# Γλώσσες Προγραμματισμού II
# Parallel & Concurrent Haskell

Μιλτιάδης Στούρας
ΑΜ: 03116022

Ακ. Έτος 2019-2020 - Ροή Λ

## Εισαγωγή

Μας δίνονται $Q$ queries της μορφής $(n,\ k,\ p)$, τα οποία ζητούν να υπολογίσουμε το υπόλοιπο της διαίρεσης του δυωνυμικού συντελεστή $\binom{n}{k}$ με τον πρώτο αριθμό $p$:

$$\binom{n}{k} \mod p$$

Γνωρίζουμε πως

$$\binom{n}{k} = \frac{n!}{(n-k)!\ k!} = \frac{\prod_{i=0}^{k-1} n-i}{k!}$$

Για να διαιρέσουμε σε αριθμητική modulo με έναν αριθμό $\alpha$, πρέπει να πολλαπλασιάσουμε με τον αντίστροφο του $\alpha$. Επειδή ο $p$ είναι πρώτος αριθμός, μπορούμε να χρησιμοποιήσουμε το μικρό θεώρημα του Fermat για να υπολογίσουμε τον $\alpha^{-1}$:

$$\alpha^{p-1} \equiv 1 \mod p \ \Rightarrow\ \alpha^{p-2} \equiv \alpha^{-1} \mod p$$

Για τον υπολογισμό του $(k!)^{p-2} \mod p$, χρησιμοποιούμε τον αλγόριθμο επαναλαμβανόμενων τετραγωνισμών του Gauss ο οποίος θα χρειαστεί $O(\log p)$.

## Παραλληλοποίηση

Υλοποιήσαμε την συνάρτηση *solve* η οποία επιστρέφει την απάντηση για ένα δεδομένο query. Αρχικά υπολογίζει τον αριθμητή του κλάσματος σε $O(n)$ και έπειτα τον αντίστροφο του παρονομαστή σε $O(k + \log p)$. Γράψαμε ένα σειριακό πρόγραμμα, *serial.hs*, που διαβάζει τα queries και τυπώνει τις απαντήσεις τους χρησιμοποιώντας την *solve*.
Η πρώτη προσπάθεια παραλληλοποίησης του προγράμματος ήταν να δημιουργήσουμε ένα spark για κάθε query, το οποίο κάναμε χρησιμοποιώντας την *parMap* της Haskell. Η υλοποίηση είναι πανομοιότυπη με το σειριακό πρόγραμμα και βρίσκεται στο αρχείο *parallel1.hs*.
Για να μετρήσουμε την βελτίωση στον χρόνο εκτέλεσης, δημιουργήσαμε ένα testcase με 1000 queries. Για κάθε query επιλέξαμε τυχαία έναν πρώτο αριθμό $p$ στο διάστημα $[1, 10^9]$, έναν αριθμό $n$ στο $[1, p)$ και έναν αριθμό $k$ στο $[1, n)$.
Το πρόγραμμα έχει δυο εγγενώς σειριακά κομμάτια, το διάβασμα της εισόδου και το τύπωμα της εξόδου. Χρησιμοποιώντας το *threadscope*, παρατηρούμε πως το διάβασμα της εισόδου διαρκεί περίπου 7 $ms$ και το τύπωμα της εξόδου περίπου 35 $ms$. Το σειριακό πρόγραμμα χρειάζεται συνολικά 4.661 $s$, επομένως το ποσοστό του προγράμματος που είναι σειριακό είναι:

$$S = \frac{42}{4661} = 1\%$$

Από τον νόμο του Amdhal γνωρίζουμε ότι το μέγιστο θεωρητικό speedup που μπορούμε να πετύχουμε με $N$ cores είναι:

$$\frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(0.01)+\frac{0.99}{N}}$$

Επομένως, το μέγιστο speedup που μπορούμε να πετύχουμε ανά αριθμό πυρήνων είναι το παρακάτω:

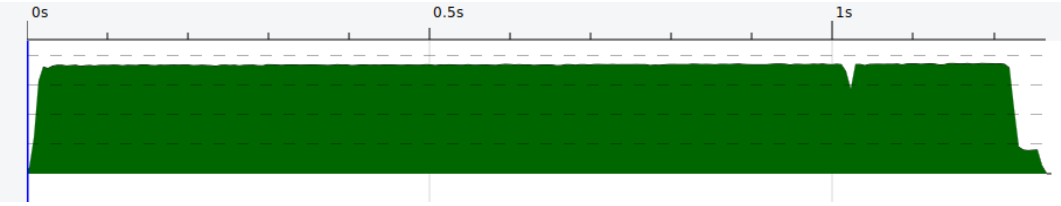
| **Number of cores** | **Ideal Speed-up** |
|---|---|
| 2 | 1.980 |
| 3 | 2.941 |
| 4 | 3.883 |
| 5 | 4.807 |
| 6 | 5.714 |
| 7 | 6.604 |
| 8 | 7.477 |

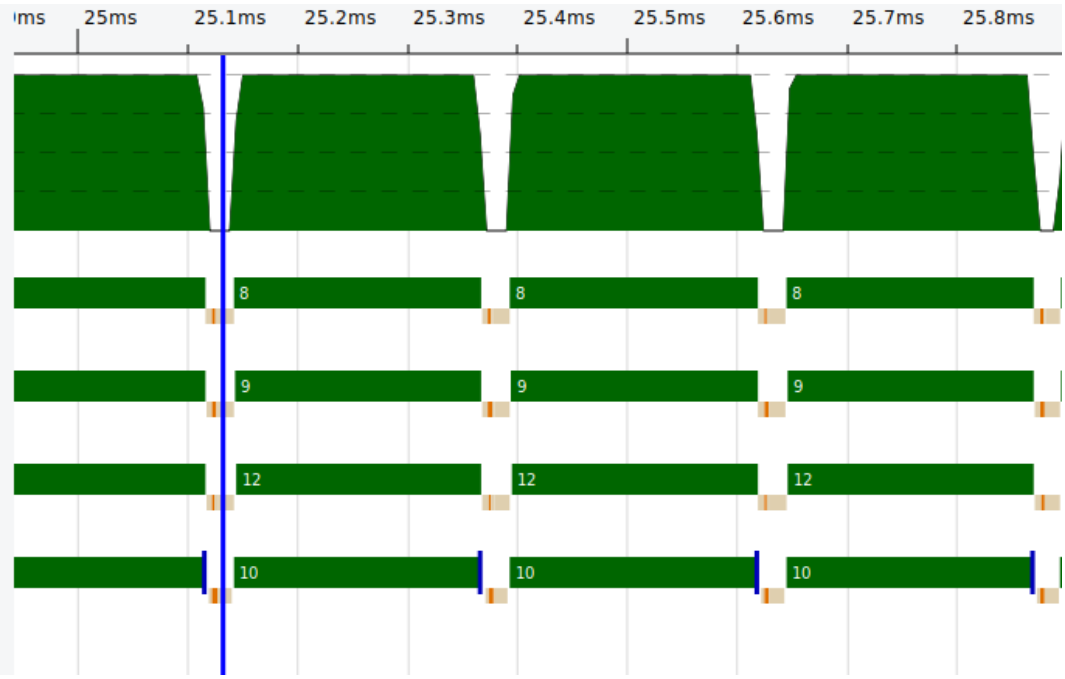
Η πρώτη απόπειρα παραλληλοποιήσης που περιγράψαμε παραπάνω, επιτυγχάνει τα ακόλουθα speedups:

| **Number of cores** | **Running time (sec)** | **Speed-up** | **Ideal Speed-up** |
|---|---|---|---|
| 2 | 2.441 | 1.912 | 1.980 |
| 3 | 1.653 | 2.823 | 2.941 |
| 4 | 1.284 | 3.634 | 3.883 |
| 5 | 1.073 | 4.349 | 4.807 |
| 6 | 0.924 | 5.050 | 5.714 |
| 7 | 0.834 | 5.595 | 6.604 |
| 8 | 0.744 | 6.271 | 7.477 |

Πίνακας 1: Par1 speed-up

Παρατηρούμε πως για μικρό αριθμό πυρήνων το speed-up είναι ικανοποιητικό και αρκετά κοντά στο θεωρητικό. Κάνοντας μια επισκόπηση της εκτέλεσης με το εργαλείο threadscope, παρατηρούμε πως οι πυρήνες βρίσκονται σε full utilization και κάνουν κάποιες διακοπές μόνο για garbage collection.

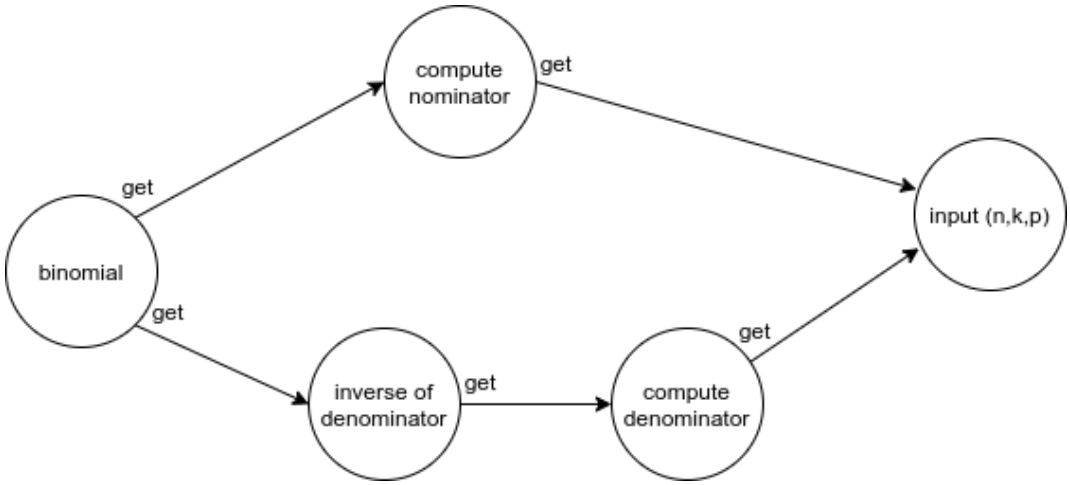





Σχήμα 1: Par1 execution on 4 cores

Προσπαθώντας να βελτιώσουμε την προηγούμενη προσέγγιση, σπάσαμε τον υπολογισμό κάθε επιμέρους ποσότητας σε μια διαφορετική δουλειά η οποία μπορεί να γίνει παράλληλα. Συγκεκριμένα, υπολογίζουμε τον αριθμητή παράλληλα με τον υπολογισμό του παρονομαστή τον οποίο επίσης σπάμε σε 2 δουλειές οι οποίες μπορούν να γίνουν από διαφορετικούς επεξεργαστές αλλά έχουν διαδοχική εξάρτηση. Με τον τρόπο αυτό αφήνουμε το λειτουργικό να κατανέμει καλύτερα και με δυναμικό τρόπο τις δουλειές, έτσι ώστε αν υπάρχει κάποιος idle επεξεργαστής η δουλειά ενός query να σπάει σε μικρότερα κομμάτια, κάποια από τα οποία μπορεί να αναλάβει και ο επεξεργαστής που κάθεται. Το τελικό αποτέλεσμα ήταν μια συστηματική βελτίωση της τάξης των 20-40$ms$ στον χρόνο εκτέλεσης του test και μια αντίστοιχη αύξηση στο speedup.

Η υλοποίηση έγινε χρησιμοποιώντας το spawnP. Ισοδύναμα θα μπορούσε να υλοποιηθεί και με IVars, το dataflow diagram των οποίων θα έμοιαζε με το παρακάτω.




Τα αποτελέσματα του Par2 ήταν τα ακόλουθα:

| Number of cores | Running time (sec) | Speed-up | Ideal Speed-up |
|---|---|---|---|
| 2 | 2.414 | 1.932 | 1.980 |
| 3 | 1.614 | 2.891 | 2.941 |
| 4 | 1.244 | 3.751 | 3.883 |
| 5 | 1.044 | 4.470 | 4.807 |
| 6 | 0.904 | 5.162 | 5.714 |
| 7 | 0.804 | 5.804 | 6.604 |
| 8 | 0.715 | 6.526 | 7.477 |

Για σύγκριση, τα running times και speed-ups των δύο μεθόδων ήταν:

| Number of cores | Par1 Speed-up | Par2 Speed-up | Ideal Speed-up |
|---|---|---|---|
| 2 | 1.912 | 1.932 | 1.980 |
| 3 | 2.823 | 2.891 | 2.941 |
| 4 | 3.634 | 3.751 | 3.883 |
| 5 | 4.349 | 4.470 | 4.807 |
| 6 | 5.050 | 5.162 | 5.714 |
| 7 | 5.595 | 5.804 | 6.604 |
| 8 | 6.272 | 6.526 | 7.477 |

| Number of cores | Par1 Running Time | Par2 Running Time |
|---|---|---|
| 2 | 2.441 | 2.414 |
| 3 | 1.653 | 1.614 |
| 4 | 1.284 | 1.244 |
| 5 | 1.073 | 1.044 |
| 6 | 0.924 | 0.904 |
| 7 | 0.834 | 0.804 |
| 8 | 0.744 | 0.715 |

Όπως είπαμε, παρατηρούμε μια συστηματική βελτίωση 20-40ms σε κάθε πείραμα.

Τελος, για να βελτιώσουμε περαιτέρω το execution, θα μπορούσαμε να ανοιγούμε sparks πριν διαβάσουμε όλο το input ώστε οι υπόλοιποι επεξεργαστές να μπορούν να αναλάβουν δουλειές όσο διαβάζεται το υπόλοιπο input. Σε μία τέτοια απόπειρα ωστόσο, παρατηρήσαμε χειρότερη απόδοση. Αυτό ίσως να γίνεται επειδή ο επεξεργαστής που διαβάζε το input αναλάμβανε κατευθείαν τα λιγοστά sparks που έβγαιναν και τελικά απλά παρατεινόταν το σειριακό μέρος του προγράμματος.