ΠΑΡΑΛΛΗΛΗ ΕΠΕΞΕΡΓΑΣΙΑ

ΑΚΑΔΗΜΑΙΚΟ ΕΤΟΣ 2011/2012

- ***Ομάδα 03***
  Φίλιππος Φιλίππου - 4662
  Γαβριήλ Δάφνη - 4697
  Γεώργα Ευγενία - 4700
  Μονοπάτης Δημήτριος - 4776

Αρχικά, μας ζητείται να γίνει χρήση των POSIX Threads.
Τα threads (νήματα) αποτελούν το καθένα τους μία ανεξάρτητη ροή εντολών, οι οποίες εφαρμόζονται αφού χρονοπρογραμματιστούν από το λειτουργικό σύστημα. Αν παρομοιάσουμε κάθε νήμα με μία διαδικασία που τρέχει ανεξάρτητα από το κύριο πρόγραμμα και θεωρήσουμε ότι το κύριο πρόγραμμα εμπεριέχει έναν αριθμό τέτοιων διαδικασιών που είναι δυνατό να τρέξουν ταυτοχρόνως, τότε έχουμε την ιδέα ενός multi-threaded προγράμματος.
Τα νήματα αποκτούν το πλεονέκτημα να μπορούν να προγραμματιστούν και να τρέξουν σαν ανεξάρτητες οντότητες κυρίως εξαιτίας του γεγονότος ότι διπλασιάζουν μονάχα τους κύριους πόρους που τα καθιστούν να υπάρχουν σαν εκτελέσιμος κώδικας.
Ο λόγος χρήσης των PThreads είναι η ικανότητά τους να προσφέρουν τις ίδιες υπηρεσίες με μία άλλη διαδικασία, όμως με πολύ λιγότερο overhead. Η διαχείριση των PThreads απαιτεί λιγότερους πόρους από το σύστημα. Αυτός είναι και ο καλύτερος λόγος για να τα προτείνουμε στον παράλληλο προγραμματισμό. Το API των PThreads εμπεριέχει υπορουτίνες οι οποίες έχουν συναρτήσεις που δουλεύουν πάνω στα νήματα αυτά, αλλά υπάρχουν και συναρτήσεις που αφορούν τον συγχρονισμό.

Έπειτα καλούμαστε να διαμορφώσουμε τον κώδικα μας κάνοντας χρήση του προτύπου OpenMP.
Το OpenMP είναι ένα API το οποίο δημιουργήθηκε από κατασκευαστές υλικού και λογισμικού. Τα αρχικά αντιστοιχούν στη φράση "Open Specifications for Multi Processing".
Αυτό το πρότυπο για παράλληλο προγραμματισμό δίνει στον χρήστη τη δυνατότητα να αναπτύξει παράλληλα προγράμματα για συστήματα κοινής μνήμης, τα οποία είναι ανεξάρτητα αρχιτεκτονικής και έχουν μεγάλη ικανότητα κλιμάκωσης.
Έτσι, ο χρήστης μπορεί πια να δώσει εντολές και οδηγίες στον μεταγλωττιστή για να ορίσει μέρη του προγράμματος που επιθυμεί να εκτελεστούν παράλληλα, να ορίσει σημεία συγχρονισμού και άλλα πιο εύκολα.

Πολυνηματική Επεξεργασία

Στην υλοποίηση με νήματα προσπαθήσαμε το υπολογιστικά χρονοβόρο κομμάτι να το αναθέσουμε σε ξεχωριστά νήματα έτσι ώστε αν το σύστημα στο οποίο τρέχουμε το πρόγραμμα μπορεί να κάνει αποδοτική διαχείριση των νημάτων να μειώσουμε τον χρόνο εκτέλεσης του προγράμματος εκμεταλλευόμενοι αυτή την παραλληλοποίηση. Με τα threads γίνεται αντιγραφή και διαχωρισμός μόνο των απαραίτητων δεδομένων και κοινή διαμοίραση των υπολοίπων.
Όπως παρατηρήσαμε, το υπολογιστικά χρονοβόρο κομμάτι του δοθέντος κώδικα, ήταν ο υπολογισμός του πίνακα βαθμολόγησης H, το οποίο και παραλληλοποιήσαμε ώστε να επιτύχουμε επιτάχυνση στην εξαγωγή των τελικών αποτελεσμάτων. Αντίθετα το κομμάτι που υλοποιεί το back tracking δεν μπορεί να παραλληλοποιηθεί λόγω των εξαρτήσεων στον υπολογισμό της διαδρομής.

POSIX Threads

Αρχικά στην συνάρτηση **main**, προσθέσαμε ένα επιπλέον **argument**, για να εισάγουμε το πλήθος των threads.

Έπειτα με την κλήση **pthread_create()** μέσα σε μια δομή επανάληψης κατασκευάζουμε όσα νήματα ορίζει η μεταβλητή THR_NUMB, δηλαδή όσα ορίσαμε κατά την εκτέλεση. Κάθε νήμα που δημιουργείται θα καλεί τη συνάρτηση **calculate_H** με όρισμα i, όπου **i** είναι ο αριθμός του νήματος.

Τέλος για τον ομαλό τερματισμό του προγράμματος το κύριο νήμα του προγράμματος μας θα πρέπει να περιμένει τα υπόλοιπα νήματα που δημιουργήθηκαν να τελειώσουν. Αυτό το πετυχαίνουμε με την κλήση **pthread_join()** η οποία περιμένει όλα τα άλλα νήματα να τερματίσουν πριν προχωρήσει η εκτέλεση του υπόλοιπου σειριακού κώδικα της main.

Στο σώμα της **calculate_H**, μετατρέψαμε τον αρχικό κώδικα που υπολόγιζε τον πίνακα βαθμολόγησης , έτσι ώστε να ακολουθεί το μοντέλο του “κυματομέτωπου” (wave front). Επίσης λάβαμε υπόψιν μας και τις εξαρτίσεις των δεδομένων. Ανάλογα το πλήθος των νημάτων που έχουμε,χωρίζουμε τους υπολογισμούς.
Για να υπολογίσουμε τα στοιχεία του πίνακα ξεκινάμε από το πάνω αριστερά στοιχείο και κάθε φορά υπολογίζουμε τα στοιχεία κάθε διαγωνίου μέσω της παραλληλοποίησης. Κάθε φορά δηλαδή μετακινούμαστε μία διαγώνιο δεξιά και την υπολογίζουμε μοιράζοντας την στα threads.

Αρχικά υπολογίζουμε το πλήθος των στοιχείων της διαγωνίου που θέλουμε να παραλληλοποιήσουμε (**plithos**). Εάν ο πίνακας έχει περισσότερες γραμμές από στήλες τότε το πλήθος αυξάνεται κατά ένα μέχρι τη μέγιστη διαγώνιο, παραμένει σταθερό μέχρι κάποιο σημείο και από εκεί και πέρα μειώνεται. Αντίθετα, αν ο πίνακας έχει περισσότερες στήλες από γραμμές ή ίσες, τότε το πλήθος αυξάνεται κατά ένα μέχρι τη μέγιστη διαγώνιο, παραμένει σταθερό μέχρι κάποιο σημείο και από εκεί και πέρα μειώνεται.

Αν το πλήθος των στοιχείων είναι μικρότερο του αριθμού των νημάτων τότε χρησιμοποιούμε νήματα (**num_of_thr**) όσα και το πλήθος, αλλιώς όσα νήματα έχουμε διαθέσιμα.

Για να βρούμε από ποια στήλη βρίσκεται το πρώτο στοιχείο κάθε νήματος (**start**): ***start = s + id*(plithos/num_of_thr)*** με **s** η πρώτη γραμμή της διαγωνίου, και **id** ο αριθμός του νήματος.
Για να βρούμε σε ποια στήλη βρίσκεται το τελευταίο στοιχείο κάθε νήματος(**end**): Αν είναι το τελευταίο thread, τότε παίρνει μέχρι και το τελευταίο στοιχείο της διαγωνίου ***end = s + plithos***, διαφορετικά ***end = s + (id+1)*(plithos/num_of_thr)***.
Για να βρούμε σε ποια γραμμή βρίσκεται το πρώτο στοιχείο κάθε νήματος(**k**): ***k=j-id*(plithos/num_of_thr)*** με **j** η πρώτη στήλη της διαγωνίου και **id** ο αριθμός του νήματος.

Εκτελείται ο αλγόριθμος για τον υπολογισμό του **H** των αντίστοιχων στοιχείων. Για ευκολία δική μας, έχουμε μεταφέρει τη συνάρτηση **find_array_max** που δεν επιβαρύνει τον παραλληλισμό.

Πριν ξεκινήσει ο υπολογισμός της επόμενης διαγωνίου περιμένουμε όλα τα νήματα να συγχρονιστούν μέσω της ***pthread_barrier_wait(&barrier)***.

OpenMP

Με την χρήση του OpenMP καταφέραμε, κάνοντας ελάχιστες αλλαγές σε σχέση με την ακολουθιακή έκδοση του κώδικα, να παραλληλοποιήσουμε το πρόγραμμα και να βρούμε καλύτερα αποτελέσματα.

Το πόσα threads θα δημιουργηθούν καθορίζεται όπως και στα posix threads, μέσω της γραμμής εντολών.

Ακολουθώντας την λογική για τον υπολογισμό των στοιχείων του πίνακα, σε κάθε διαγώνιο υπολογίζουμε το πλήθος στοιχείων (**plithos**) όπως πριν και παραλληλοποιούμε τον υπολογισμό τους.

Επειδή πλέον δεν χρησιμοποιούμε συνάρτηση έχουμε κοινές μεταβλητές από τις οποίες η k θέλουμε να είναι private ώστε κάθε νήμα να έχει το δικό του αντίγραφο. Καλούμε την εντολή ***#pragma omp parallel num_threads (THR_NUMB) private(k)*** για τη δημιουργία των νημάτων την ***#pragma omp for schedule(static)*** για την ισότιμη διαμοίραση της διαγωνίου στα threads.

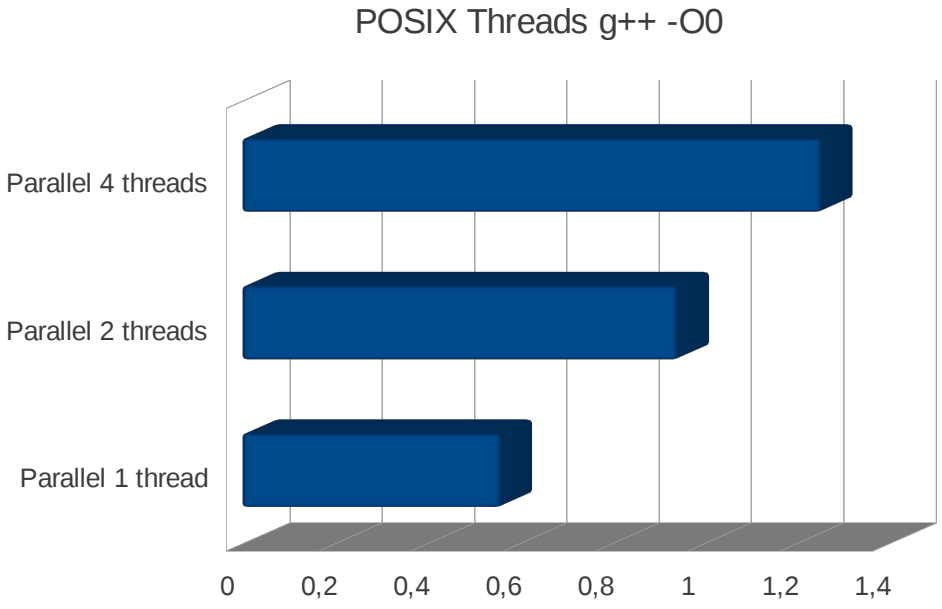
Το **k** τώρα, μας δείχνει σε ποια στήλη βρίσκεται το κάθε thread, κατά τον υπολογισμό των στοιχείων που του αντιστοιχούν και υπολογίζεται από τον τύπο ***k=j+s-i*** όπου **j** είναι η πρώτη στήλη της διαγωνίου που βρισκόμαστε, **s** είναι η πρώτη γραμμή από όπου ξεκινά το thread και **i** είναι η γραμμή που βρίσκεται το κάθε thread κατά τον υπολογισμό των στοιχείων.

Πριν ξεκινήσει ο υπολογισμός της επόμενης διαγωνίου περιμένουμε όλα τα νήματα να συγχρονιστούν μέσω της ***#pragma omp barrier***.

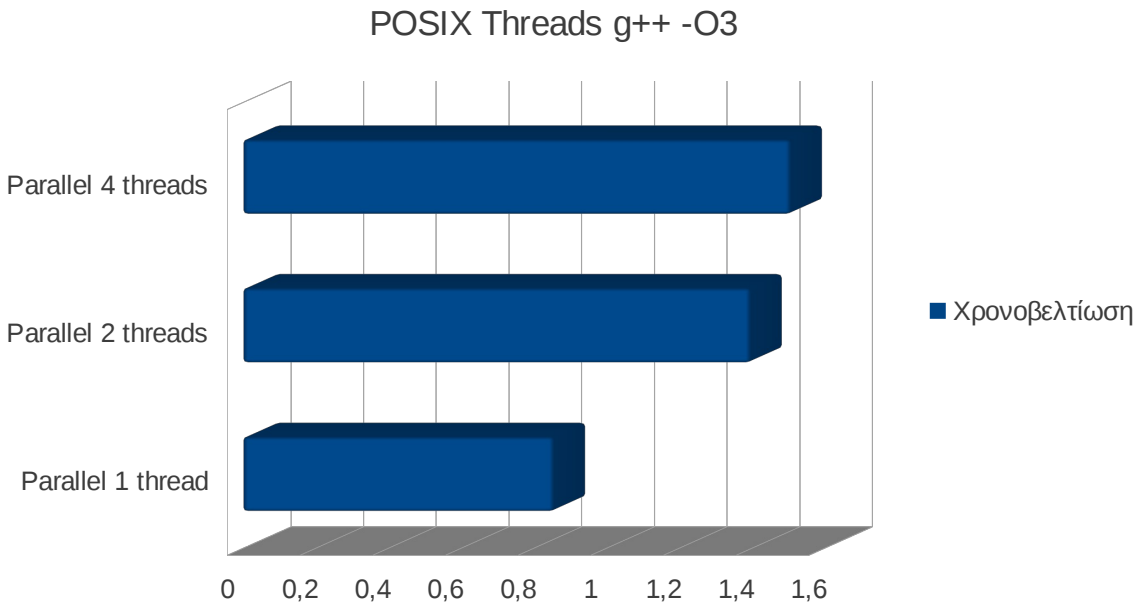
Πίνακες Χρόνου Εκτέλεσης και Διαγράμματα Χρονοβελτίωσης

Καταρχάς δοκιμάσαμε όλες τις υλοποιήσεις μας για τα μικρότερα αρχεία,χωρίς κανένα πρόβλημα.
Ωστόσο για τα ζητούμενα αρχεία δεν καταφέραμε στο μηχάνημα που διαθέταμε να το τρέξουμε επιτυχώς,για αυτό κι εμείς δημιουργήσαμε δυο δικά μας αρχεία μεγέθους 13.000 και 15.000 πλήθος βασικών συστατικών στοιχείων ενός νουκλεϊκού οξέος.

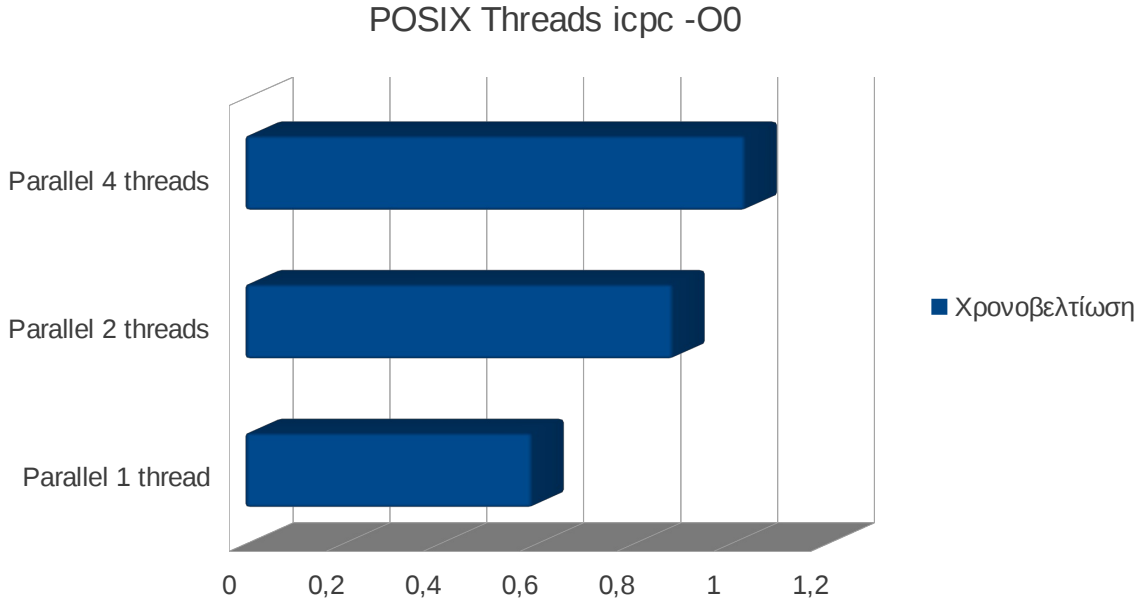
POSIX Threads g++ -O0	seconds
Σειριακό	13,51
Parallel 1 thread	24,12
Parallel 2 threads	14,32
Parallel 4 threads	10,77



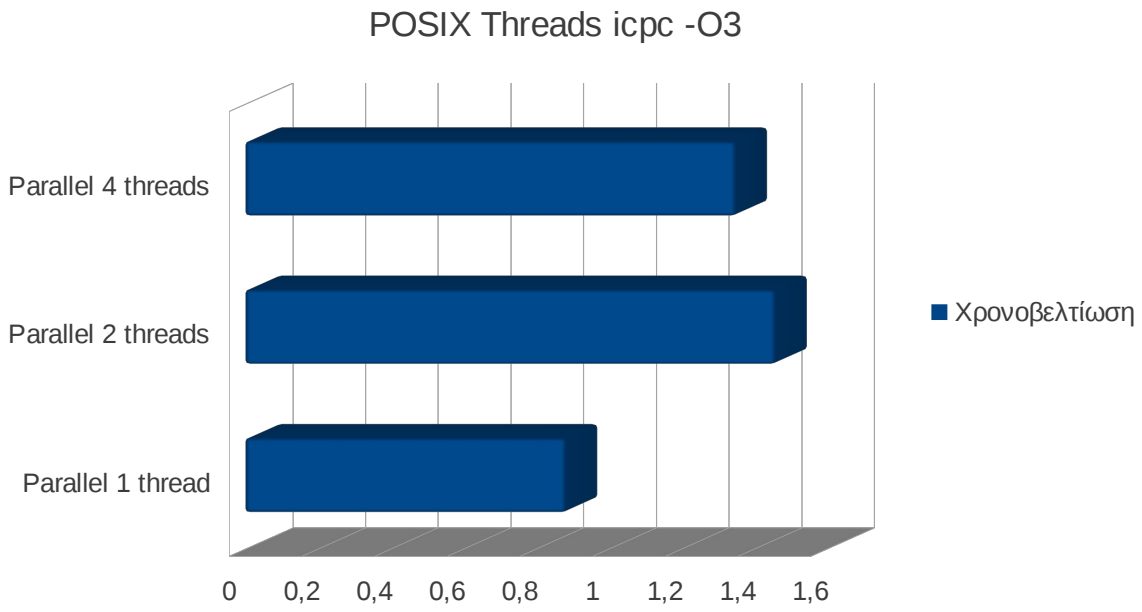
POSIX Threads g++ -O3	seconds
Σειριακό	13,51
Parallel 1 thread	15,81
Parallel 2 threads	9,68
Parallel 4 threads	8,97



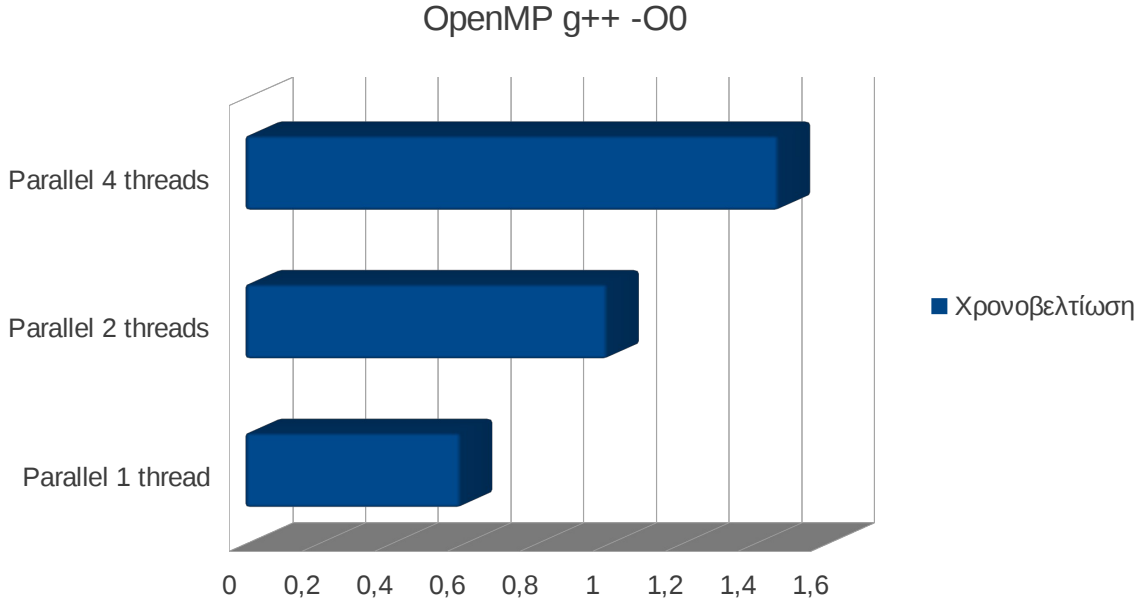
POSIX Threads icpc -O0	seconds
Σειριακό	13,51
Parallel 1 thread	22,79
Parallel 2 threads	15,29
Parallel 4 threads	13,08



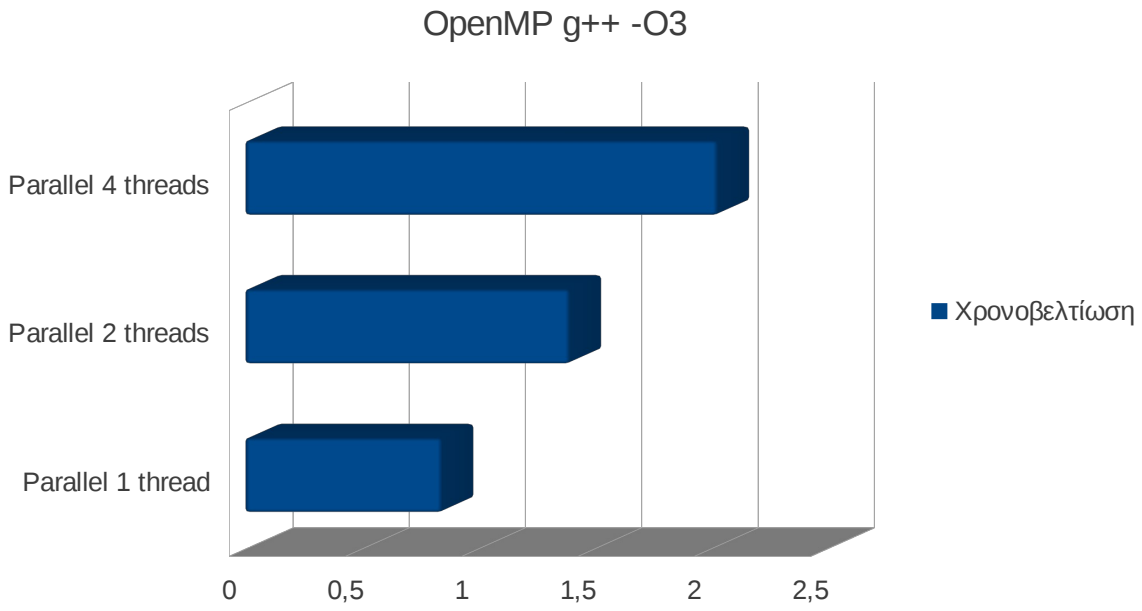
POSIX Threads icpc -O3	seconds
Σειριακό	13,51
Parallel 1 thread	15,31
Parallel 2 threads	9,26
Parallel 4 threads	10,01



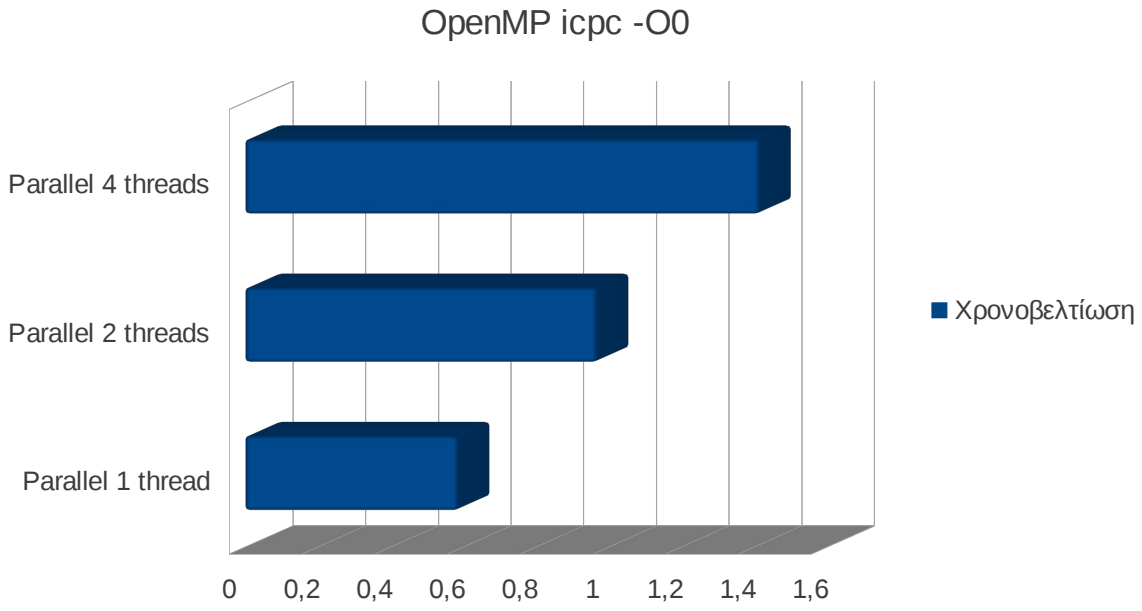
OpenMP g++ -O0	seconds
Σειριακό	13.51
Parallel 1 thread	22.76
Parallel 2 threads	13.55
Parallel 4 threads	9.2



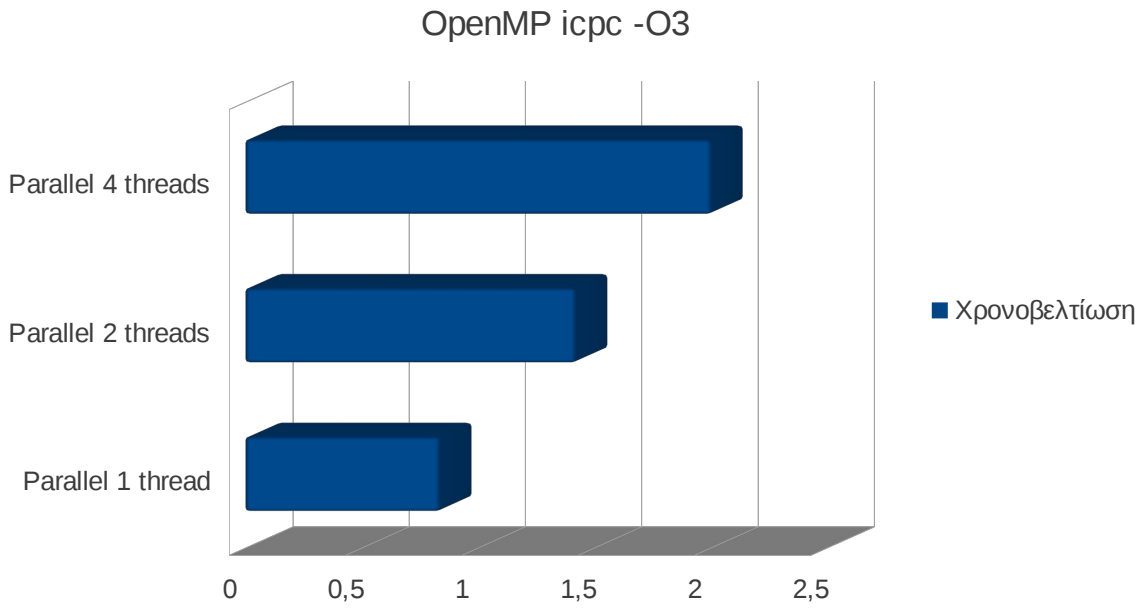
OpenMP g++ -O3	seconds
Σειριακό	13.51
Parallel 1 thread	15.97
Parallel 2 threads	9.68
Parallel 4 threads	6.65



OpenMP icpc -O0	seconds
Σειριακό	13,51
Parallel 1 thread	23,07
Parallel 2 threads	13,96
Parallel 4 threads	9,55



OpenMP icpc -O3	seconds
Σειριακό	13.51
Parallel 1 thread	16.12
Parallel 2 threads	9.5
Parallel 4 threads	6.74



Από τις μετρήσεις μας παρατηρούμε ότι σε σχέση με το σειριακό κώδικα που μας δόθηκε στο **1 thread** παρατηρούμε ότι **αυξάνεται ο χρόνος** εκτέλεσης για όλες τις υλοποιήσεις ,αυτό είναι φυσικό καθώς υπάρχει **μεγαλύτερη πολυπλοκότητα**.
Βλέπουμε λοιπόν ότι στο 1 thread έχουμε χρονοβελτίωση < 1 (κακό..), ενώ στα 2 και 4 thread χρονοβελτίωση > 1 (καλό!!).
Παρατηρούμε ότι ο **compiler της intel** κάνει **καλύτερους χρόνους** συγκριτικά με τον gcc compiler.
Μεγάλη επίπτωση στους χρόνους έχουμε όταν μεταγλωττίζουμε τις υλοποιήσεις χωρίς βελτιστοποιήσεις σε σχέση με τις μέγιστες βελτιστοποιήσεις.
Στα παραδοτέα συμπεριλάβαμε και ένα αρχείο **Makefile** το οποίο μπορείτε να χρησιμοποιήσετε για να τρέξετε τον κώδικα μας. Επίσης συμπεριλάβαμε ένα script file (**run.sh**),το οποίο εμφανίζει μόνο τους χρόνους για κάθε υλοποίηση.