Πολυτεχνειο Κρητης
Τμημα Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

Αρχιτεκτονική Παράλληλων και Κατανεμημένων Συστημάτων

ΗΡΥ 418 **Αναφορά Εργασίας** 2014

Περικλής Χρυσόγελος Α.Μ.: 2011030064
Νικόλαος Κοφινάς Α.Μ.: 2011039025

Στην εργασία αυτή δημιουργήσαμε μία παράλληλη υλοποίηση του αλγορίθμου negascout για το παιχνίδι 2048. Όπως αποδείχτηκε στην πράξη στο παιχνίδι αυτό μπορούμε να δούμε αρκετές κινήσεις μπροστά αλλά ο σχεδιασμός μια καλής evaluation function δεν είναι εύκολος.

2048

Το παιχνίδι 2048 είναι ένα σχετικά καινούργιο παιχνίδι που είχε αρκετά μεγάλη απήχηση λόγο του απλού τρόπου που παίζετε. Αποτελείτε από ένα board μεγέθους 4×4 όπου το κάθε κελί μπορεί να περιέχει ένα tile το οποίο έχει μία τιμή ίση με κάποια δύναμη του 2(π.χ. 2, 8, 64, 512). Στην εικόνα 1 μπορείτε να δείτε ένα στιγμιότυπο του παιχνιδιού. Σκοπός του παίχτη είναι να ενώνει tiles που έχουν την ίδια τιμή για να δημιουργήσει ένα νέο tile το οποίο θα αποτελείτε από την επόμενη δύναμη του 2 (π.χ. αν ενώσει 2 tiles με τιμή 8 θα δημιουργηθεί το tile με τιμή 16). Οι κινήσεις του παίχτη είναι το πολύ 4: πάνω, κάτω, δεξιά και αριστερά. Ύστερα από κάθε κίνηση του παίχτη ο αντίπαλος τοποθετεί ένα νέο tile, το οποίο έχει τιμή 2 ή 4 μέσα στο board σε οποία θέση θέλει. Στο κανονικό παιχνίδι ο αντίπαλος παίζει τυχαία. Οι παίκτες παίζουν εναλλάξ ξεκινώντας από τον παίχτη ο οποίος κάνει τις 4 κινήσεις και με ένα board το οποίο περιέχει 2 tiles τυχαία τοποθετημένα και με τιμές 2 ή 4. Για περισσότερες πληροφορίες μπορείτε να παίξετε το κανονικό παιχνίδι σε αυτό το link: http://gabrielecirulli.github.io/2048/

Σχήμα 1: Ένα στιγμιότυπο του παιχνιδιού

MiniMax

Στην θεωρία της τεχνητής νοημοσύνης ο αλγόριθμος Mini-Max είναι ίσος ο πιο γνωστός αλγόριθμος αναζήτησης που χρησιμοποιείται σε ανταγωνιστικά παιχνίδια δύο ατόμων. Η γενική ιδέα πίσω από τον αλγόριθμο είναι ότι κοιτάζοντας μπροστά στον χώρο των πιθανών κινήσεων πια είναι η βέλτιστη κίνηση που μπορώ να κάνω αυτή την στιγμή. Για να δει μέσα στο χώρο όλων των πιθανών κινήσεων, 'παίζει' οποιαδήποτε κίνηση μπορεί να κάνει ο παίχτης στην κατάσταση στην οποία βρίσκετε το παιχνίδι...

Mini-Max alpha beta prunning

Ο Mini-max είναι αρκετά αργός και απαιτεί εκθετικό χρόνο για να εκτελεστεί. Με την επέκταση alpha-beta pruning, δεν μπορούμε να διώξουμε τον εκθέτη, αλλά μπορούμε να τον μειώσουμε στο μισό. Η λογική του alpha-beta pruning είναι πολύ απλή, όταν ένας κόμβος καλεί μια συνάρτηση για να αξιολογήσει την κατάσταση στην οποία βρέθηκε μετά από μία κίνηση, ελέγχει τι του επιστρέφει και αν του κάνει, σε σχέση με το προηγούμενο που είχε, το κρατάει και πετάει το προηγούμενο. Με το alpha-beta pruning μπορούμε να προωθούμε την τιμή που έχουμε, στους από κάτω κόμβους, ώστε αν αυτοί βρουν τιμή μεγαλύτερη από αυτή που έχει προωθήσει ο από πάνω κόμβος (στην περίπτωση του max κόμβου) ή μικρότερη (στην περίπτωση του min κόμβου) τότε την επιστρέφουν αμέσως, χωρίς να επεκτείνουν τα εναπομείναντα παιδιά. Ο λόγος που λειτουργεί αυτό, είναι διότι ο minimax αλγόριθμος, θα τα απέρριπτε

αυτά τα παιδιά από αυτή και μόνο την τιμή, χωρίς να τον ενδιαφέρει αν υπάρχει μεγαλύτερη η μικρότερη τιμή, γιατί σίγουρα δεν θα επέλεγε αυτή την κίνηση, οπότε δεν χρειάζεται περαιτέρω ψάξιμο.

Negascout

Ο αλγόριθμος negascout είναι μία βελτίωση του mini-max alpha beta prunning που δουλεύει πάνω σε μία απλή υπόθεση. Η υπόθεση είναι ότι πάντα θα εξετάζει πρώτο το παιδί που με μεγάλη πιθανότητα θα του δώσει το καλύτερο score. Για να λειτουργήσει αυτό χρειάζεται να ξέρει από προηγούμενες, πιο ρηχές αναζητήσεις την σειρά με την οποία θα εξετάσει τα παιδιά. Ο αλγόριθμος αφού εξετάσει το καλύτερο παιδί και βρει το score του, εξετάζει τα υπόλοιπα παιδιά θέτοντας $\beta = \alpha + 1$ το επονομαζόμενο και null window. Η υπόθεση πίσω από το null window είναι ότι τα υπόλοιπα παιδιά θα γίνουν γρήγορα κλάδεμα διότι θα δώσουν χειρότερο score από το τωρινό. Αν το null window αποτύχει για ένα παιδί και αυτό γυρίσει διαφορετικό score τότε ο αλγόριθμος επανεξετάζει αυτό το παιδί χωρίς το null window. Στην πράξη έχει δειχθεί ότι με σωστό move ordering ο negascout έχει ένα speedup της τάξης του 10% σε σχέση με τον alpha beta prunning.

Algorithm 1 Negascout algorithm

Input: node,depth,α,β,color

Output: score

```
1: if node is a terminal node or depth = 0 then
2:    return  color × heuristic evaluation
3: end if
4: for all childs of node do
5:    if child is not first child then
6:       score := -negascout(child, depth−1, −α − 1, −α, -color)
7:       if α <score< β then
8:          score := -negascout(child, depth−1, −β, −α, -color)
9:       end if
10:   else
11:      score := -negascout(child, depth−1, −β, −α, -color)
12:   end if
13:   α:=max(α,score);
14:   if α ≥ β then
15:      break
16:   end if
17: end for
18: return  α
```

Board

Μέχρι αυτό το σημείο, για να υλοποιηθεί οποιοσδήποτε από αυτούς τους αλγορίθμους, το μόνο επιπλέον που χρειάζεται είναι μια αναπαράσταση της κατάστασης του παιχνιδιού και υλοποίηση των κανόνων/κινήσεων του παιχνιδιού πάνω σε αυτή την αναπαράσταση. Για λόγους απόδοσης επιλέχθηκε το Board να αναπαριστάται με τέσσερα bitboard. Δηλαδή, για την αναπαράσταση θα χρησιμοποιούνται τέσσερις 64bit ακέραιοι, οι οποίοι θα περιέχουν συνολικά 16 boards. Κάθε bitboard θα περιέχει 16 bits ένα για κάθε τετράγωνο του ταμπλό. Από τα 16 boards, τα ένα αποφασίστηκε να κρατάει τα τετράγωνα που είναι γεμάτα, ενώ τα άλλα 15 χρησιμοποιούνται ένα για κάθε δύναμη του 2 (ξεκινώντας από το 2) και δίνοντας δυνατότητες για κομμάτια τιμής μέχρι και $2^{15} = 32768$. Επίσης 256 bits χωράν σε μια γραμμή της L1 cache των περισσότερων επεξεργαστών, έτσι ο compiler έχει την δυνατότητα να κάνει σωστή ευθυγράμμιση ώστε να γίνεται καλύτερη αξιοποίηση της L1 cache μνήμης. Επιπλέον, με αυτή την υλοποίηση που έγινε, οι περισσότερες πράξεις αφορούσαν bitwise operations και μάλιστα πολλά από αυτά μέσα σε vectorizable loops ή ακόμη γενικότερα, μπορούσαν να χρησιμοποιηθούν για τις περισσότερες πράξεις SSE/AVX instructions, δίνοντας την δυνατότητα σε έναν καλό compiler να βελτιώσει και να επιταχύνει πολύ την assembly του κώδικα. Προφανώς αυτή η σχεδίαση αυξάνει στο μέγιστο δυνατό

την παραλληλοποίηση σε επίπεδο bit και εντολών. Επιπροσθέτως αξίζει να σημειωθεί ότι με αυτή την υλοποίηση οι προσβάσεις στην μνήμη για τις περισσότερες ενέργειες τουλάχιστον υποτετραπλασιάζονται σε σχέση με την υλοποίηση με πίνακα.

Parallel Negascout
Η λογική που αποφασίσαμε να ακολουθήσουμε για να παραλληλοποίησουμε τον αλγόριθμο είναι η εξής:

1. Ψάχνουμε το ακραίο αριστερό παιδί μέχρι κάτω
2. Αναδρομικά ψάχνουμε τα αδέρφια του και τους κόμβους από πάνω του
3. Από ένα δεδομένο βάθος και πάνω αναλαμβάνει την αναζήτηση κάθε κόμβου στο δεδομένο βάθος ένα thread

Η ιδέα πίσω από αυτή την υλοποίηση είναι πολύ απλή, ψάχνουμε να βρούμε τα πρώτα alpha beta bounds το main thread και ύστερα το main thread διαχωρίζει την δουλειά σε επιμέρους threads. Ο αλγόριθμος παρουσιάζετε στην εικόνα 2.

Algorithm 2 Parallel Negascout algorithm

Input: node,depth,α,β,color
Output: score

```
1: if node is a terminal node or depth = 0 then
2:   return color × heuristic evaluation
3: end if
4: for all childs of node do
5:   if child is not first child then
6:     score := -spawnThread(child, depth−1, −α − 1, −α, -color)
7:     continue
8:   else
9:     score := -parallelNegascout(child, depth−1, −β, −α, -color)
10:  end if
11:  α:=max(α,score);
12:  if α ≥ β then
13:    break
14:  end if
15: end for
16: while thread finished execution do
17:  α:=max(α,score);
18:  if α changed then
19:    Inform all thread about the new alpha
20:  end if
21:  if α ≥ β then
22:    Wait all other threads
23:    break
24:  end if
25: end while
26: return α
```

Για να υλοποιήσουμε την ιδέα καταρχάς δημιουργήσαμε ένα δικό μας thread pool. Το thread pool είναι αυτό το οποίο έχει δημιουργήσει από την αρχή της εκτελέσεις του προγράμματος N threads και διαμοιράζει την δουλειά που του στέλνουμε σε αυτά τα threads. Η χρήση ενός thread pool αποδείχτηκε μονόδρομος καθώς η δημιουργία νέου thread είναι πολύ ακριβή υπολογιστικά και έτσι είχαμε ένα μεγάλο overhead. Το thread pool είναι κατάλληλα δομημένο ώστε να τρέχει όπια συνάρτηση θέλουμε εμείς που δέχεται μόνο ένα όρισμα τύπου struct το οποίο το έχουμε ορίσει και πάλι εμείς.

Τα thread που δημιουργούνται μέσα στο thread pool πρέπει να μένουν ζωντανά ακόμα και όταν τελειώσουν την δουλειά που του έχουμε αναθέσει. Για να το επιτύχουμε αυτό, χρησιμοποιούμε condition variable και ένα mutex το οποίο κλειδώνει την ουρά στην οποία αποθηκεύονται οι δουλειές που αναθέτουμε στο thread pool. Με την άφιξη κάθε νέας δουλειάς και με την εισαγωγή της στην ουρά, ενημερώνονται όλα τα thread ώστε να ξυπνήσουν και όποιο προλάβει να ξεκινήσει να εκτελείτε.

Αυτό που παρατηρήσαμε άμεσα ήταν ότι στην παράλληλη εκδοχή του αλγορίθμου είναι απαραίτητο να ενημερώνονται όλα τα threads για οποιαδήποτε αλλαγή γίνετε στα alpha και στα beta. Παρατηρήσαμε ότι στην πρώτη έκδοση που δεν το είχαμε υλοποιήσει αυτό, πήραμε αρνητικό speedup ενώ μόλις το υλοποιήσαμε η διαφορά ήταν πολύ μεγάλη. Ο λόγος που γινόταν αυτό ήταν διότι μπορεί ένα thread να έβρισκε τα ιδανικά alpha beta και όλα τα άλλα thread να τερμάτιζαν άμεσα αν τα γνώριζαν.

Transposition Table

Ο επιλεγμένος αλγόριθμος όμως κρύβει καλά μια παραδοχή, ότι το αριστερό παιδί έχει την βέλτιστη τιμή ή τουλάχιστον την προσεγγίζει αρκετά. Για να το δούμε αυτό καλύτερα πρέπει να εξετάσουμε δυο περιπτώσεις.

Η πρώτη περίπτωση είναι το αριστερό παιδί να είναι το βέλτιστο. Σε αυτή την περίπτωση το main thread θα ψάξει μέχρι το τέλος του υπό-δέντρου (χρησιμοποιώντας κάποια threads στα επόμενα βάθη). Αφού το βέλτιστο βρίσκεται σε αυτό το υπό-δέντρο, σύμφωνα με την θεωρία που βρίσκεται πίσω από το alpha beta pruning, το κόστος να ψάξουμε αυτό το υπό-δέντρο θα είναι μεγαλύτερο από οποιοδήποτε άλλο δέντρο ίδιου μεγέθους. Μάλιστα, αφού έχει εντοπισθεί η βέλτιστη κίνηση, η δουλειά των threads θα είναι αρκετά μικρότερη, ώστε να θεωρήσουμε ότι θα τελειώσουν σε σχετικά κοντινούς χρόνους.

Στην δεύτερη περίπτωση όπου το αριστερό παιδί δεν είναι το βέλτιστο, μπορεί να επιστραφεί ένα αρκετά μεγάλο παράθυρο. Αυτό μεταφράζετε σε πολύ λιγότερα cut-offs, δηλαδή πολύ δουλειά για τουλάχιστον ένα thread, άρα κακό balancing και μεγάλο idle time.

Για αυτό έπρεπε κάπου να αποθηκεύονται για κάποιες θέσεις οι βέλτιστες κινήσεις από προηγούμενες αναζητήσεις, ώστε να αναζητούνται πρώτες. Επίσης, όλοι οι αλγόριθμοι αναζήτησης σε δέντρα παιχνιδιών στα οποία συναντάμε πολλές φορές την ίδια θέση παίζοντας διαφορετικές κινήσεις (transpositions), επιταχύνονται αισθητά με την χρήση του Transposition Table.

Το Transposition Table είναι ένα hashtable στο οποίο με βάση ένα hash που προσδιορίζεται από την κατάσταση του παιχνιδιού αποθηκεύονται διάφορες πληροφορίες για εκείνη την θέση. Με βάση τον τρόπου που στήσαμε το board, αποφασίσαμε ότι το hash θα έχει μέγεθος 64bit και θα είναι μοναδικό για κάθε θέση, αλλά δεν θα συμπεριλαμβάνει πληροφορία για το ποιος παίχτης είναι ο επόμενος. Επίσης σε 32bit μπορούσε να αποθηκευτεί όλη η πληροφορία που θέλαμε να κρατάμε για κάθε θέση, έτσι αποφασίσαμε να δημιουργήσουμε προσεκτικά το Transposition Table ώστε να αποθηκεύει δυο entries ανά θέση, ένα για κάθε παίχτη που έχει σειρά, αλλά στην ίδια θέση (ίδιο πλήρες hash). Σε αυτόν τον χώρο (32bit) αποθηκευόταν πληροφορία για το βάθος στο οποίο είναι υπολογισμένα τα αποτελέσματα του entry, ένα score, το οποίο μπορούσε να ήταν πάνω όριο, κάτω όριο, ή ακριβές score, άρα και η πληροφορία για το τι είδους ήταν και την καλύτερη κίνηση killer move[1]. Μαζί με κάθε ζεύγος από entries είναι απαραίτητο να αποθηκευτεί και το hash

Το μέγεθος του Transposition Table ορίστηκε για τις περισσότερες δοκιμές σε 0x7fffff hash entries (ή 0xfffffe entries), αλλά αλλάζει πολύ εύκολα. Το κλειδί για την αναζήτηση μέσα στο Transposition Table ήταν το hash mod #hashEntries.

Από το Transposition Table με κατάλληλο πρωτόκολλο αντικατάστασης και κάποιες άλλες προσεκτικές ενέργειες, μπορεί στο τέλος της αναζήτησης να εξαχθεί η τελική κίνηση και μάλιστα ένα μέρος της βέλτιστης σειράς κινήσεων.

Simultaneous Access to Transposition Table

Κάθε κόμβος του δέντρου αντλεί δεδομένα με το που ξεκινάει και σώζει λίγο πριν βγει, εκτός αν είναι leaf node. Αυτό καθιστά δύσκολη την πρόσβαση στο Transposition Table μέσω κάποιου μηχανισμού με locks. Μάλιστα, πειραματικά μπορεί να δειχτεί ότι στις περισσότερες περιπτώσεις είναι πολύ μικρό το

[1] όρος από τις σκακιστικές μηχανές, ονομάζεται έτσι επειδή δεδομένου ότι θα ψαχτεί σχετικά νωρίς θα σκοτώσει αρκετές αναζητήσεις νωρίς λόγω cut-offς από το παράθυρο που θα προκαλέσει

κέρδος ή σε ακραίες περιπτώσεις μπορεί να είναι καλύτερο να μην υπάρχει το Transposition Table από το να γίνεται η πρόσβαση με locks.

Με βάση αυτή την παρατήρηση, οι σκακιστικές μηχανές χρησιμοποιούν μια αρκετά ενδιαφέρουσα lock-less προσέγγιση η οποία υιοθετήθηκε και στο project. Αφού τελικώς θα αποθηκευτούν δυο 64bit δεδομένα (hash + data), θα αποθηκευτεί το hash XORed με τα data αντί του hash με τα data. Έτσι, όταν θα γίνει ανάγνωση από το Transposition Table των δυο τιμών, ($A \triangleq hash_{TT} \oplus data_{TT}, B \triangleq data_{TT}$) μπορούν να υπολογιστούν τα $hash_{TT} \equiv A \oplus B, data_{TT} \equiv A$. Μετά, το $hash$ θα συγκριθεί με το $A \oplus B$ το οποίο περιέχει πληροφορία από όλο το entry. Αν κάποιο άλλο/άλλα thread είναι στην διαδικασία να γράψουν το entry αλλά δεν την έχουν ολοκληρώσει, τότε η πιθανότητα το hash να ταιριάξει με το $A \oplus B$ είναι πολύ μικρή, καθώς θα πρέπει $hash = hash_{TT} \oplus data_{T1_{TT}} \oplus data_{T2_{TT}}$. Σε αυτή την περίπτωση (που δεν είχαμε ταίριασμα), αν κάνουμε ανάγνωση η καλύτερη μεθοδολογία λέει απλώς συνεχίζουμε σαν να είχαμε miss. Προφανώς αν κάποιο άλλο thread έχει γράψει τα δεδομένα του πλήρως και μας ταιριάζουν, ο έλεγχο $A \oplus B = hash$ θα βγει αληθής και θα γίνει κανονικά η ανάγνωση. Η όλη διαδικασία και στις δυο περιπτώσεις είναι lock-less αλλά και thread safe.

Evaluation function

Η συνάρτηση αξιολόγησης που φτιάξαμε είναι αρκετά απλή και δημιουργήθηκε κυρίως από την δικιά μας πίρα παίζοντας το παιχνίδι. Από τα πρώτα παιχνίδια καταλάβαμε ότι για να σχηματίσουμε όσο το δυνατών μεγαλύτερο tile πρέπει να 'κολλήσουμε' το μεγαλύτερο tile που έχουμε την εκάστοτε χρονική στιγμή σε μία γωνία. Ο λόγος που γίνετε αυτό είναι διότι συνήθως το μέγιστο tile δεν μας εξυπηρετεί στην διάρκεια του παιχνιδιού και θέλουμε να έχουμε χώρο να ενώνουμε όλα τα άλλα μας tiles.

Βασιζόμενοι στην παραπάνω ιδέα βρήκαμε ότι πρέπει γενικά τα tiles μας να είναι στοιχισμένα διαδοχικά από το μεγαλύτερο στο μικρότερο σε μία λογική αλυσίδας. Η λογική της αλυσίδας βοηθάει στο να κάνουμε collapse όλα τα tiles σε ένα μεγάλο και μπορούμε να δούμε ένα παράδειγμα αλυσίδας στην εικόνα 1. Σε αυτή την εικόνα είναι εύκολο να δούμε ότι σε 4 κινήσεις θα έχουμε δημιουργήσει το tile 256. Επίσης είδαμε ότι δύο ακόμη παράγοντες που επηρεάζουν σχετικά λίγο το παιχνίδι είναι ο αριθμός τον ελεύθερων tiles που έχουμε καθώς και την ποικιλία από tiles.

Το δύσκολο κομμάτι το οποίο ακόμα είναι υπό υλοποίηση είναι να βρεθούν τα κατάλληλα βάρη για αυτές τις τιμές. Εμείς όπως θα δείξουμε και στα results φτάσαμε πολύ κοντά στο να δημιουργήσουμε το tile 4096 και πιστεύουμε ότι με λίγο tune του evaluation function θα μπορέσουμε να φτάσουμε και πιο μεγάλα tiles. Τέλος να πούμε ότι όταν ο 'παίκτης' βλέπει τέλος παιχνιδιού, το score που γυρνάει η evaluation function είναι το μεγαλύτερο tile που έχει φτιάξει μέχρι εκείνη την στιγμή. Ουσιαστικά εμάς μας νοιάζει μόνο το μεγαλύτερο tile και όχι το συνολικό score του παιχνιδιού.

Compile and Run

Για να γίνει build το project, στον κύριο φάκελό του, εκτελείται make all και μετά σε τρία διαφορετικά terminal τα:

- *./parallel_server* 1
  για να τρέξει ο game server ο οποίος περιμένει να συνδεθούν οι παίχτες και παρουσιάζει το παιχνίδι με γραφικά
- *./parallel_client placer*
  στο terminal που θέλετε να βρίσκεται ο placer
- *./parallel_client normal*
  στο terminal που θέλετε να βρίσκεται ο normal, θα εμφανίζονται οι έξοδοι του normal και τα αποτελέσματα ανά βάθος του iterative deepening

Σημειώνεται ότι χρειάζεται compiler που υποστηρίζει C++11 και όλες οι δοκιμές έχουν γίνει με τον g++ 4.8.2.

Results

Πριν προχωρήσουμε στην παρουσίαση των αποτελεσμάτων πρέπει να πούμε ότι στα παιχνίδια που έπαιζε ο παίχτης μας, ο αντίπαλος (ο παίχτης που τοποθετεί τα νέα tiles στο board) ήταν πάντα ο random.

Είδαμε ότι όταν βάζαμε τον negascout να παίζει σαν αυτόν τον παίχτη τότε δεν πετυχαίναμε μεγάλο score διότι είναι πολύ εύκολο να εμποδίσεις τις κινήσεις του normal παίχτη.

Όπως αναφέραμε και πιο πάνω καταφέραμε και φτιάξαμε το tile 2048 και φτάσαμε και πολύ κοντά στο 4096. Το ενδιαφέρον αποτέλεσμα όμως δεν είναι το highest tile που φτάσαμε αλλά το ότι όταν ο negascout έτρεχε με την παράλληλη έκδοση το βάθος που ψάχναμε έφτασε κατά μέσο όρο να είναι το 23 ενώ όταν έτρεχε σειριακά έφτανε κατά μέσο όρο μέχρι 18. Η διαφορά αυτή είναι αρκετά μεγάλη μιας και με κάθε αύξηση του βάθους το search space αυξάνετε εκθετικά. Επίσης πρέπει να προσθέσουμε ότι για κάθε κίνηση είχε στην διάθεση του μόνο 0.8 second μιας και όπως βρήκαμε από διάφορα στατιστικά αυτή είναι η μέση ταχύτητα που παίζει ένα έμπυρος παίκτης στα πρώτα στάδια του παιχνιδιού.

Γενικά το επιπλέον βάθος που πήραμε με την βοήθεια της παραλληλοποίησης βοήθησε αρκετά στο να φτάνουμε καλύτερο τελικό score. Στην βιβλιογραφία υπάρχουν και άλλοι τρόποι να τρέξει παράλληλα ο negascout άλλα το κύριο πρόβλημα με αυτούς τους αλγορίθμους είναι η δυσκολία υλοποίησης τους στην πράξη. Γενικά θα είχε ενδιαφέρον να εξετάσει κανείς το τελικό βάθος που θα μπορούσε να φτάσει κάποιος με μια νέα υλοποίηση!