Γλώσσες Προγραμματισμού 2
'Ασκηση 6

'Αγγελος Πλεύρης
3115038

April 24, 2020

Παραλληλισμός και ταυτοχρονισμός στη Haskell

Σειριακός Κώδικας

Το πρόγραμμα που ζητείται λαμβάνει για είσοδο έναν αριθμό N και στις επόμενες N γραμμές λαμβάνει τριάδες αριθμών n, k , p. Για κάθε γραμμή εισόδου υπολογίζει το $C(n,k) \mod p = \frac{n!}{k!(n-k)!} \mod p$. Ο σειριακός κώδικας δίνεται στο αρχείο Serialcore.hs που περιέχει συναρτήσεις διαβάσματος της εισόδου από το stdin, καθώς και τον υπολογιστικό πυρήνα που θα χρησιμοποιηθεί στις επόμενες υλοποιήσεις:

```
import Math.Combinatorics.Exact.Binomial

simple_calc::(Integer,Integer,Integer)->Integer
simple_calc (n,k,p) = (n `choose` k) `mod` p
```

Ο υπολογιστικός αυτός πυρήνας δέχεται μια τούπλα 3 ακεραίων (n,k,p) και χρησιμοποιεί την συνάρτηση 'choose' που είναι στο package exact-combinatorics για να υπολογίσει το ζητούμενο αποτέλεσμα. Για την σειριακή υλοποίηση αρκεί να κάνουμε map την παραπάνω συνάρτηση σε μια λίστα δοσμένων τριάδων εισόδου. Έτσι καταλήγουμε στο παρακάτω συνολικό πρόγραμμα της σειριακής υλοποίησης.

```
readInts :: IO [Integer]
readInts = fmap (map read.words) getLine

read_input::Integer->[Integer]->IO [Integer]
read_input n lista =
  if n==0 then do return lista
  else do
    x<- readInts
    read_input (n-1) (lista++x)

transform_res::[Integer]->[(Integer,Integer,Integer)]
transform_res [] = []
transform_res (n:k:p:xs) = (n,k,p):transform_res xs

calc_results::[(Integer,Integer,Integer)]->[Integer]
calc_results xs = map simple_calc xs
```

```
main :: IO ()
main = do
  [t] <- readInts
  lista <- read_input t []
  mapM_ print (calc_results (transform_res lista))
```

Παραλληλισμός

Για την υλοποίηση του παραλληλισμού στη Haskell έγινε χρήση των Evaluation Strategies. Στην παραλλαγή αυτή κρατάμε τον υπολογιστικό πυρήνα simple_calc και με τη βοήθεια του φτιάχνουμε την συνάρτηση par_calc_results, σε αντιστοιχία με την calc_results του σειριακού κώδικα:

```
par_calc_results :: [(Integer, Integer, Integer)] -> Eval [Integer]
par_calc_results xs = mapM (rpar . simple_calc) xs
```

Ο παραπάνω κώδικας φτιάχνει ένα spark για κάθε στοιχείο της λίστας xs, κάθε στοιχείο της οποίας είναι μια τούπλα (n,k,p) για τον υπολογισμό C(n,k) mod p. Κάθε spark δεν αποτελεί ένα πραγματικό thread του λειτουργικού συστήματος αλλά τρέχοντας το πρόγραμμα με κατάλληλες σημαίες για multithreaded support τα sparks κατανέμονται σε πραγματικά threads μέσω της τεχνικής του work-stealing, επιτυχγάνοντας έτσι παραλληλισμό και βελτίωση της απόδοσης. Το τελικό τρέξιμο του προγράμματος γίνεται με το παρακάτω κομμάτι κώδικα:

```
main :: IO ()
main = do
  [t] <- readInts
  lista <- read_input t []
  mapM_ print (runEval (par_calc_results (transform_res lista)))
```

Ταυτοχρονισμός

Οι δυνατότητες που μας παρέχει η Haskell σχετικά με τον ταυτοχρονισμό είναι 2, τα MVars στο IO Monad και το Software Transaction Memory με τα TVars. Στη συγκεκριμένη άσκηση χρησιμοποιήθηκαν τα MVars. Η λογική του κώδικα είναι ότι για κάθε στοιχείο που πρέπει να υπολογιστεί γεννάμε ένα πραγματικό thread το οποίο υπολογίζει το ζητούμενο αποτέλεσμα και το αποθηκεύει σε ένα MVar. Έπειτα επιστρέφει αυτό το MVar. Τέλος για την εκτύπωση των αποτελεσμάτων το κύριο thread παίρνει τις τιμές από όλα τα MVar και τις τυπώνει.

```
calcThread::MVar Integer -> (Integer,Integer,Integer) -> IO ()
calcThread resultMVar nkp =
  do
    pseq f (return ())
    putMVar resultMVar f
  where
    f = simple_calc nkp

conc_calc::(Integer,Integer,Integer) -> IO (MVar Integer)
conc_calc x = do
  resMVar <-newEmptyMVar
  forkIO (calcThread resMVar x)
  return resMVar

conc_sum::[(Integer,Integer,Integer)] -> IO ([MVar Integer])
conc_sum lst = mapM conc_calc lst

main:: IO ()
main = do
  [t] <- readInts
  lista <- read_input t []
  mvar_list <- conc_sum (transform_res lista)
  result_list <- mapM takeMVar mvar_list
  mapM_ print result_list
```

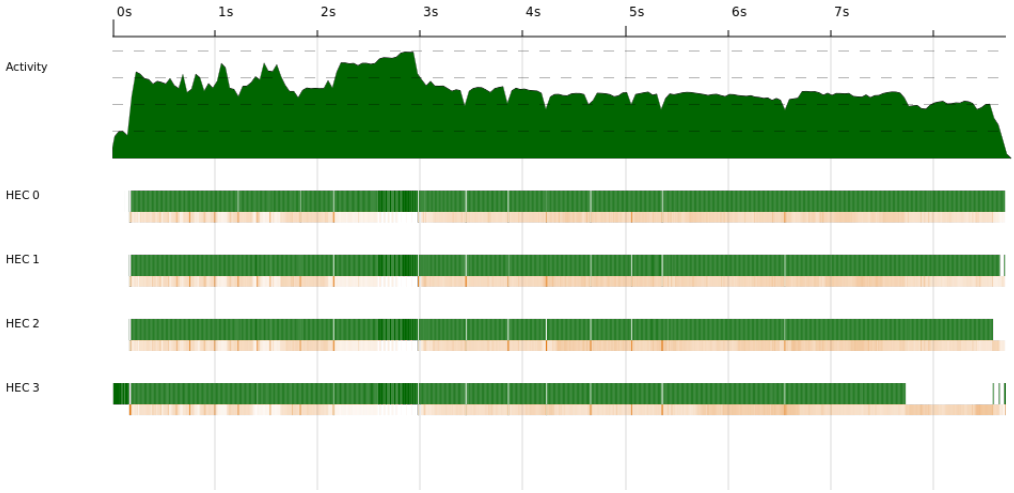
Στον παραπάνω κώδικα αξίζει να παρατηρήσουμε τη χρήση του pseq που αναγκάζει τον υπολογισμό της τιμής f και ουσιαστικά μας γλιτώνει από την οκνηρή αποτίμηση. Επίσης παρατηρούμε στην conc_calc το κομμάτι κώδικα με παρενέργειες που δημιουργούμε ένα MVar, έπειτα το thread που γεννάμε γράφει στο MVar και τέλος το επιστρέφουμε.

Παρουσίαση Αποτελεσμάτων

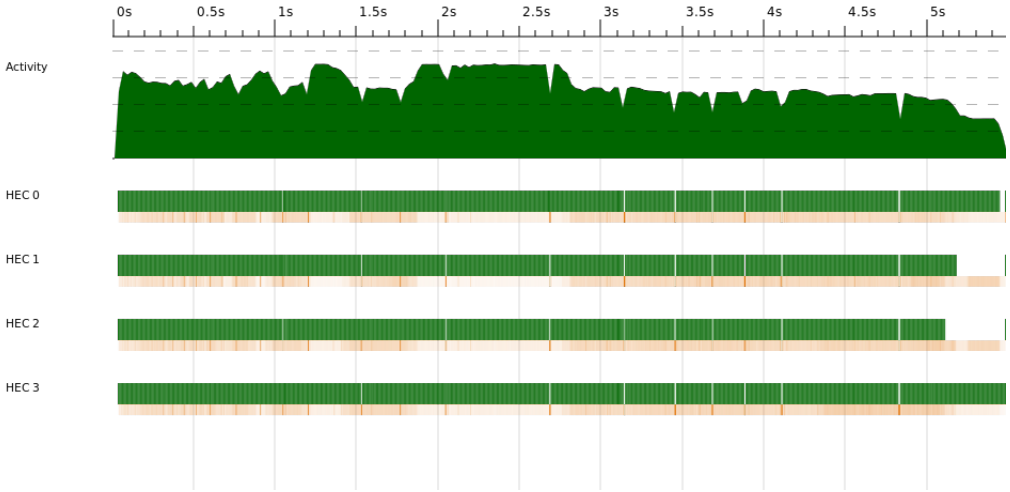
Παρακάτω θα παρουσιαστούν και θα σχολιαστούν οι επιδόσεις των προγραμμάτων που δημιουργήθηκαν και θα δούμε και τη χρήση των υπολογιστικών πυρήνων του υπολογιστή με τη βοήθεια του εργαλείου threadscope.

Αρχικά θα δούμε την χρησιμοποίηση των επεξεργαστών του μηχανήματος μας.:

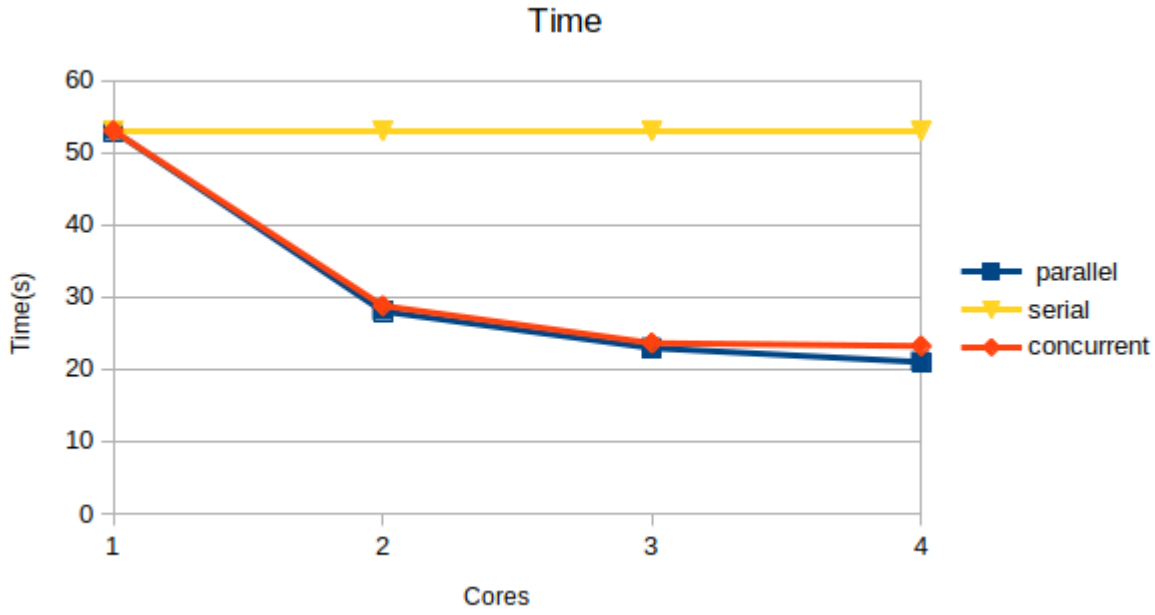
Parallel Version

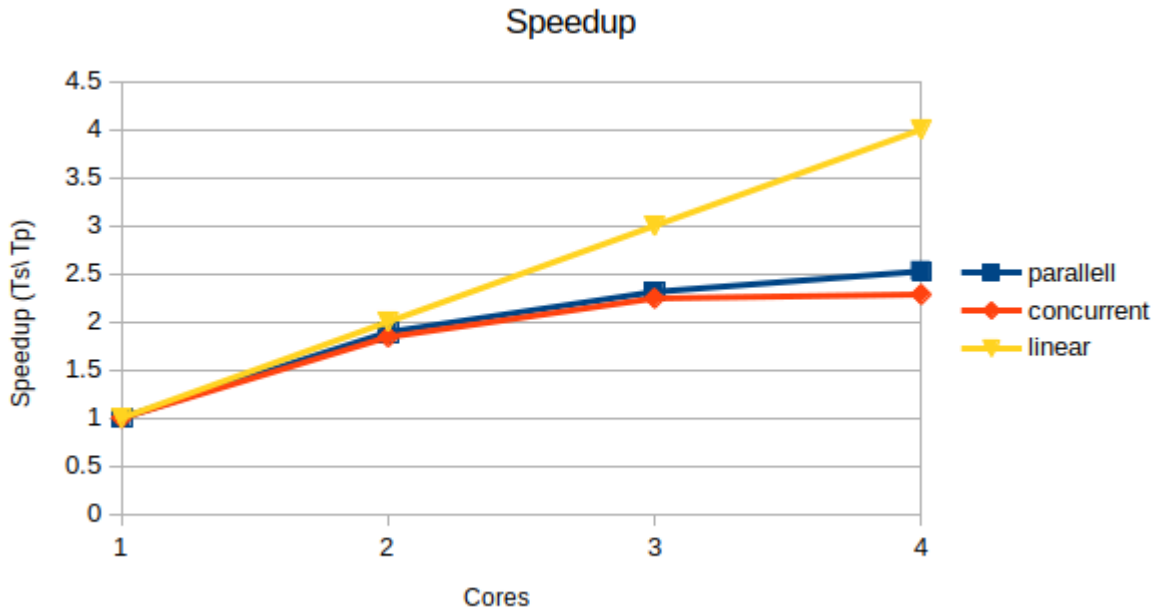


Concurrent Version



Παρατηρούμε ότι και στα δύο προγράμματα χρησιμοποιούνται και τα 4 φυσικά cores του επεξεργαστή μας με μικρές διαφορές στον χρόνο που είναι ενεργό το κάθε core. Αυτό συμβαίνει γιατί έχοντας ορίσει ως task τον υπολογισμό για κάθε ένα (n,k,p) ,ανάλογα την τάξη μεγέθους των n,k ο χρόνος υπολογισμού είναι διαφορετικός.'Ετσι υπάρχει περίπτωση τα task να μην κατανέμονται δίκαια καθώς δεν υπάρχει κάποιο στοιχείο που να μας μαρτυρά εκ των προτέρων πως να χωρίσουμε τα tasks κατάλληλα σε επεξεργαστές. Οι παραπάνω μετρήσεις λήφθηκαν για σχετικά ίσα tasks. Το πρόβλημα που αναφέραμε, γίνεται καλύτερα αισθητό παρακάτω που παρουσιάζουμε τα διαγράμματα χρόνου και επιτάχυνσης όπου δεν παίρνουμε πάρα πολύ καλό speedup.





Ακόμη, αξίζει να παρτηρήσουμε ότι η concurrent εκδοχή στην γενική περίπτωση έχει λίγο χειρότερη επίδοση από την parallel καθώς τα sparks της parallel είναι πιο ελαφριά από τα OS threads που δημιουργεί η concurrent.