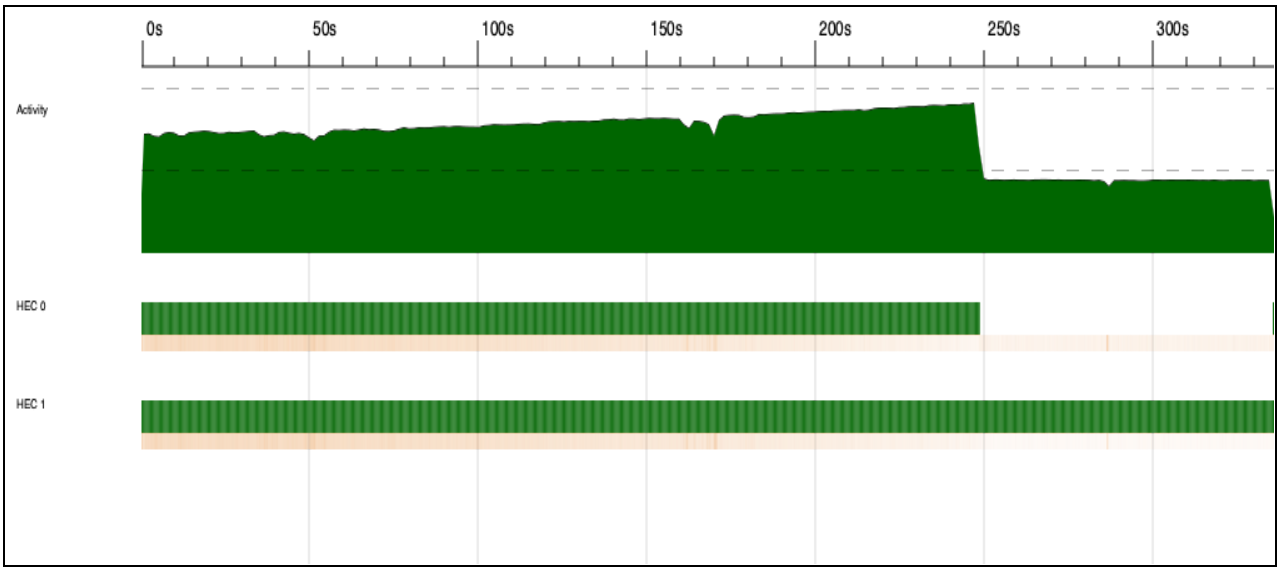

Figure 4:Threadscope profile for 20k cubes, Parallel execution with 2 cores

3.2 BETTER PARALLELIZATION - `parChunkList`

For larger inputs of size 40k and above, the parMap approach's execution time becomes more then the serial approach. This happens due to uneven load distribution. The task of solving one cube is a fine grain problem in terms of granularity. To solve this problem we used the parListChunk monad on the contents list of input.
By definition,

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

It divides the list into chunks and on each chunk it applies evalList strategy. Dividing the list into chunks and assigning each core a chunk ensure better workload distribution. Also, for each core the task is to solve an entire chunk which makes the task more coarser than before.

```
file <- readFile "test-40k.txt"
    let cubes = lines file
         solutions = map (faceletList2 p) cubes `using` parListChunk
1000 rdeepseq
```

The above code snippet from our code reads the input file containing the 40,000 cubes and maps over it calling the solver function faceletList2 specifying the strategy to be used as rdeepseq. rdeepseq forces Haskell to evaluate the result preventing weak head normal form creation. Dividing the input into chunks might also prevent spark overflow since for our machine we observed the maximum size of spark pool is 8192. This is because based on the chunk size each core is assigned a sublist of size N/C where N= number of total inputs and C=Chunk size and if N/C <8192 then sparks overflow will not occur.

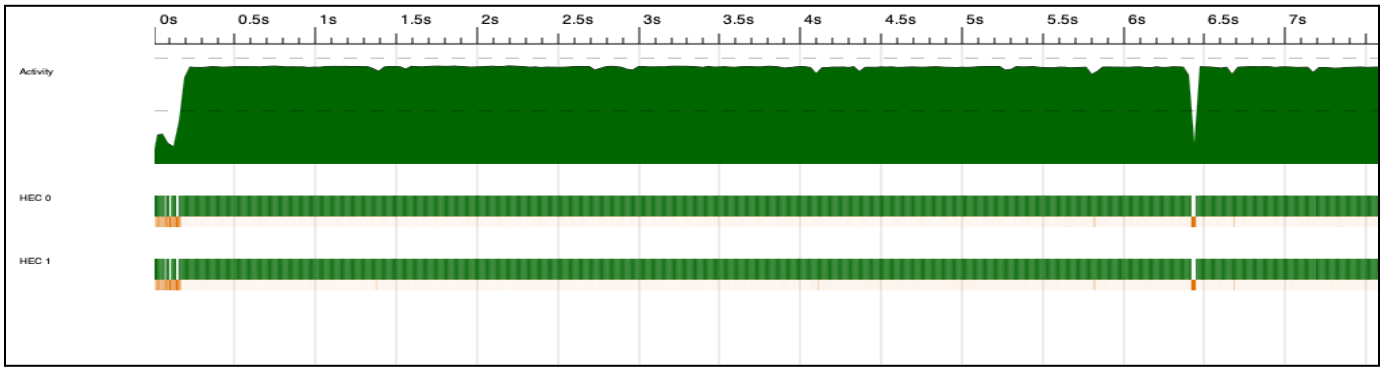

Figure 5: Threadscope profile for 20k cubes using 2 cores and 1000 chunks

In the above diagram we can see that there are instances where the Garbage Collector (GC) causes all the cores to halt until it collects the sparks which are not needed anymore. This is denoted by the dip in activity section and represented by the orange color in HEC 0 and HEC 1.

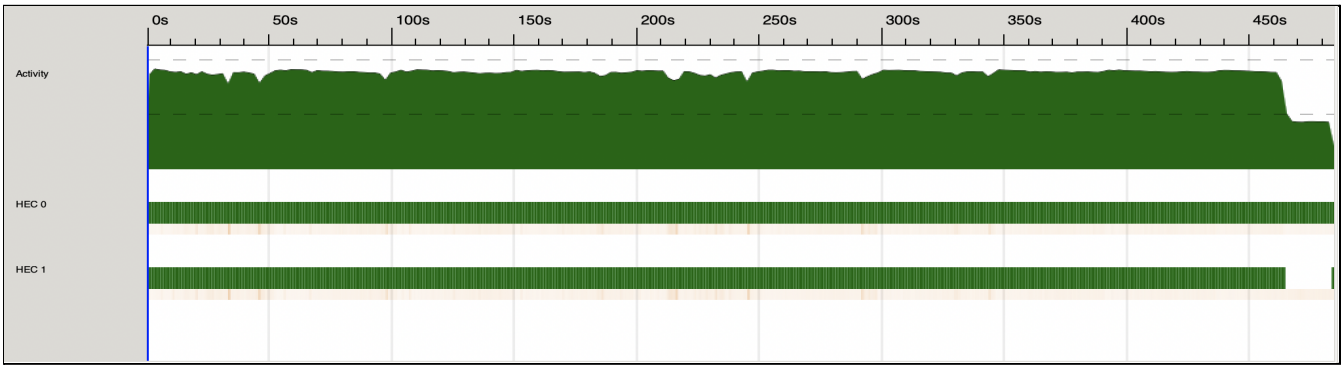

Figure 6: Threadscope profile for 20k cubes using 2 cores and 1000 chunks

As seen from the 2nd figure above there is some time for which the core sits idle. This idleness time is very less compared to the core idleness time in parMap implementation.

4 Results

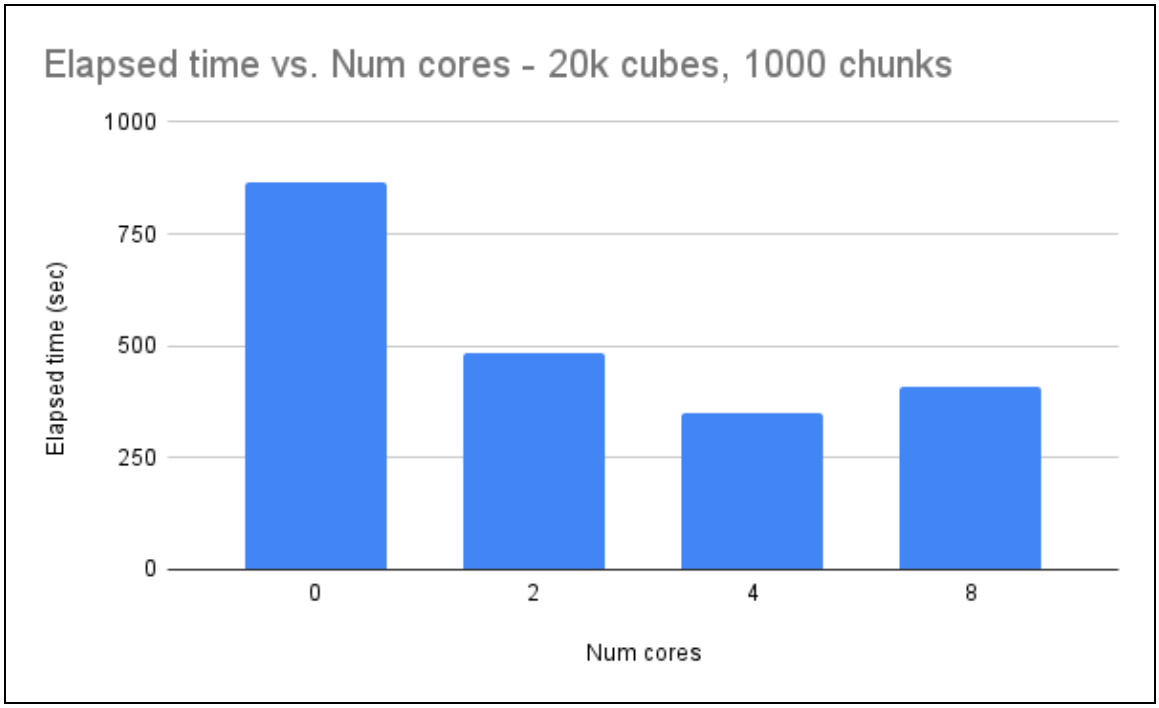

Figure 7: Graph of Elapsed time versus Number of Cores 20k

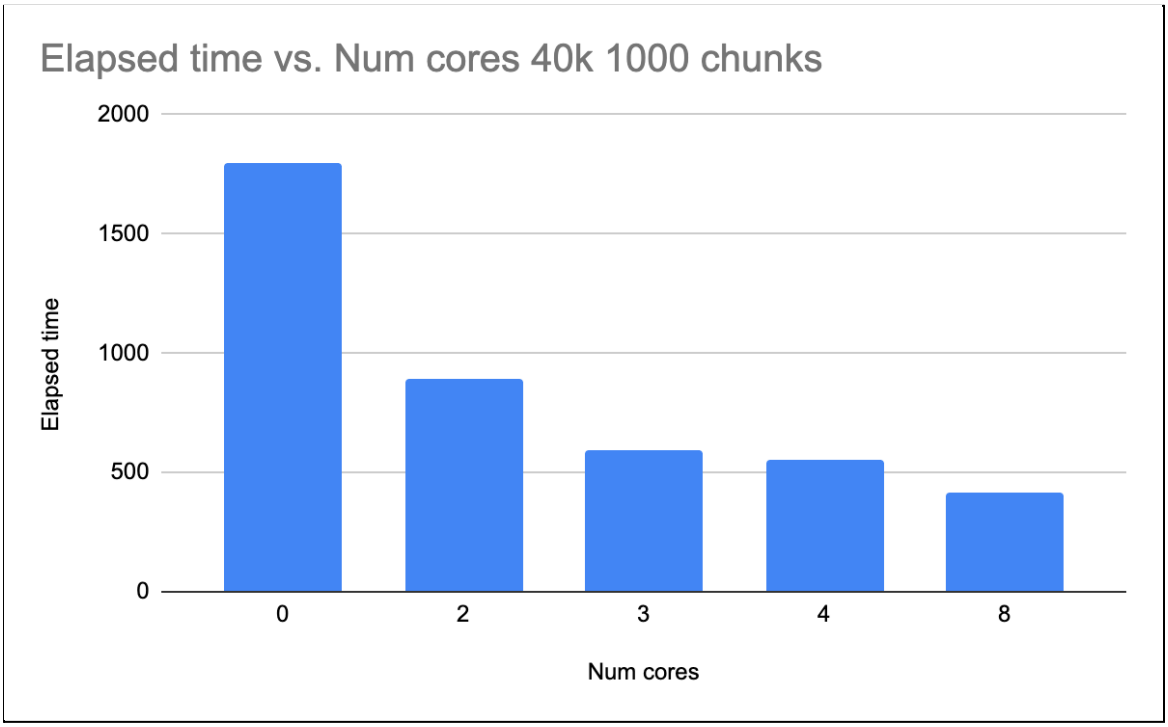

Figure 8: Graph of Elapsed time versus Number of Cores 40k

4.1 EFFECT OF NUMBER OF CORES

To observe the effect of the number of cores we kept the number of chunks as 1000(constant) and used an input of 40,000 cubes.

Cores	Execution Time(1000 chunks)	Speedup
2	893.379	1.98
3	596.077	2.86
4	549.844	3.26
8	417.049	4.31

Table 1: Effect of Number of Cores

4.2 EFFECT OF CHUNK SIZE

We have seen that the performance of `parListChunk` on the huge input files is very encouraging. The results enumerated above are for a constant (1000) number of chunks into which the input list is divided. This inspired us to fix the number of cores and vary the number of chunks to find the optimal parameters for maximum speedup.

As we observed that 8 cores is performing well for input of size 40,000 cubes, we fixed the number of cores at 8 and measured the elapsed time with various chunks. The trend is shown below -

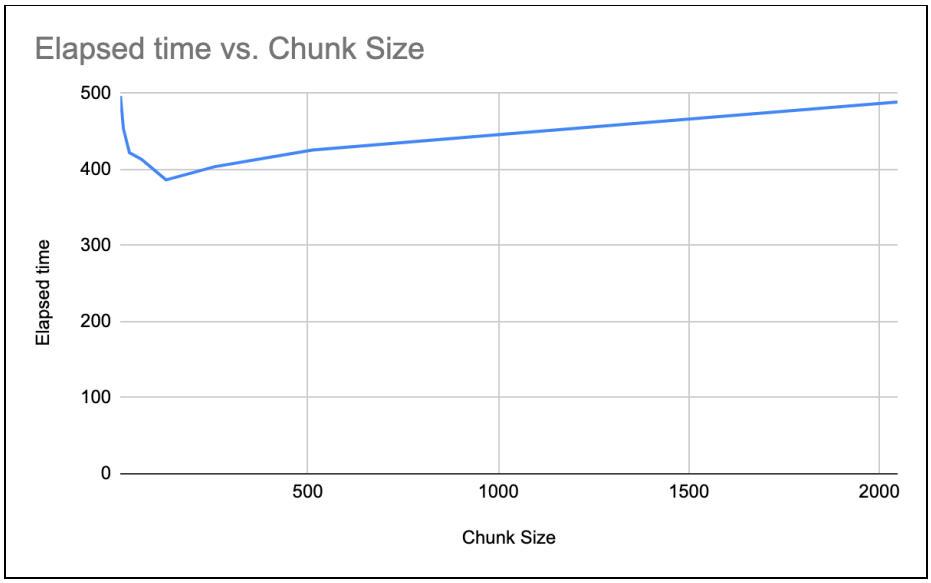

Figure 9: Graph of Elapsed time versus Chunk Size

The best performance is observed for 128 chunks - 386.4 seconds.

Speedup = 1797 / 386.4 = **4.65** as compared with the serial program for 40,000 cubes.

```
TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 313 (313 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.002s  (  0.055s elapsed)
MUT     time  897.018s  (317.333s elapsed)
GC      time  1682.139s  ( 69.014s elapsed)
EXIT    time    0.002s  (  0.004s elapsed)
Total   time  2579.338s  (386.406s elapsed)
```

4.3 EFFECT OF SYSTEM SPECIFICATIONS:

We tested all the above results on systems with different configurations like Macbook Pro(M1-chip 8 core) Macbook air(Intel chip 4 core).

The variation in results due to system configuration in both the mentioned devices was negligible.

5 FUTURE WORK

We explored different ways to parallelize the serial implementation. Adding the functionality of solving higher dimensional cubes apart from 3x3 in future. Another improvement area that can be explored in the future is parallelizing the Iterative-Deepening A* search algorithm's iteration. This might help for solving cubes with higher dimensionality since precomputing the states for all higher order cubes will not be optimal.

6 CONCLUSION

In conclusion, we achieved a 4.65 speedup by dividing the input list into 128 chunks and parallelizing them using 8 cores. We explored multiple approaches to parallelize the serial rubik's cube solver and identified the factors that govern their execution time.

7 REFERENCES

1. https://github.com/Lysxia/twentyseven
2. http://kociemba.org/cube.htm
3. https://hackage.haskell.org/package/threadscope
4. **Parallel and Concurrent Programming in Haskell, Simon Marlow**

8 CODE

twentyseven.hs

```
{-# LANGUAGE NamedFieldPuns, RecordWildCards #-}

import Rubik.Cube
import Rubik.Misc
import qualified Rubik.Solver.Optimal as Optimal
import qualified Rubik.Solver.TwoPhase as TwoPhase
import qualified Rubik.Tables.Internal as Option

import Control.Exception
import Control.Monad

import Data.Time.Clock
```

```
import Data.Char
import Data.Monoid

import Numeric ( showFFloat )

import Options.Applicative hiding ( value )
import qualified Options.Applicative as Opt

import System.Exit
import System.IO.Error

import
Control.Parallel.Strategies(Eval,rpar,runEval,using,parListChunk
,rdeepseq)

type Solver = Cube -> Move

data Parameters = Parameters {
    verbose :: Bool,
    solve :: Solver,
    tsPath :: Maybe FilePath,
    precompute :: Bool,
    overwrite :: Bool,
    noFiles :: Bool,
    strict :: Bool,
    debug :: Bool
  }

optparse :: Parser Parameters
optparse = Parameters
  <$> switch ( long "verbose" <> short 'v'
        <> help "Print time taken to solve every cube" )
  <*> flag TwoPhase.solve Optimal.solve ( long "optimal"
        <> help "Use optimal solver (experimental)" )
  <*> (optional . strOption) ( long "ts-dir" <> short 'd'
        <> metavar "DIR"
        <> help "Location of precomputed tables" )
  <*> switch ( long "precompute" <> short 'p'
        <> help "Precompute and store tables \
                \(do enable this at the first invocation)" )
  <*> switch ( long "overwrite"
```

```
        <> help "Recompute and overwrite tables even when they
exist already" )
  <*> switch ( long "no-files"
        <> help "Do not read or write any files \
                \(recompute tables for this session)" )
  <*> switch ( long "strict"
        <> help "Force loading tables before doing anything
else" )
  <*> switch ( long "debug" )


parallel_map::(a->b)->[a]->Eval [b]
parallel_map func [] =return []
parallel_map func (x:xs) = do
  b<- rpar (func x)
  bs <- parallel_map func xs
  return (b:bs)


main :: IO ()
main = do
    p <- execParser $ info (helper <*> optparse) briefDesc
    setOptions p
    file <- readFile "test-40k.txt"
    let cubes = lines file
        --solutions = runEval(parallel_map (faceletList2 p)
cubes)
        solutions = map (faceletList2 p) cubes `using`
parListChunk 1000 rdeepseq
        --solutions = map (faceletList2 p) cubes
    print(solutions)
    return ()

setOptions :: Parameters -> IO ()
setOptions Parameters{..} = do
  mapM_ Option.setTsPath tsPath
  Option.setPrecompute precompute
  Option.setOverwrite overwrite
  Option.setNoFiles noFiles
  Option.setDebug debug
  when strict . void $ evaluate
```

```
    (solve . either undefined moveToCube . stringToMove $
"ulfrbd")

-- A sequence of moves, e.g., "URF".
moveSequence s = putStrLn $
  case stringToMove s of
    Left c -> "Unexpected '" ++ [c] ++ "'."
    Right ms -> stringOfCubeColors . moveToCube . reduceMove $
ms

faceletList2 p s = case readCube (filter (not . isSpace) s) of
    Left err -> show err
    Right cube -> justSolve2 p cube

readCube s
  = case colorFacelets'' s of
      Nothing -> Left "Expected string of length 54 of a set of
(any) 6 \
                      \characters. Centers must be distinct."
      Just colors ->
        case colorFaceletsToCube colors of
          Left fs ->
            Left $ "Facelets " ++ show fs
                ++ " (" ++ show (map (s !!) fs) ++ ") \
                   \do not match any regular cubie."
          Right Nothing ->
            Left "Not a permutation of cubies \
                 \(a cubie is absent, and a cubie occurs
twice)."
          Right (Just c) | solvable c -> Right c
          _ -> Left "Unsolvable cube."

justSolve2 :: Parameters -> Cube -> String
justSolve2 p c = moveToString (solve p c)

unlessQuiet' :: IO () -> Parameters -> IO ()
unlessQuiet' a = unlessQuiet (const a) ()

-- Strict in its second argument
unlessQuiet :: (a -> IO ()) -> a -> Parameters -> IO ()
unlessQuiet f a p = evaluate a >> when (verbose p) (f a)
```

```
clock :: IO a -> IO Double
clock a = do
  t <- getCurrentTime
  a
  t' <- getCurrentTime
  return (diffTimeToSeconds (diffUTCTime t' t))
  where
    diffTimeToSeconds = fromRational . toRational

listSeq' :: [a] -> [a]
listSeq' s = s `listSeq` s

vPutStrLn :: String -> Parameters -> IO ()
vPutStrLn s = unlessQuiet putStrLn (listSeq' s)

vPutStr :: String -> Parameters -> IO ()
vPutStr s = unlessQuiet putStrLn (listSeq' s)
```

tstables.hs

```
import TwoPhase

import Control.Applicative

import System.Directory
import System.Environment
import System.Exit
import System.FilePath
import System.IO

main :: IO ()
main = do
  args <- getArgs
  path <- case args of
        [] -> (</> ".tseven") <$> getHomeDirectory
        p : _ -> return p
  createDirectoryIfMissing
    True -- createParents
    path
```

```
  let p1 = path </> "phase1"
      p2 = path </> "phase2"
  fileExists <- and <$> mapM doesFileExist [p1,p2]
  case fileExists of
    True -> do
      hPutStrLn stderr $ "File(s) already exist(s) in '" ++ path
++"'."
      exitFailure
    False -> do
      putStrLn "Phase 1"
      encodeFile p1 phase1Compressed'
      putStrLn "Phase 2"
      encodeFile p2 phase2Compressed'
      exitSuccess
```

Solver.hs

```
{-# LANGUAGE ScopedTypeVariables, RecordWildCards, TypeFamilies,
TypeOperators,
    ViewPatterns #-}
module Rubik.Solver where

import Rubik.Cube
import Rubik.IDA
import Rubik.Misc
import Rubik.Symmetry

import Control.Applicative

import Data.Coerce
import Data.Foldable
import Data.Int (Int8)
import Data.Maybe
import Data.Tuple.Extra
import qualified Data.Vector as V
import qualified Data.Vector.Storable.Allocated as S

type MaybeFace = Int
type SubIndex = Int
type DInt = Int8
```

```
data Projection x a0 as a = Projection
  { convertP :: x -> a
  , isIdenP :: a -> Bool
  , indexP :: as -> a -> a
  , subIndexSize :: Int
  , unfoldP :: a0 -> SubIndex -> [as]
  , subIndexP :: a -> SubIndex
  }

type Projection' m a = Projection Cube (MoveTag m [RawMove a])
(RawMove a) (RawCoord a)
type SymProjection m sym a = Projection Cube (MoveTag m [SymMove
sym a]) (SymMove sym a) (SymCoord sym a)

newtype Distance m a = Distance { distanceP :: a -> DInt }

infixr 4 |*|, |.|

{-# INLINE (|*|) #-}
(|*|) :: (TupleCons b0, TupleCons bs, TupleCons b)
  => Projection x a0 as a
  -> Projection x b0 bs b
  -> Projection x (a0 :| b0) (as :| bs) (a :| b)
a |*| b = Projection
  { convertP = liftA2 (|:|) (convertP a) (convertP b)
  , isIdenP = \(split -> (a_, b_)) -> isIdenP a a_ && isIdenP b
b_
  , indexP = \(split -> (as_, bs_)) (split -> (a_, b_)) ->
indexP a as_ a_ |:| indexP b bs_ b_
  , subIndexSize = subIndexSize a * subIndexSize b
  , unfoldP = \(split -> (a0_, b0_)) ci ->
      let (ai, bi) = ci `divMod` subIndexSize b
      in zipWith (|:|) (unfoldP a a0_ ai) (unfoldP b b0_ bi)
  , subIndexP = \(split -> (a_, b_)) -> flatIndex (subIndexSize
b) (subIndexP a a_) (subIndexP b b_) }

{-# INLINE (|.|) #-}
(|.|) :: forall x a0 as a b0 bs b
  . Projection x a0 as a
  -> Projection x b0 bs b
  -> Projection x (a0, b0) (as, bs) (a, b)
```

```
a |.| b = a |*| (coerce b :: Projection x (Tuple1 b0) (Tuple1
bs) (Tuple1 b))

{-# INLINE (>$<) #-}
(>$<) :: forall m a b. (b -> a) -> Distance m a -> Distance m b
(>$<) = coerce (flip (.) :: (b -> a) -> (a -> DInt) -> (b ->
DInt))

{-# INLINE maxDistance #-}
maxDistance :: forall f m a. Foldable f => f (Distance m a) ->
Distance m a
maxDistance = foldl' (\(Distance f) (Distance g) -> Distance $
\x -> max (f x) (g x)) (Distance $ const 0)

-- | ==Branching reduction
--
-- The @Int@ projection keeps track of the latest move (@== 6@
-- for the starting point).
--
-- 18 moves
--
-- We can indeed reduce the branching factor from 18 to 15
-- by considering that successive moves on the same face
-- can and will be shortened as a single move.
--
-- Furthermore, since moves on opposite faces commute, we may
force
-- them to be in an arbitrary order, reducing the branching
factor
-- to 12 after half of the moves (U, L, F).
--
-- 10 moves
--
-- Instead of a factor 10, we have factors
--
-- - 9 after R, B;
-- - 8 after L, F;
-- - 7 after D;
-- - 4 after U.

{-# INLINE solveWith #-}
```

```
solveWith
  :: Eq a
  => MoveTag m [ElemMove] -> a0
  -> Projection Cube a0 as a
  -> Distance m a
  -> Cube -> Move
solveWith (MoveTag moveNames) ms ps pd
  = fromJust . search Search{..} . tag . convertP ps
  where
    goal = isIdenP ps . snd
    estm = distanceP pd . snd
    edges (i, t)
      = fmap
          (\(l, succs, j') ->
            let x = indexP ps succs t in Succ l 1 (j', x))
          (succVector V.! (subIndexP ps t * 7 + i))
    -- For every move, filter out "larger" moves for an
arbitrary total order of faces
    succVector = V.fromList $ do
      subi <- [0 .. subIndexSize ps - 1]
      let as = unfoldP ps ms subi
      i' <- [0 .. 6]
      return
        [ (l, m, fromEnum j)
        | (l@(_, j), m) <- zip moveNames as
        , i' == 6 || (let i = toEnum i' in not (i == j ||
oppositeAndGT j i)) ]

type Tag a = (Int, a)

tag :: a -> Tag a
tag = (,) 6

{-# INLINE rawProjection #-}
rawProjection :: (FromCube a, RawEncodable a) => Projection' m a
rawProjection = Projection
  { convertP = convert
  , isIdenP = (== convert iden)
  , indexP = (!$)
  , subIndexSize = 1
  , unfoldP = \(MoveTag as) _ -> as
```

```
  , subIndexP = \_ -> 0
  }
  where
    convert = encode . fromCube

{-# INLINE symProjection #-}
symProjection :: (FromCube a, RawEncodable a)
  => (a -> SymCoord sym a) -> SymProjection m sym a
symProjection convert = Projection
  { convertP = convert'
  , isIdenP = let (x0, _) = convert' iden in \(x, _) -> x == x0
  , indexP = symMove' 16
  , subIndexSize = 16
  , unfoldP = \(MoveTag as) i -> [ as !! j | j <- symAsMovePerm
(sym16 !! i) ]
  , subIndexP = \(_, SymCode i) -> i
  }
  where
    convert' = convert . fromCube

-- TODO newtype this
{-# INLINE symmetricProj #-}
symmetricProj :: Eq c => Symmetry sym
  -> Projection Cube (MoveTag m [b]) as c
  -> Projection Cube (MoveTag m [b]) as c
symmetricProj sym proj = proj
  { convertP = convert
  , unfoldP = \as i -> rawMoveSym sym (unfoldP proj as i)
  }
  where
    convert = convertP proj . conjugate (inverse (symAsCube
sym))

{-# INLINE distanceWith2 #-}
distanceWith2
  :: (RawEncodable a, RawEncodable b)
  =>  S.Vector DInt -> Distance m (RawCoord a, RawCoord b)
distanceWith2 v = Distance $ \(RawCoord a_, b@(RawCoord b_)) ->
  v S.! flatIndex (range b) a_ b_
```

Symmetry.hs

```
{- |
 - Tables of symmetry classes
 -}
{-# Language GeneralizedNewtypeDeriving, ScopedTypeVariables,
ViewPatterns #-}
module Rubik.Symmetry where

import Rubik.Cube
import Rubik.Misc

import Control.DeepSeq
import Control.Monad

import Data.Binary.Storable
import Data.Foldable
import Data.List
import Data.Maybe
import Data.Ord
import qualified Data.Heap as H
import qualified Data.Vector as V
import qualified Data.Vector.Storable.Allocated as S

-- | Smallest representative of a symmetry class.
-- (An element of the symClasses table)
type SymRepr a = RawCoord a

type SymClass' = Int
-- | Symmetry class. (Index of the smallest representative in
the symClasses table)
newtype SymClass symType a = SymClass { unSymClass :: SymClass'
}
  deriving (Eq, Ord, Show)

type SymCoord sym a = (SymClass sym a, SymCode sym)

-- | An @Int@ representing a pair @(Repr, Sym)@.
--
-- If @x = symClass * symOrder + symCode@,
```

```
-- where @symClass :: SymClass@ is the index of the symmetry
class with
-- smallest representative @r :: SymRepr@ (for an arbitrary
order relation),
-- @symOrder@ is the size of the symmetry group,
-- @symCode :: Sym@ is the index of a symmetry @s@;
-- then @s^(-1) <> r <> s@ is the value represented by @x@.
type SymCoord' = Int
type SymOrder' = Int

newtype Action s a = Action [a -> a]
newtype SymClassTable s a = SymClassTable { unSymClassTable ::
S.Vector RawCoord' }
  deriving (Eq, Ord, Show, Binary, NFData)
newtype SymReprTable s a = SymReprTable { unSymReprTable ::
S.Vector Int }
  deriving (Eq, Ord, Show, Binary, NFData)
newtype SymMove s a = SymMove (S.Vector SymCoord')
  deriving (Eq, Ord, Show, Binary, NFData)

type Symmetries sym a = MoveTag sym (V.Vector (RawMove a))

-- | Compute the table of smallest representatives for all
symmetry classes.
-- The @RawCoord'@ coordinate of that representative is a
@Repr@.
-- The table is sorted in increasing order.
symClasses
  :: RawEncodable a
  => Action s a    {- ^ Symmetry group, including the identity,
                    -   represented by its action on @a@ -}
  -> SymClassTable s a {- ^ Smallest representative -}
symClasses = SymClassTable . S.fromList . fmap unRawCoord .
symClasses'

symClasses' :: forall a s. RawEncodable a => Action s a ->
[RawCoord a]
symClasses' action@(Action sym)
  = foldFilter (H.empty :: H.MinHeap (RawCoord a))
      (fmap RawCoord [0 .. range action - 1])
  where
```

```
    foldFilter _ [] = []
    foldFilter (H.view -> Nothing) (x : xs) = x : foldFilter
(heapOf x) xs
    foldFilter (h@(H.view -> Just (y, ys))) (x : xs)
      | x < y = x : foldFilter (H.union h (heapOf x)) xs
      | otherwise = foldFilter ys xs
    heapOf :: RawCoord a -> H.MinHeap (RawCoord a)
    heapOf x
      = let dx = decode x
            nub' = map head . group . sort
        in H.fromAscList . tail . nub' $ map (\z -> (encode . z)
dx) sym

symClassTable
  :: Int
  -> SymReprTable s a
  -> SymClassTable s a
symClassTable nSym (SymReprTable s)
  = SymClassTable . S.ifilter (==) $ S.map (`div` nSym) s

symReprTable
  :: forall a s t. (RawEncodable a, Foldable t)
  => Int -- ^ Number of symmetries @nSym@
  -> (RawCoord a -> t (RawCoord a))
  -> SymReprTable s a
symReprTable nSym f
  = SymReprTable (symReprTable' (range ([] :: [a])) nSym f')
  where
    f' = fmap unRawCoord . toList . f . RawCoord

{-# INLINE symReprTable' #-}
symReprTable'
  :: Foldable t
  => Int -- ^ Number of elements @n@
  -> Int -- ^ Number of symmetries @nSym@
  -> (Int -> t Int) -- ^ @f x@, symmetrical elements to @x@,
including itself
  -> S.Vector Int
  -- ^ @v@, where @(y, i) = (v ! x) `divMod` nSym@ gives
  -- the representative @y@ of the symmetry class of @x@
  -- and the index of one symmetry mapping @x@ to @y@:
```

```
  --
  -- > f x !! i == y.
symReprTable' n nSym f
  = S.create $ do
      v <- S.replicate n (-1)
      forM_ [0 .. n-1] $ \x -> do
        let ys = f x
        y <- S.read v x
        when (y == -1) .
          forM_ ((zip [0 ..] . toList . f) x) $ \(i, x') ->
            S.write v x' (flatIndex nSym x i)
      return v

-- |
symMoveTable
  :: RawEncodable a
  => Action s a      {- ^ Symmetry group -}
  -> SymClassTable s a   {- ^ (Sorted) table of representatives
-}
  -> (a -> a)        {- ^ Endofunction to encode -}
  -> SymMove s a
symMoveTable action@(Action syms) classes f
  = SymMove (S.map move (unSymClassTable classes))
  where
    n = length syms
    move = flat . symCoord action classes . f . decode .
RawCoord
    flat (SymClass c, SymCode s) = flatIndex n c s

symMoveTable'
  :: RawEncodable a
  => Int -- ^ Symmetry group order
  -> SymReprTable s a
  -> SymClassTable s a
  -> (a -> a)
  -> SymMove s a
symMoveTable' nSym reps classes f
  = SymMove (S.map move (unSymClassTable classes))
  where
    move = flat . symCoord' nSym reps classes . encode . f .
decode . RawCoord
```

```
    flat (SymClass c, SymCode s) = flatIndex nSym c s

{-# INLINE symMove #-}
symMove :: SymOrder' -> SymMove s a -> SymClass s a -> SymCoord
s a
symMove n (SymMove v) (SymClass x) = (SymClass y, SymCode i)
  where (y, i) = (v S.! x) `divMod` n

{-# INLINE symMove' #-}
symMove' n v (x, j) = (y, i `composeSym` j)
  where (y, i) = symMove n v x

reprToClass :: SymClassTable s a -> RawCoord a -> SymClass s a
reprToClass (SymClassTable cls) = SymClass . fromJust . flip
iFind cls . unRawCoord

-- | Find the representative as the one corresponding to the
smallest coordinate
symCoord :: RawEncodable a => Action s a -> SymClassTable s a
  -> a -> SymCoord s a
symCoord (Action syms) classes x
  = (reprToClass classes r, SymCode s)
  where
    xSym = [ encode (s x) | s <- syms ]
    (r, s) = minimumBy (comparing fst) (zip xSym [0 ..])

symCoord' :: Int -> SymReprTable s a -> SymClassTable s a ->
RawCoord a -> SymCoord s a
symCoord' nSym (SymReprTable reps) (SymClassTable classes)
(RawCoord x)
  = (SymClass r, SymCode i)
  where
    (y, i) = (reps S.! x) `divMod` nSym
    r = fromJust $ iFind r classes

symToRaw
  :: SymClassTable s a -> (RawCoord a -> SymCode s -> RawCoord
a)
  -> SymCoord s a -> RawCoord a
symToRaw (SymClassTable classes) sym (SymClass c, i)
  = sym (RawCoord (classes S.! c)) i
```

```
sym :: Symmetries s a -> RawCoord a -> SymCode s -> RawCoord a
sym (MoveTag syms) r (SymCode i) = syms V.! i !$ r
```

TwoPhase.hs

```
{- | Two phase algorithm to solve a Rubik's cube -}

{-# LANGUAGE RecordWildCards, ViewPatterns #-}
module Rubik.Solver.TwoPhase where

import Rubik.Cube
import Rubik.Misc
import Rubik.Solver
import Rubik.Tables.Moves
import Rubik.Tables.Distances

import Data.Function ( on )
import Data.Monoid

{-# INLINE phase1Proj #-}
phase1Proj
  =   rawProjection
  |*| rawProjection
  |.| rawProjection

phase1Convert = convertP phase1Proj

phase1Dist = maxDistance
  [ (\((,,) co _ uds) -> (co, uds)) >$< distanceWith2
d_CornerOrien_UDSlice
  , (\((,,) _ eo uds) -> (eo, uds)) >$< distanceWith2
d_EdgeOrien_UDSlice
  ]

phase1 :: Cube -> Move
phase1 = solveWith move18Names moves phase1Proj phase1Dist
  where
    moves = (,,) move18CornerOrien move18EdgeOrien move18UDSlice
```

```
-- | > phase1Solved (phase1 c)
phase1Solved :: Cube -> Bool
phase1Solved = ((==) `on` phase1Convert) iden

--

phase2Proj
  =   rawProjection
  |*| rawProjection
  |.| rawProjection

phase2Convert = convertP phase2Proj

phase2Dist = maxDistance
  [ (\((,,) cp _ udsp) -> (cp, udsp)) >$< distanceWith2
d_CornerPermu_UDSlicePermu2
  , (\((,,) _ udep udsp) -> (udep, udsp)) >$< distanceWith2
d_UDEdgePermu2_UDSlicePermu2
  ]

phase2 :: Cube -> Move
phase2 = solveWith move10Names moves phase2Proj phase2Dist
  where
    moves = (,,) move10CornerPermu move10UDEdgePermu2
move10UDSlicePermu2

-- | > phase1Solved c ==> phase2Solved (phase2 c)
phase2Solved :: Cube -> Bool
phase2Solved = (== iden)

-- | Solve a scrambled Rubik's cube.
--
-- Make sure the cube is actually solvable with
'Cubie.solvable',
-- before calling this function.
solve :: Cube -> Move
solve c =
  let s1 = phase1 c
      c1 = c <> moveToCube s1
      s2 = phase2 c1
  in reduceMove $ s1 ++ s2
```

Optimal.hs

```
module Rubik.Solver.Optimal where

import Rubik.Cube
import Rubik.Solver
import Rubik.Tables.Moves
import Rubik.Tables.Distances
import Rubik.Tables.Internal

import qualified Data.Vector.Generic as G

{-# INLINE optiProj #-}
optiProj
  = fudsp |*| sfudsp |*| s sfudsp |*| co |*| sco |*| s sco |.|
cp
  where
    fudsp = symProjFlipUDSlicePermu
    sfudsp = s fudsp
    co = rawProjection :: Projection' Move18 CornerOrien
    sco = s co
    cp = symProjCornerPermu
    s x = symmetricProj symmetry_urf3 x

{-# INLINE optiDist #-}
optiDist = maxDistance
  [ maxOrEqualPlusOne
      ( (\((,,,,,,) fudsp _ _ co _ _ _) -> (fudsp, co)) >$<
fudsp_co
      , (\((,,,,,,) _ fudsp _ _ co _ _) -> (fudsp, co)) >$<
fudsp_co
      , (\((,,,,,,) _ _ fudsp _ _ co _) -> (fudsp, co)) >$<
fudsp_co
      )
  , (\((,,,,,,) _ _ _ co _ _ cp) -> (cp, co)) >$< cp_co
  ]

{-# INLINE maxOrEqualPlusOne #-}
maxOrEqualPlusOne (Distance f, Distance g, Distance h)
```

```
  = Distance $ \x -> let a = f x ; b = g x ; c = h x
    in if a == b && b == c && a /= 0 then a + 1
      else a `max` b `max` c

solve :: Cube -> Move
solve = solveWith move18Names moves optiProj optiDist
  where
    moves = (,,,,,,) m_fudsp m_fudsp m_fudsp m_co m_co m_co
move18SymCornerPermu
    m_fudsp = move18SymFlipUDSlicePermu
    m_co = move18CornerOrien

{-# INLINE toIdx #-}
toIdx = uncurry $ indexWithSym invertedSym16CornerOrien (range
([] :: [CornerOrien]))

{-# INLINE fudsp_co #-}
fudsp_co = toIdx >$< Distance (fromIntegral .
(dSym_CornerOrien_FlipUDSlicePermu G.!))
{-# INLINE cp_co #-}
cp_co = toIdx >$< Distance (dSym_CornerOrien_CornerPermu G.!)
```

Cubie.hs

```
{- |
   Cubie representation.

   A Rubik's cube is the cartesian product of a permutation of
cubies
   and an action on their orientations.
-}

module Rubik.Cube.Cubie (
  -- * Complete cube
  CubeAction (..),
  FromCube (..),
  Cube (..),

  -- ** Solvability test
  solvable,
```

```
-- * Corners
numCorners,
CornerPermu,
CornerOrien,
Corner (..),

-- ** (De)construction
cornerPermu,
cornerOrien,
fromCornerPermu,
fromCornerOrien,

-- * Edges
numEdges,
EdgePermu,
EdgeOrien,
Edge (..),

-- ** (De)construction
edgePermu,
edgeOrien,
fromEdgePermu,
fromEdgeOrien,

-- * Conversions
stringOfCubeColors,
toFacelet,
colorFaceletsToCube,

-- * UDSlice
numUDSliceEdges,
UDSlicePermu,
UDSlice,
UDSlicePermu2,
UDEdgePermu2,
FlipUDSlice,
FlipUDSlicePermu,

-- ** (De)construction
uDSlicePermu,
uDSlice,
```

```
  uDSlicePermu2,
  uDEdgePermu2,
  edgePermu2,
  fromUDSlicePermu,
  fromUDSlice,
  fromUDSlicePermu2,
  fromUDEdgePermu2,

  -- ** Symmetry
  conjugateUDSlicePermu,
  conjugateFlipUDSlice,
  conjugateFlipUDSlicePermu,
  conjugateCornerOrien
  ) where

import Rubik.Cube.Cubie.Internal
```

Facelet.hs

```
{- |
   Facelet representation

   Facelets faces are unfolded and laid out like this:

   @
       U
     L F R B
       D
   @

   Faces (or colors) are ordered @U, L, F, R, B, D@.

   A Rubik's cube is a permutation of facelets numbered as
follows:

   >            0  1  2
   >            3  4  5
   >            6  7  8
   >
   >  9 10 11  18 19 20  27 28 29  36 37 38
   > 12 13 14  21 22 23  30 31 32  39 40 41
```

```
   > 15 16 17  24 25 26  33 34 35  42 43 44
   >
   >           45 46 47
   >           48 49 50
   >           51 52 53

-}

module Rubik.Cube.Facelet (
  -- * Facelet permutation
  numFacelets,
  Facelets,
  facelets,
  fromFacelets,

  -- * Colors
  Color,
  colorOf,
  colorChar,

  -- * Color list
  ColorFacelets,
  colorFacelets,
  fromColorFacelets,
  colorFaceletsOf,

  -- * List conversions
  fromFacelets',
  facelets',
  fromColorFacelets',
  colorFacelets',
  colorFacelets'',

  -- * Pretty conversion
  stringOfFacelets,
  stringOfColorFacelets,
  stringOfColorFacelets',

  -- * Facelets corresponding to each cubie

  -- | The first letter in the name of a cubie is
```

```
  -- the color of its reference facelet
  -- (illustrated at @http://kociemba.org/math/cubielevel.htm@).
  --
  -- Corner colors are given in clockwise order.
  --
  -- Corners are lexicographically ordered
  -- (@U>L>F>R>B>D@).
  --
  -- Edges are gathered by horizontal slices (@U, D, UD@).

  -- ** Centers
  centerFacelets,

  -- ** Corners
  cornerFacelets,
  ulb, ufl, urf, ubr, dlf, dfr, drb, dbl,

  -- ** Edges
  edgeFacelets,
  ul, uf, ur, ub, dl, df, dr, db, fl, fr, bl, br
  ) where

import Rubik.Cube.Facelet.Internal
```