Γλώσσες Προγραμματισμού 2
'Ασκηση 6

'Αγγελος Πλεύρης
3115038

April 24, 2020

Παραλληλισμός και ταυτοχρονισμός στη Haskell

Συνολικά δημιουργήθηκαν 4 αρχεία .hs . Στο αρχείο Serialcore.hs δίνονται συναρτήσεις για το διάβασμα της εισόδου (σειριακό) καθώς και ο κύριος υπολογιστικός πυρήνας πάνω στον οποίο πατάνε οι υπόλοιποι κώδικες. Στα αρχεία Serhaskell.hs , Parhaskell.hs , Conchaskell.hs δίνονται οι επιμέρους αλλαγές για την υλοποίηση του σειριακού, του παράλληλου και του ταυτόχρονου κώδικα.

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 που περιέχει συναρτήσεις διαβάσματος της εισόδου από το stdin, καθώς και τον υπολογιστικό πυρήνα που θα χρησιμοποιηθεί στις επόμενες υλοποιήσεις, δίνεται παρακάτω:

```
gcdExtended :: Integer->Integer->Integer->Integer
                  ->(Integer, Integer, Integer, Integer)
gcdExtended a b x y =
  if a == 0 then (a,b,0,1)
  else (a,b,x1,y1) where
    (_,_,retx,rety) = gcdExtended (b `mod` a) a x y
    x1 = rety - b `div` a * retx
    y1 = retx

inverse :: Integer->Integer->Integer
inverse x p =
  let (_,_,ret,_) = gcdExtended x p 0 0 in
    if ret < 0 then ret + p
      else ret

modMult :: Integer->Integer->Integer->Integer
modMult p a b= (a*b) `mod` p

modFact :: Integer->Integer->Integer->Integer
modFact n k p =
  foldl' (modMult p) 1 [k+1..n]
```

```
simple_calc::(Integer,Integer,Integer)->Integer
simple_calc (n,k,p) =
  modMult p nk invnminusk where
    nk = modFact n k p
    invnminusk = inverse (modFact (n-k) 1 p) p
```

Ο υπολογιστικός αυτός πυρήνας δέχεται μια τούπλα 3 ακεραίων (n,k,p) και χρησιμοποιεί τις υπόλοιπες συναρτήσεις για να υπολογίσει το ζητούμενο αποτέλεσμα. Συγκεκριμένα υπολογίζει τον αριθμό $\frac{n!}{k!} \mod p$, έπειτα τον inverse αριθμό του $(n-k)!$ σε σχέση με το p, όπου ορίζουμε ως $a = inverse(x,p)$ τον αριθμό a ώστε $ax \mod p = 1$. Για τον υπολογισμό του inverse κάνουμε χρήση του Εκτενούς Αλγορίθμου του Ευκλείδη. Με αυτόν τον τρόπο ο υπολογισμός του C(n,k) mod p δίνεται από τον τύπο $\frac{n!}{k!} inverse((n-k)!,p) \mod p$ Για την σειριακή υλοποίηση αρκεί να κάνουμε map την συνάρτηση simple_calc 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. Για την συλλογή αποτελεσμάτων τρέξαμε το αρχείο εισόδου inputeven.in που περιέχει 64 ίδια tasks.

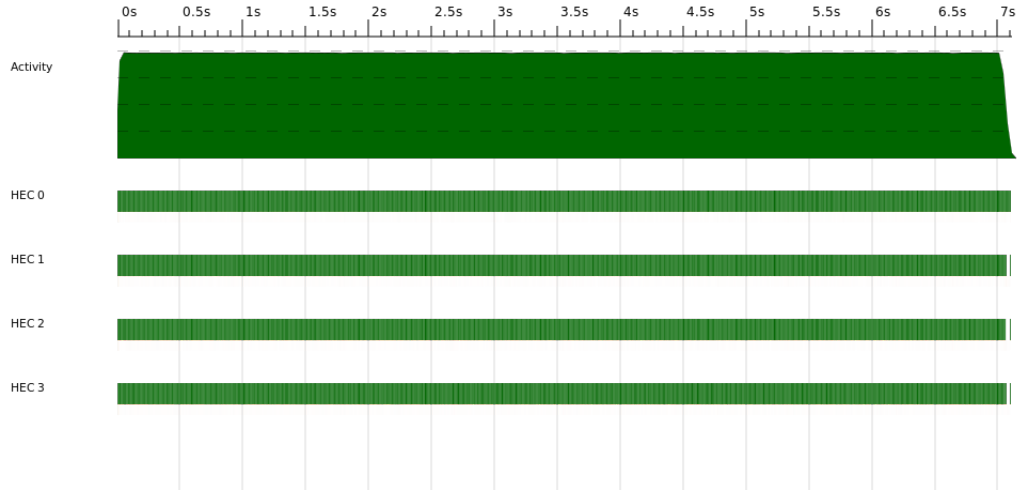
Αρχικά θα δούμε την χρησιμοποίηση των επεξεργαστών του μηχανήματος μας καθώς και την δημιουργία των sparks για την παράλληλη έκδοση:

Sparks created

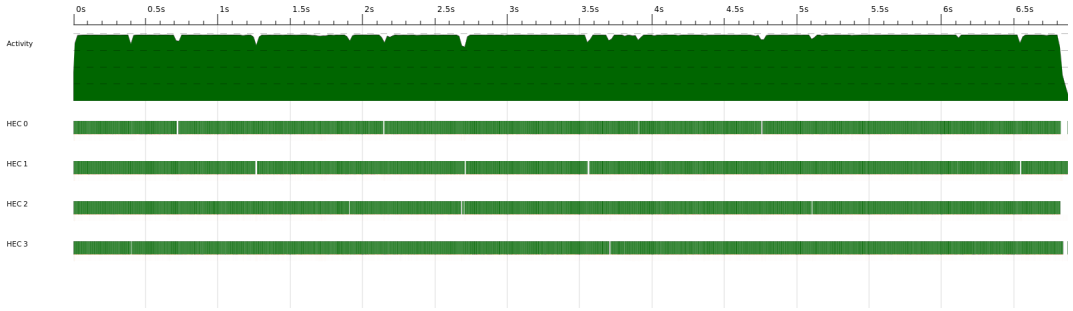
```
SPARKS: 64 (64 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.002s  (  0.002s elapsed)
MUT     time   12.814s  (  7.029s elapsed)
GC      time   15.437s  (  0.065s elapsed)
EXIT    time    0.005s  (  0.006s elapsed)
Total   time   28.257s  (  7.102s elapsed)
```

Parallel Version



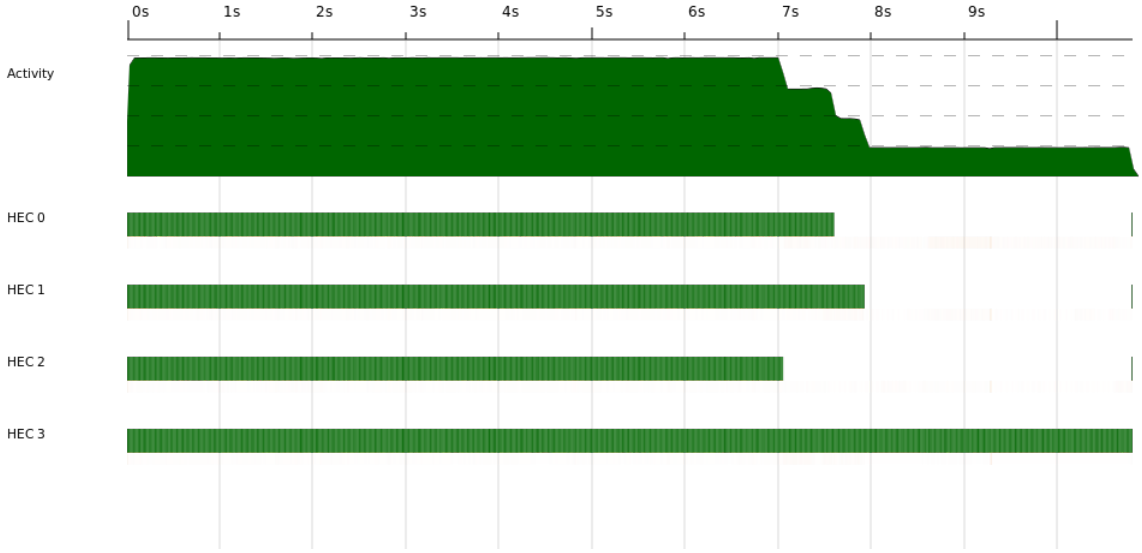
Concurrent Version



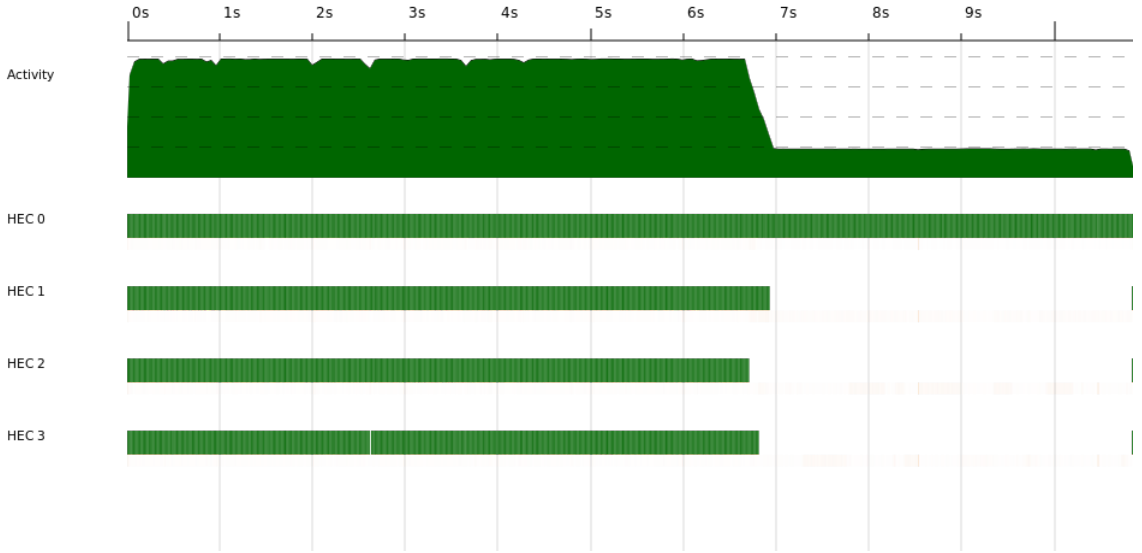
Παρατηρούμε ότι και στα δύο προγράμματα χρησιμοποιούνται και τα 4 φυσικά cores του επεξεργαστή μας.Σε αυτό το σημείο πρέπει να τονίσουμε ότι υπάρχει ισοκατανομή των tasks γιατί όλα έχουν τον ίδιο χρόνο υπολογισμού στο αρχείο inputeven.in . Αντίθετα επειδή δεν υπάρχει τρόπος ,δεδομένου του προβλήματος , να ξέρουμε τον χρόνο υπολογισμού για κάθε task εκ των προτέρων ,στις περισσότερες περιπτώσεις (με τυχαία νούμερα) δεν μπορούμε να έχουμε τέτοια ισοκατανομή των εργασιών και επακόλουθα τόσο καλό speedup όσο στις συγκεκριμένες μετρήσεις. Παρακάτω φαίνονται οι μετρήσεις για το συγκεκριμένο αρχείο , όπου παίρνουμε πάρα πολυ ικανοποιητικά αποτελέσματα καθώς και η χρησιμοποίηση των επεξεργαστών για το αρχείο εισόδου inputuneven.in στο οποίο φαίνεται ότι κάποια tasks καθυστερούν παραπάνω κάποιον επεξεργαστή με εμφανή αποτελέσματα στην απόδοση. Στη γενική περίπτωση επειδή διαβάζουμε τυχαία είσοδο στο πρόβλημα μας η αναμενόμενη συμπεριφορά του επεξεργαστή είναι η παρακάτω , δηλαδή με ανισοκατανομή . Το μέγιστο speedup το παίρνουμε για tasks με ίδιο υπολογιστικό φόρτο το οποίο φαίνεται στα αρχικά διαγράμματα καθώς και στα διαγράμματα χρόνου και επιτάχυνσης.

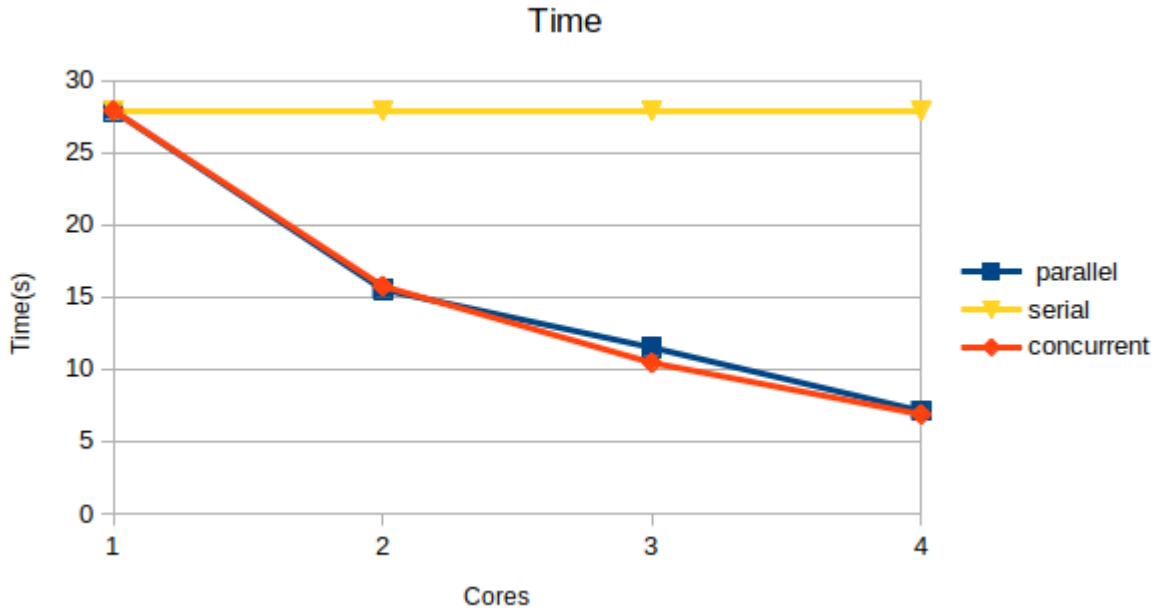
Ανισοκατανομή φόρτου

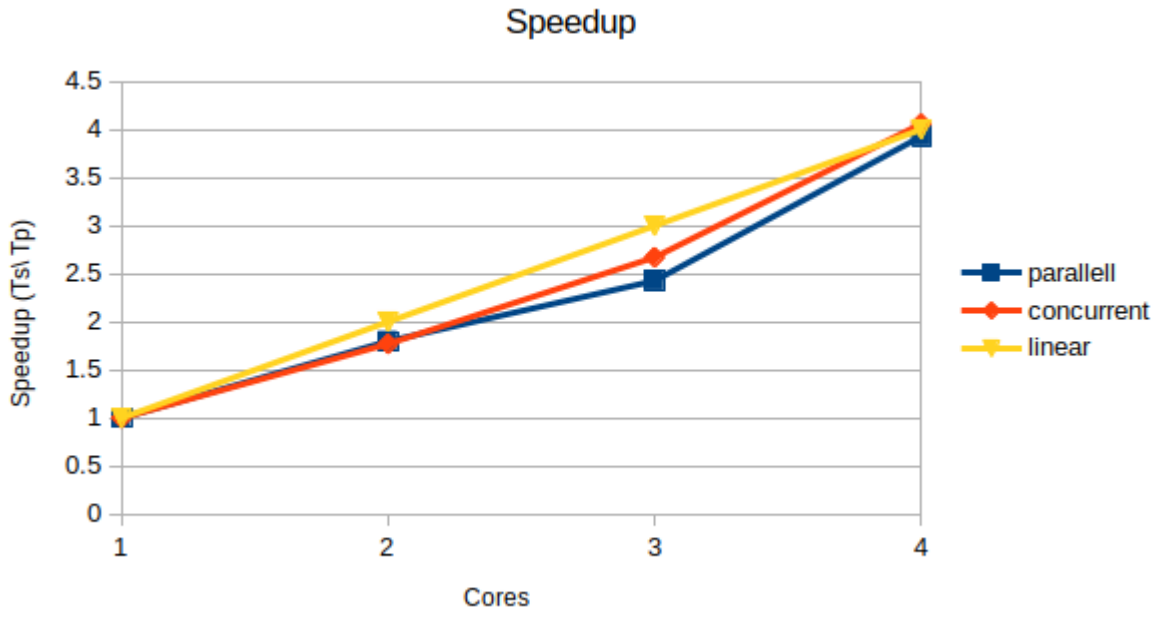
Parallel Version



Concurrent Version







Για ισοκατανομή φόρτου εργασίας (ισοκατανεμημένη είσοδος) βλέπουμε ότι παίρνουμε σχεδόν γραμμικό speedup.