Project Report

Chengtian Xu

Abstract

This is the project report for my final project for Parallel Functional Programming class 2019. My project was Othello played by two AIs and with minimax of depth 3 and 4. To finish a game with depth > 3 with single core takes really long time, but with parallelism and 4 cores it is quite fast.

The program does a little simple rendering of the game, one snapshot of the final board looks like:

```
+----------+----------+----------+----------+----------+----------+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+----------+----------+----------+----------+----------+----------+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | X |
+----------+----------+----------+----------+----------+----------+
| 1 | X | 0 | 0 | 0 | X | 0 | X |
+----------+----------+----------+----------+----------+----------+
| 2 | X | 0 | 0 | 0 | X | X | X |
+----------+----------+----------+----------+----------+----------+
| 3 | X | X | 0 | 0 | X | X | X |
+----------+----------+----------+----------+----------+----------+
| 4 | X | X | X | X | X | 0 | X |
+----------+----------+----------+----------+----------+----------+
| 5 | 0 | X | X | X | X | X | X |
+----------+----------+----------+----------+----------+----------+
| 6 | 0 | X | X | X | X | X | X |
+----------+----------+----------+----------+----------+----------+
| 7 | 0 | X | X | X | X | X | X |
+----------+----------+----------+----------+----------+----------+
```

Compile & Run

- Prerequisites: stack/cabal ghc, threadscope
- To compile, run `stack ghc -- -O2 -threaded -rtsopts -eventlog othello.hs`
- To run with single core and display time analytics, run `./othello +RTS -N1 -s`
To run with four cores and display time analytics, run 

```
./othello +RTS -N4 -s
```

To run and output an eventlog for threadscope to inspect, run 

```
./othello +RTS -N1 -l
```

this outputs othello.eventlog

To inspect with threadscope, run threadscope othello.eventlog

Performance Enhancement with Strategy

**Parts of Program Parallelized**

The major place for the program to be parallelled at was inside the minimax algorithm. When a player A using minimax tries to maximize its advantage over the opponent, it evaluates multiple branches (depends on games, in this case the possible legal moves) down to a certain depth, and then choose the one with maximum advantage.

**Haskell tool used for Parallelism**

I choose to use the Strategy package (parList, parWith, rdeepSeek, etc.) because they provide very easily usable parallel strategies on top of different datastructures, also I really like how it separates the algorithm from the parallelism using keywords like using. They make the code easy to understand. The exact strategy I used is using parList rseq when I called map.

**Result and Performance measurement**

1. Experiments with minimax of depth 3

   - Since my Mac is quad-core, I tested at most with 4 cores.
   - When I ran with ./othello +RTS -N1 -s, the result is the following:
Threadscope shows the following:

When I ran with `.othello +RTS -N4 -s`, the result is the following:

- 57,254,026,704 bytes allocated in the heap
- 137,107,488 bytes copied during GC
- 87,360 bytes maximum residency (28 sample(s))
- 29,192 bytes maximum sloop
- 0 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th></th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>54823 colls, 0 par</td>
<td>0.422s</td>
<td>0.0000s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>28 colls, 0 par</td>
<td>0.0003s</td>
<td>0.0003s</td>
</tr>
</tbody>
</table>

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

SPARKS: 368083 (0 converted, 0 overflowed, 0 dud, 241450 GC'd, 126633 fizzled)

- INIT time 0.000s (0.002s elapsed)
- MUT time 21.105s (21.343s elapsed)
- GC time 0.425s (0.462s elapsed)
- EXIT time 0.000s (0.008s elapsed)
- Total time 21.530s (21.815s elapsed)

Alloc rate 2,712,810,935 bytes per MUT second

Productivity 98.0% of total user, 97.8% of total elapsed

- When I ran with `.othello +RTS -N4 -s`, the result is the following:
Threadscope shows the following:

---

X won by 4
60,763,541,784 bytes allocated in the heap
153,064,880 bytes copied during GC
367,544 bytes maximum residency (213 sample(s))
70,528 bytes maximum slop
0 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th>Gen</th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15018 colls, 15018 par</td>
<td>14.695s</td>
<td>0.197s</td>
</tr>
<tr>
<td>1</td>
<td>213 colls, 212 par</td>
<td>0.287s</td>
<td>0.014s</td>
</tr>
</tbody>
</table>

Parallel GC work balance: 69.96% (serial 0%, perfect 100%)

TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

SPARKS: 384477 (35697 converted, 0 overflowed, 0 dud, 225243 GC'd, 123537 fizzled)

INIT  time  0.001s (  0.003s elapsed)
MUT  time  9.339s (  6.098s elapsed)
GC    time  14.982s (  0.211s elapsed)
EXIT time  0.000s (  0.012s elapsed)
Total time  24.323s (  6.324s elapsed)

Alloc rate  6,506,094,748 bytes per MUT second

Productivity 38.4% of total user, 96.4% of total elapsed

---

- Threadscope shows the following:
2. Experiment with minimax of depth 4 (4 core test only).

- When I ran with `.othello +RTS -N4 -s`, the result is the following:
From the above experiment, I was able to achieve a speedup of $(21.815 / 6.324) = 3.45$, which is quite good given we are using 4 times as many cores.

**Speedup Analysis**

**Code Listing**

```haskell
import Data.List as L
import Data.Maybe
import qualified Data.Map as M
import Control.Parallel.Strategies(using, parList, rseq)

data Color = White | Black | Empty deriving (Eq, Show)
type Pos = (Int, Int)
type Board = M.Map Pos Color

-- Flip the current color to get next color
flipC :: Color -> Color
flipC White = Black
flipC Black = White
flipC _ = Empty

-- All possible legal moves a given current player and board
allMoves :: Color -> Board -> [Pos]
allMoves color board = filter (isLegal color board) [(x, y) | x <- [0..7], y <- [0..7]]
  where isLegal color board pos = cellsChangedColor board pos /= []
```
-- Number of cells changed due to a step

cellsChanged :: Color -> Board -> Pos -> [Pos]
cellsChanged color board pos
  | null flipped = []
  | otherwise   = pos : flipped
  where flipped = concatMap (rowChange True color board pos)
    [(0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1)]

rowChange isFirst color board pos dir
  | nextColor == Just (flipC color) = case restOfRow of
    []      -> []
    (x:xs) -> if isFirst then
              restOfRow
              else pos :
    restOfRow
      | nextColor == Just color = [pos | not isFirst]
      | otherwise = []
      where nextPos = ((x, y) (dx, dy) -> (x + dx, y + dy)) pos dir
           nextColor = M.lookup nextPos board
           restOfRow = rowChange False color board nextPos dir

-- Calculates advantage of a player/color
advCount :: Color -> Board -> Int
advCount color board = sum $ map (\(_, x) -> advPerCell x) $ M.toList board
  where
    advPerCell x
      | x == color = 1
      | x == Empty = 0
      | otherwise = -1

-- Heuristic for bottom level miniMax
heuristic :: Color -> Board -> Int
heuristic color board = advCount color board + 20 * optCountAdv color board
  where
    optCountAdv :: Color -> Board -> Int
    optCountAdv color board = optCounts color board - optCounts (flipC color) board
    optCounts cl bd = length $ allMoves cl bd

-- Played a move and get a new board
step :: Color -> Board -> Pos -> Board
step color board pos = M.union
  (M.fromList (zip (cellsChanged color board pos) (repeat color))) board

-- Optimal move a player can take
optMove :: Color -> Board -> Pos
optMove color board =
  fst $ maximumBy (\(_, x) (_, y) -> compare x y)
  (map (\pos -> (pos, miniMax 4 color (step color board pos)))
       (allMoves color board))
-- The minimax algorithm
miniMax :: Int -> Color -> Board -> Int
miniMax depth color board
| gameOver = if advCount color board > 0
  then 10000000
  else -10000000
| depth <= 0 = heuristic color board
| otherwise = if nc /= color
  then -maxAdvOp
  else maxAdvOp
where
  opMoves = allMoves (flipC color) board
  moves = allMoves color board
  gameOver = null moves && null opMoves
  nc = if opMoves /= [] then flipC color else color
  ncMoves = if nc /= color then opMoves else moves
  maxAdvOp = maximum (map
    (miniMax (depth - 1) nc . step nc board)
    ncMoves `using` parList rseq)

-- Renders the color pieces or empty cells
colorToChar :: Color -> String
colorToChar Empty = " "
colorToChar White = "O"
colorToChar Black = "X"

-- Renders the entire board
renderBoard :: Board -> String
renderBoard board = "\n  0 1 2 3 4 5 6 7 \n  +---+---+---+---+---+---+---+---+
++ "\n  +---+---+---+---+---+---+---+---+
" ++ intercalate "\n  +---+---+---+---+---+---+---+---+
" (map (renderRow board) [0 .. 7])
++ "\n  +---+---+---+---+---+---+---+---+
" where renderRow board row = show row ++ " | " ++
  intercalate " | " [helper (x, row) | x <- [0 .. 7]] ++ " | "
  helper position = colorToChar (fromMaybe Empty (M.lookup position board))

-- Executions after game is over
gameOver :: Color -> Int -> IO ()
gameOver color advCount
| advCount == 0 = putStrLn "Game tie"
| advCount > 0 = putStrLn (colorToChar color ++ " won by " ++
show advCount ++ "\n")
| otherwise = putStrLn (colorToChar (flipC color) ++ " won by " ++
show (-advCount) ++ "\n")

-- Major function of interactive gameplay
go :: Color -> Board -> IO ()
go color board =
  if null (allMoves color board) && null (allMoves (flipC color) board)
  then gameOver color $ advCount color board
else do
  let oc = flipC color
  move = optMove color board
  nb = step color board move
  nc = if allMoves oc nb /= [] then oc else color
  putStrLn (renderBoard board)
  go nc nb

main :: IO ()
main = go White newBoard
  where newBoard = M.fromList
        [((3, 3), White), ((4, 4), White), ((3, 4), Black), ((4, 3), Black)]