# *Παράλληλη Επεξεργασία*

Ομάδα 33:

Παναγιώτης Ντενέζος 5853

Κωνσταντίνος Κανούτος 5775

Ανδρόνικος Κυριακού 5806

Θωμάς Πλούμης 5880

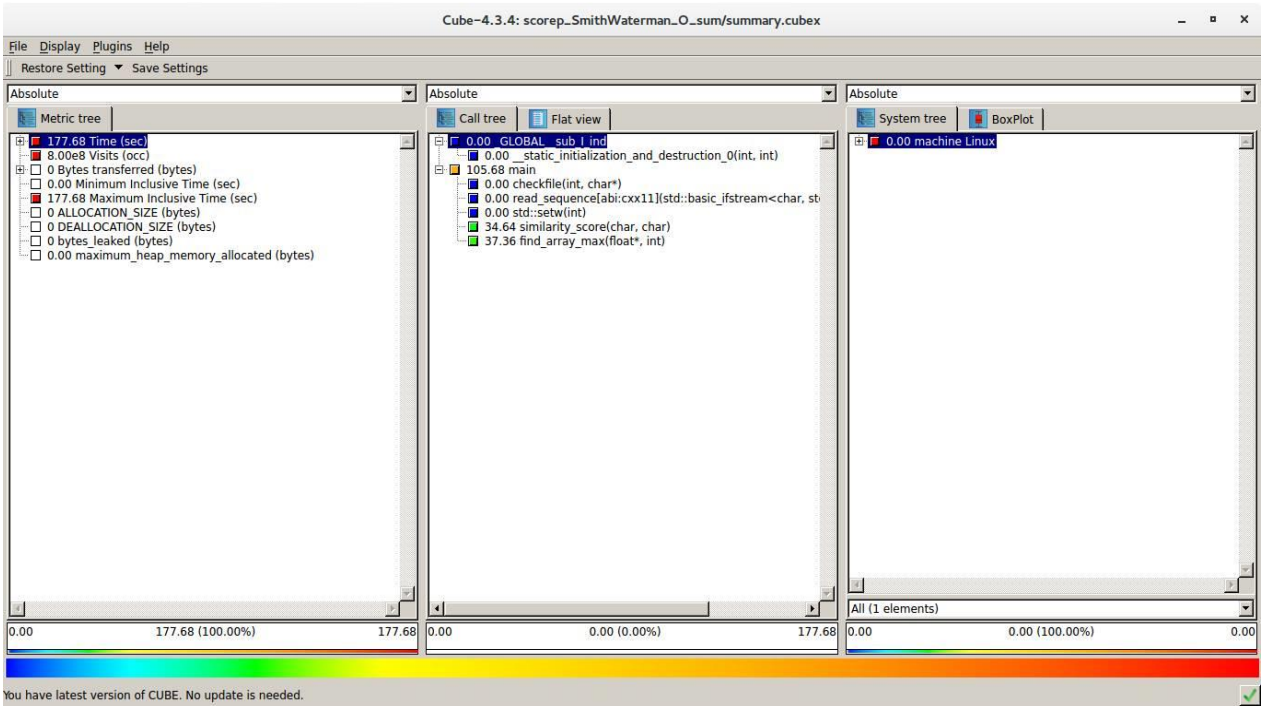
## 1. Χαρακτηριστικά Υπολογιστή

Η υλοποίηση της άσκησης έγινε σε υπολογιστή με τα εξής χαρακτηριστικά :

- CPU : Intel Core i5 4200U (3MB Cache) 64bit – 2 cores, 4 threads @1.6 GHz
- Μνήμη Ram : 4 GB
- OS : Ubuntu Gnome 15.10
- Έκδοση gcc : 5.2.1

## 2. Ανάλυση σειριακού κώδικα

Αρχικά, έγινε χρήση του εργαλείου Scalasca για να γίνουν εμφανείς οι συναρτήσεις, στις οποίες καταναλώνεται ο περισσότερος χρόνος για την εκτέλεση του προγράμματος. Για να γίνει σωστή μέτρηση των χρόνων [1] έπρεπε να γίνει instrumentation του εκτελέσιμου το οποίο και επιτεύχθη με την χρήση του προγράμματος ανάλυσης για παράλληλους κώδικες Score-P. Στην συνέχεια, έγινε ανάλυση και εξέταση (examination) με χρήση του Scalasca και τέλος οπτικοποίηση των αποτελεσμάτων με χρήση του Cube. Μετά την εκτέλεση των παραπάνω προκύπτουν τα εξής αποτελέσματα:




Εικόνα 1.1 : Χρόνοι εκτέλεσης συναρτήσεων

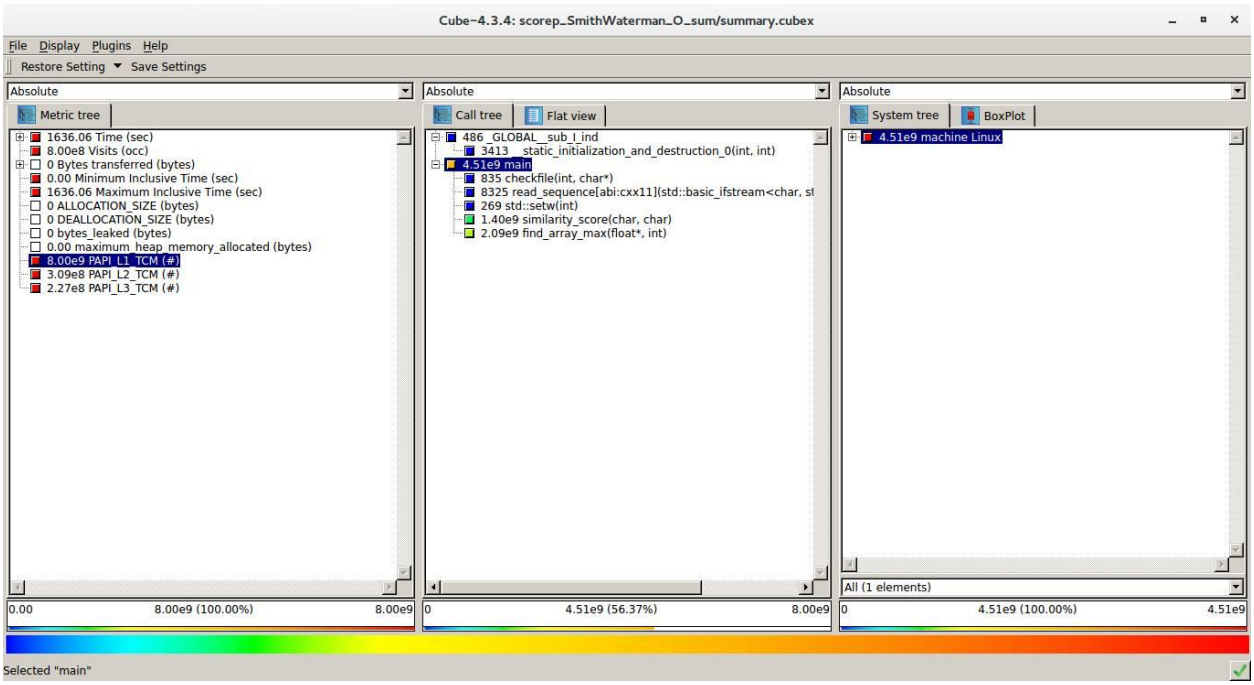
Από την Εικόνα 1.1 είναι εύκολο να παρατηρηθεί ότι οι συναρτήσεις *similarity_score()* και *find_array_max()* καταναλώνουν τον περισσότερο χρόνο εκτέλεσης του προγράμματος. Συνεπώς, στις συγκεκριμένες συναρτήσεις θα γίνει επέμβαση στον κώδικα, με σκοπό να επιτευχθεί παραλληλοποίηση.

Για περισσότερη εμβάθυνση, χρησιμοποιήθηκαν οι παρακάτω Performance Counters, ώστε να γίνει κατανοητό, γιατί οι συγκεκριμένες συναρτήσεις δεν έχουν καλή απόδοση.

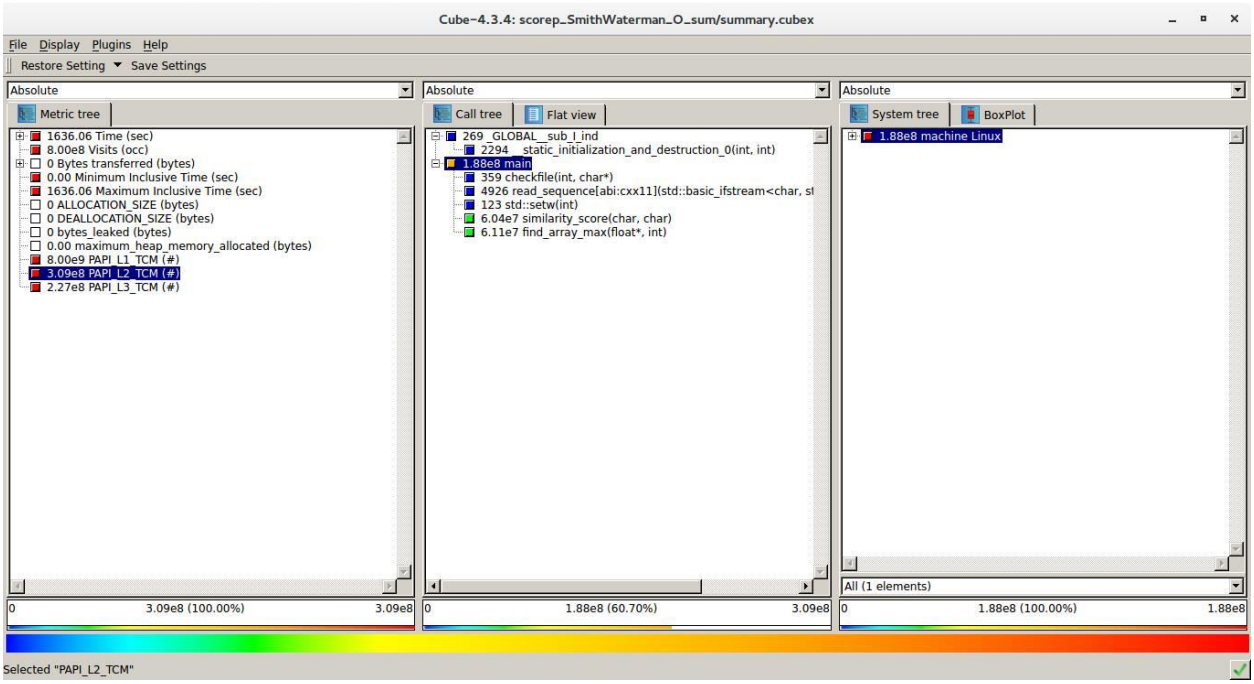
- PAPI_L1_TCM
- PAPI_L2_TCM
- PAPI_L3_TCM
- PAPI_LD_INS
- PAPI_SR_INS

Ο λόγος που γίνεται χρήση των παραπάνω Performance Counters είναι γιατί τα cache misses και η εκτέλεση των εντολών load και store είναι αυτά που κυρίως επιβαρύνουν την εκτέλεση ενός προγράμματος.

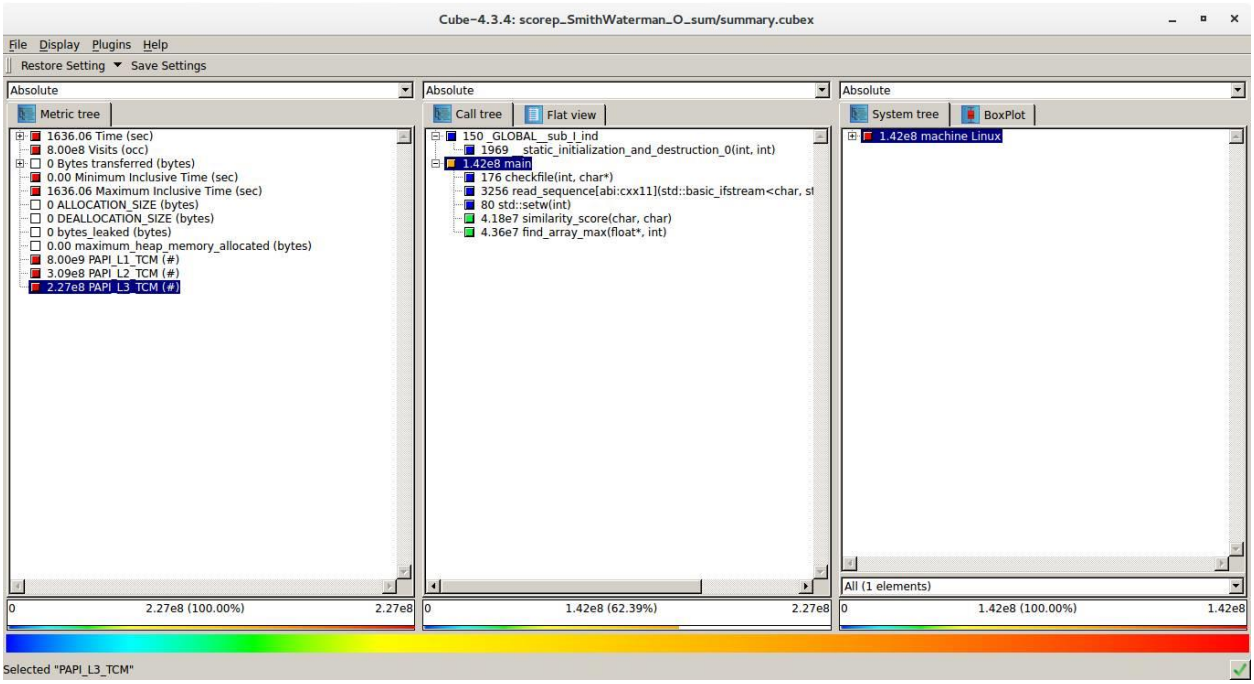
Παρακάτω απεικονίζονται για κάθε μετρητή οι τιμές των συναρτήσεων, επιβεβαιώνοντας ότι η περισσότερη καθυστέρηση γίνεται στις συναρτήσεις *similarity_score()* και *find_array_max()*.



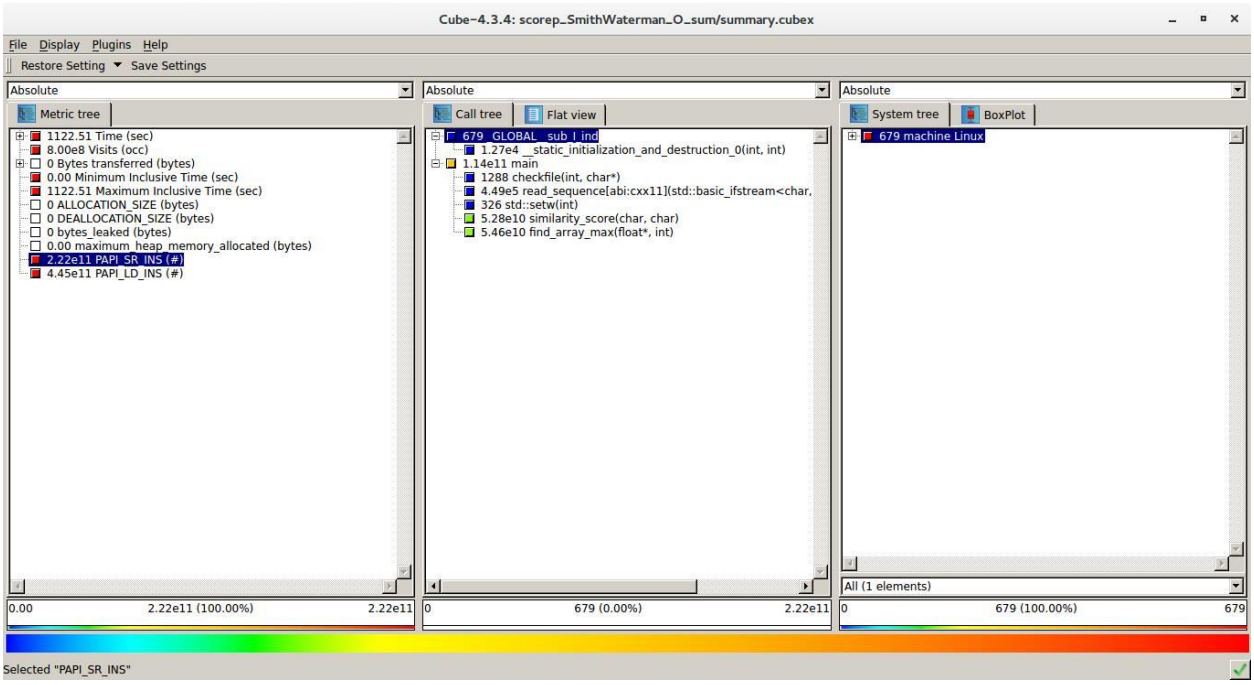

Εικόνα 1.2: Τιμές των συναρτήσεων για τον μετρητή PAPI_L1_TCM



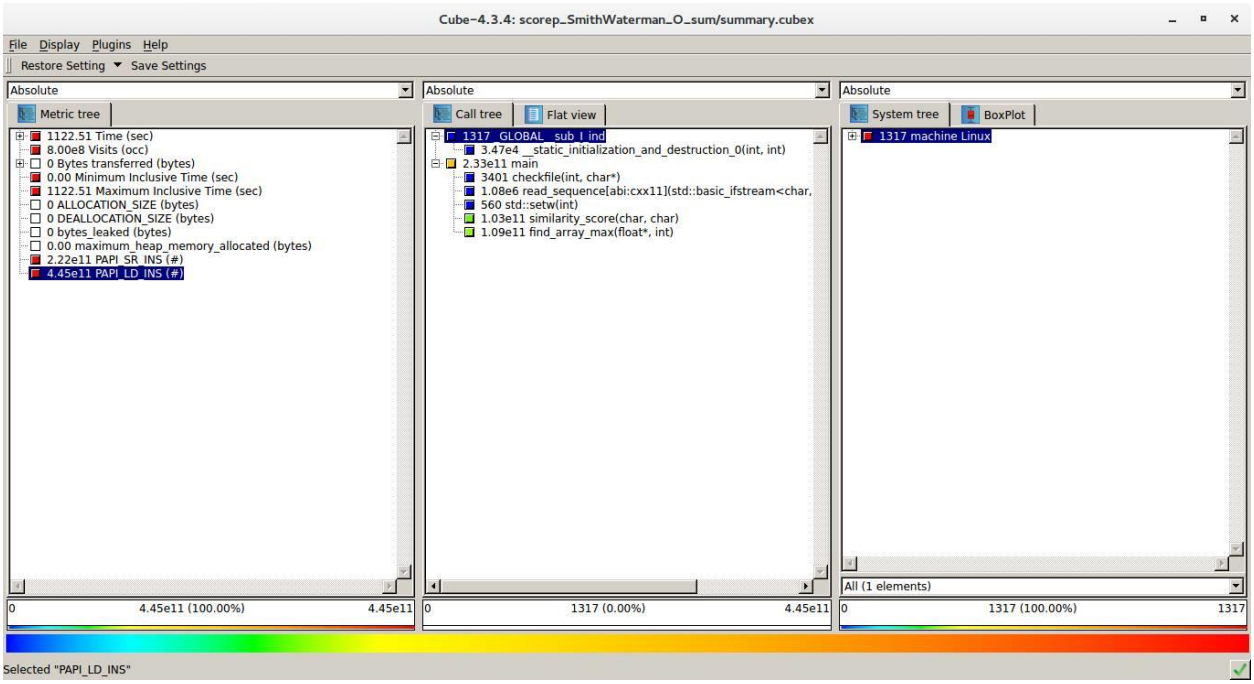

Εικόνα 1.3: Τιμές των συναρτήσεων για τον μετρητή PAPI_L2_TCM




Εικόνα 1.4: Τιμές των συναρτήσεων για τον μετρητή PAPI_L3_TCM



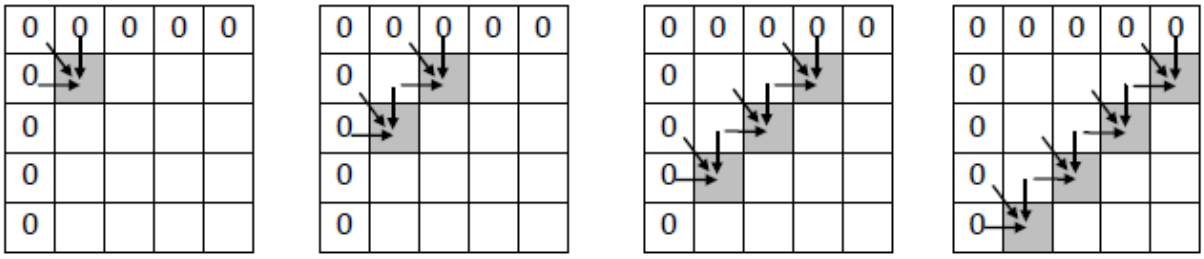

Εικόνα 1.5: Τιμές των συναρτήσεων για τον μετρητή PAPI_SR_INS




Εικόνα 1.6: Τιμές των συναρτήσεων για τον μετρητή PAPI_LD_INS

## 3. Παραλληλοποίηση με χρήση OpenMP

Με βάση τα αποτελέσματα του εργαλείου Scalasca, η παραλληλοποίηση του προγράμματος αρχικά έγινε στις συναρτήσεις *similarity_score()* και *find_array_max()*, όπου και εμφανίστηκαν οι μεγαλύτερες καθυστερήσεις. Υλοποιώντας την ιδέα αυτή, παρατηρήθηκε ότι η παραλληλοποίηση κάποιας συγκεκριμένης συνάρτησης δεν έχει ιδιαίτερη μείωση στους χρόνους εκτέλεσης, καθώς οι πράξεις που εκτελούνται μέσα στις συναρτήσεις αυτές είναι λίγες. Για το λόγο αυτό, η παραλληλοποίηση θα γίνει στη διπλή-for που διαμορφώνει τον πίνακα H. Το πρόβλημα που προκύπτει είναι ότι στις πράξεις που υπάρχουν μέσα στην διπλή-for υπάρχουν εξαρτήσεις δεδομένων. Για να εξαλειφθούν αυτές οι εξαρτήσεις τροποποιήθηκε ο κώδικας που δίνεται έτσι ώστε ο υπολογισμός των στοιχείων να γίνεται με βάση τον αλγόριθμο Wavefront. Έπειτα, η παραλληλοποίηση της διπλής-for έγινε στο νέο πρόγραμμα.



Εικόνα 2.1: Οπτικοποίηση προσπέλασης πίνακα με χρήση του αλγορίθμου Wavefront

Πιο συγκεκριμένα, καθώς ο σκοπός είναι η προσπέλαση του μητρώου H αντιδιαγώνια, τροποποιούνται κατάλληλα οι εντολές επανάληψης. Αυτό που ουσιαστικά επιτυγχάνεται είναι κάθε φορά η προσπέλαση μιας και μόνο αντιδιαγωνίου και η παραλληλοποίηση στον υπολογισμό των στοιχείων αυτής. Με αυτόν τον τρόπο όλα τα στοιχεία της κάθε αντιδιαγωνίου του H μπορούν να υπολογιστούν ανεξάρτητα, καθώς όλα τα προαπαιτούμενα στοιχεία που χρειάζονται για να επιτευχθεί ο υπολογισμός αυτός είναι ήδη γνωστά.

Για την πραγματοποίηση της παραλληλοποίησης χρησιμοποιήθηκε στην εσωτερική εντολή-for η εντολή προεπεξεργαστή *#pragma omp parallel for private(ind,temp,i,j)*. Ο λόγος που έγινε χρήση της συγκεκριμένης εντολής είναι γιατί αποσκοπεί σε παραλληλοποίηση εντολής-for *(pragma omp parallel for)* και επειδή πρέπει για τις μεταβλητές *ind,temp,i,j* το κάθε νήμα να έχει το δικό του αντίγραφο (χρήση του clause *private(ind,temp,i,j)*). Η μεταβλητή *ind,* καθώς είναι global, για να διατηρεί κάθε νήμα ξεχωριστό αντίγραφό της, προστέθηκε στην συνάρτηση *find_array_max* και η κλήση της επιλέχθηκε να γίνεται με αναφορά για λόγους συγχρονισμού.

Τέλος, πέρα από τις απαραίτητες αλλαγές στο κρίσιμο κομμάτι του κώδικα, στο οποίο υπήρχαν οι μεγάλες καθυστερήσεις, επιλέχθηκε να γίνει παραλληλοποίηση και σε άλλα σημεία του κώδικα. Τα σημεία αυτά είναι τα κομμάτια όπου γίνεται η αρχικοποίηση των πινάκων H, I_i και I_j και η εύρεση του μέγιστου score στον πίνακα H. Για την παραλληλοποίηση τους έγινε χρήση των εντολών προεπεξεργαστή *#pragma omp parallel, #pragma omp for, #pragma omp sections* και *#pragma omp section.* Η εντολή *pragma omp parallel* δηλώνει την περιοχή όπου θα γίνει η παραλληλοποίηση, η εντολή *pragma omp for* δηλώνει ότι θα γίνει παραλληλοποίηση των επαναλήψεων της εντολής-for, η εντολή *pragma omp sections* δηλώνει την περιοχή στην οποία βρίσκονται μέσα τα τμήματα κώδικα που θα δοθούν χωριστά σε κάθε νήμα και η εντολή *pragma omp section* δηλώνει το τμήμα κώδικα που θα αναλάβει το κάθε νήμα. Με τον τρόπο αυτό επιτεύχθηκε κατά την αρχικοποίηση, όταν ο αριθμός των threads το επιτρέπει, οι πίνακες να δημιουργούνται παράλληλα και έτσι να έχουμε ένα μικρό χρονικό κέρδος. Αντίστοιχα, κατά την εύρεση του μέγιστου score, εφόσον δεν υπάρχουν εξαρτήσεις ο πίνακας H μπορεί να τμηματοποιηθεί και έτσι να έχουμε χρονοβελτίωση.

## 4. Παραλληλοποίηση με χρήση OpenMP και διανυσματοποίηση

Χρησιμοποιώντας ως βάση τον κώδικα ο οποίος αναπτύχθηκε στα πλαίσια του προηγούμενου ερωτήματος (ερώτημα 3ο) έγιναν κάποιες τροποποιήσεις προκειμένου να επιτευχθεί η διανυσματοποίηση.

Αρχικά, προκειμένου να υλοποιήσουμε το ερώτημα αυτό αναζητήθηκε η κατάλληλη μέθοδος για να γίνει η διανυσματοποίηση. Η επιλογή που έγινε είναι αυτή του προτύπου OpenMP καθώς επιτυγχάνει φορητότητα και μεταφερσιμότητα του κώδικα (οι εντολές που χρησιμοποιούνται είναι ανεξάρτητες από τον τύπο, την τεχνολογία και την κατασκευάστρια εταιρία επεξεργαστών), αλλά και δίνει περισσότερες πληροφορίες στον μεταφραστή για την δομή του κώδικα. Παράλληλα, ο τρόπος (αλλά και ο λόγος) που δημιουργήθηκαν οι εντολές simd του OpenMP δίνουν την δυνατότητα να επιτύχουμε τα επιθυμητά αποτελέσματα με ελάχιστες αλλαγές στον ήδη υπάρχοντα κώδικα.

Για την επίτευξη της διανυσματοποίηση, όμως, υπήρχαν κάποιες προϋποθέσεις οι οποίες έπρεπε να τηρηθούν. Αρχικά, διανυσματοποίηση δεν γίνεται σε κώδικα ο οποίος έχει εξαρτήσεις δεδομένων. Όπως φαίνεται και παραπάνω, αυτό επιτεύχθη με χρήση του αλγορίθμου Wavefront. Ύστερα, σύμφωνα με το specification του OpenMP 4.5, [2] οι επαναλήψεις που βρίσκονται σε κομμάτι κώδικα το οποίο διανυσματοποιείται πρέπει να βρίσκονται σε κανονική μορφή (canonical loop form). Για να πληρούμε και αυτή την προϋπόθεση, δημιουργήσαμε μια μεταβλητή ακεραίου w, η οποία έγινε μετρητής για την εσωτερική επανάληψη της τιμοδότησης του πίνακα H. Στη συνέχεια, όπως γνωρίζουμε από την θεωρία [3], η ύπαρξη συνθηκών απαγορεύεται εκτός και αν πρόκειται για δυαδική λογική. Έτσι, η δομή *switch*, η οποία γέμιζε τους πίνακες I_i και I_j, αντικαταστάθηκε από 4 δομές *if*. Κατόπιν, καθώς η κλήση σε συναρτήσεις υπόκειται σε περιορισμούς, αποφασίστηκε η συνάρτηση *find_array_max()* να τοποθετηθεί μέσα στο σώμα της *main* και να τροποποιηθεί έτσι ώστε να υπολογίζει ταυτόχρονα όλα τα μέγιστα της κάθε αντιδιαγωνίου. Αντίθετα, η συνάρτηση *similarity_score()* μιας και κάνει μια απλή σύγκριση, θα μπορούσε να χαρακτηριστεί *inline* και προτιμήθηκε να

χρησιμοποιηθεί το directive *#pragma omp declare simd* [4] προκειμένου να δηλωθεί ότι μπορεί να διανυσματοποιηθεί. Για τον πιο αποδοτικό υπολογισμό των δεδομένων, ο πίνακας *temp* έγινε δυο διαστάσεων (temp2) με σκοπό να υπολογίζονται ταυτόχρονα και οι 4 τιμές για κάθε στοιχείο της αντιδιαγωνίου. Για τα παραπάνω, οι μεταβλητές max,m και my_j (ο μετρητής της εσωτερικής επανάληψης στο προηγούμενο ερώτημα) ορίστηκαν ως private. Τέλος, προκειμένου να γίνει ταχύτερη μεταφορά και επεξεργασία των δεδομένων[5], αποφασίστηκε η μνήμη που θα δεσμευτεί από τους πίνακες H, I_i και I_j να στοιχηθεί ανά 16 bytes. Η επιλογή των 16 bytes έγινε βάση της τεχνολογίας του επεξεργαστή που είναι SSE4.1/4.2 [6] και άρα η μεταφορά των δεδομένων είναι βέλτιστη με αυτή την στοίχιση [7].

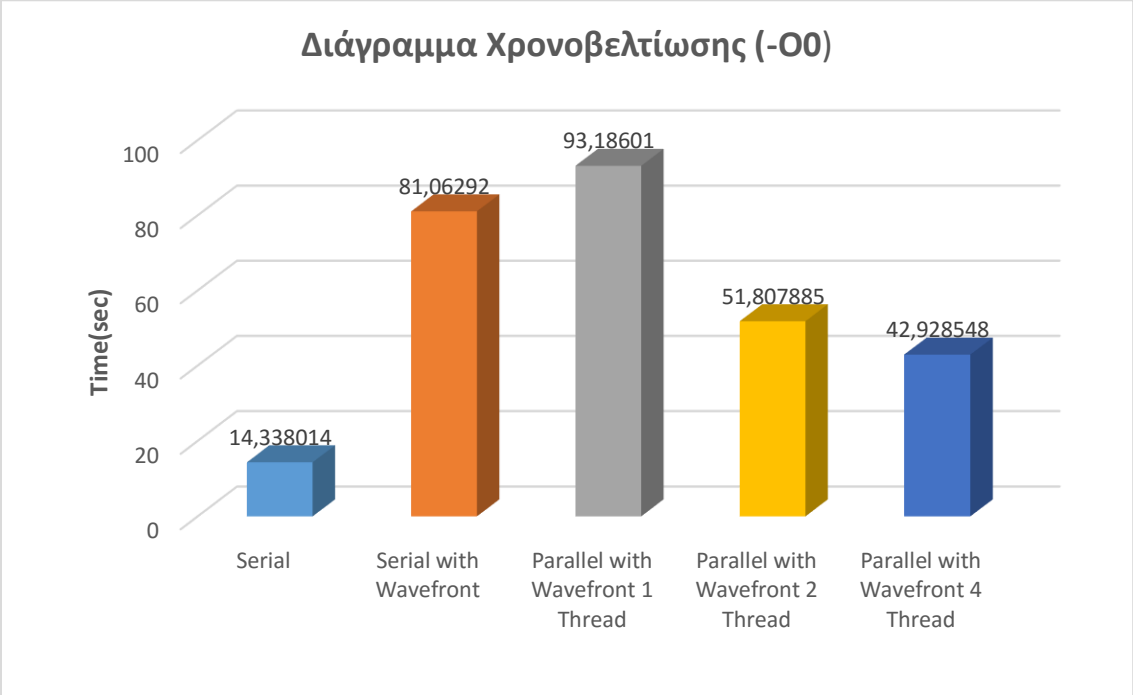
Όσον αφορά την υλοποίηση, επιλέχθηκε να προστεθεί το directive simd στη γραμμή 177 του αρχείου SmithWaterman_parallel_simd.cpp. Έτσι όταν ο μεταφραστής θα συναντήσει το *#pragma omp parallel for simd* θα παραλληλοποιήσει σε νήματα τον κώδικα αλλά και ταυτόχρονα οι πράξεις που θα γίνονται θα είναι τύπου simd με αποτέλεσμα τον πολύ μικρότερο υπολογιστικό χρόνο.

## 5. Σύγκριση σειριακού με παράλληλες εκδόσεις

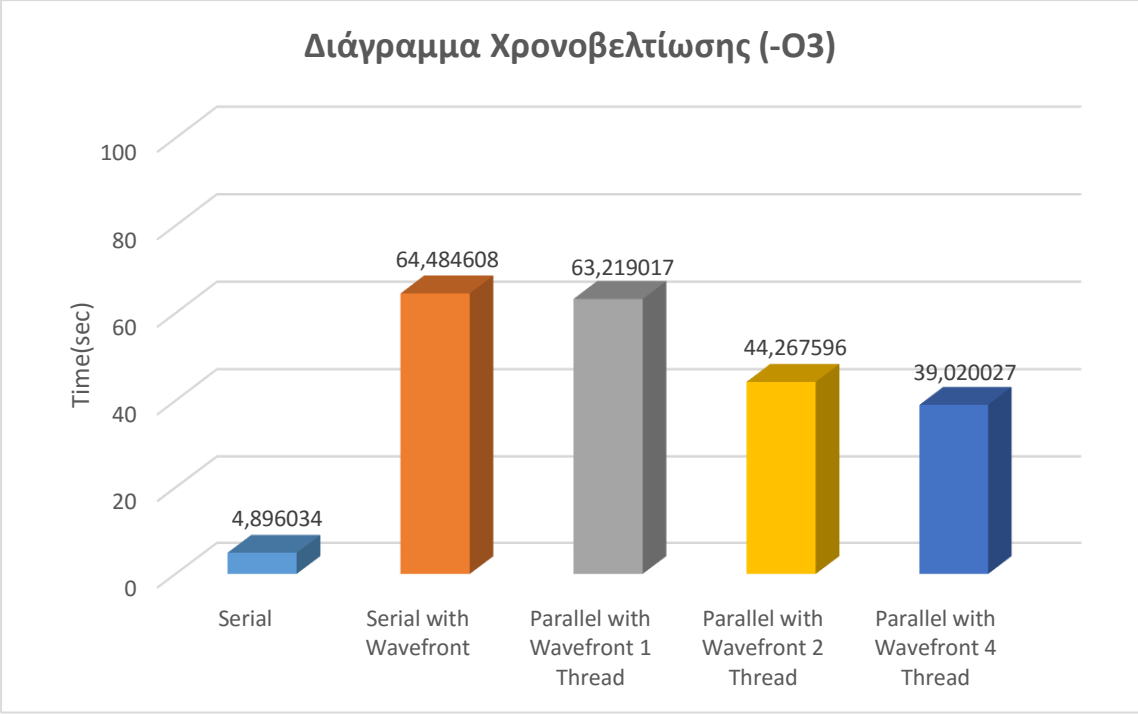
| Threads | Optimization | Time (sec) |
|---|---|---|
| Serial | -O0 | 14.338014 |
| Serial with Wavefront | -O0 | 81.06292 |
| Parallel with Wavefront 1 Thread | -O0 | 93.18601 |
| Parallel with Wavefront 2 Threads | -O0 | 51.807885 |
| Parallel with Wavefront 4 Threads | -O0 | 42.928548 |
| | | |
| Serial | -O3 | 4.896034 |
| Serial with Wavefront | -O3 | 64.484608 |
| Parallel with Wavefront 1 Thread | -O3 | 63.219017 |
| Parallel with Wavefront 2 Threads | -O3 | 44.267596 |
| Parallel with Wavefront 4 Threads | -O3 | 39.020027 |

Πίνακας 5.1: Χρόνοι εκτέλεσης Σειριακού και Παραλληλοποιημένου Προγράμματος

Από τις παραπάνω μετρήσεις, φαίνεται η διαφορά που έχει το πρόγραμμά μας στο χρόνο εκτέλεσης. Αρχικά, παρατηρείται ότι το αρχικό σειριακό πρόγραμμα είναι πολύ πιο γρήγορο από το τροποποιημένο σειριακό με χρήση του αλγορίθμου Wavefront. Αυτό οφείλεται στην έλλειψη τοπικότητας των αναφορών καθώς η αντιδιαγώνια προσπέλαση του πίνακα επεξεργάζεται στοιχεία τα οποία δεν είναι σε συνεχόμενες θέσεις μνήμης. Επίσης, διακρίνεται ότι όταν χρησιμοποιείται ένα μόνο νήμα ο χρόνος είναι περίπου ίδιος με την σειριακή εκτέλεση, αποτέλεσμα το οποίο ήταν αναμενόμενο. Ακόμα, παρατηρείται ότι όσο πιο πολλά είναι τα νήματα που χρησιμοποιούνται τόσο πιο εμφανής είναι η βελτίωση στο τροποποιημένο πρόγραμμα.




Διάγραμμα 5.1 : Χρονοβελτίωση με –O0

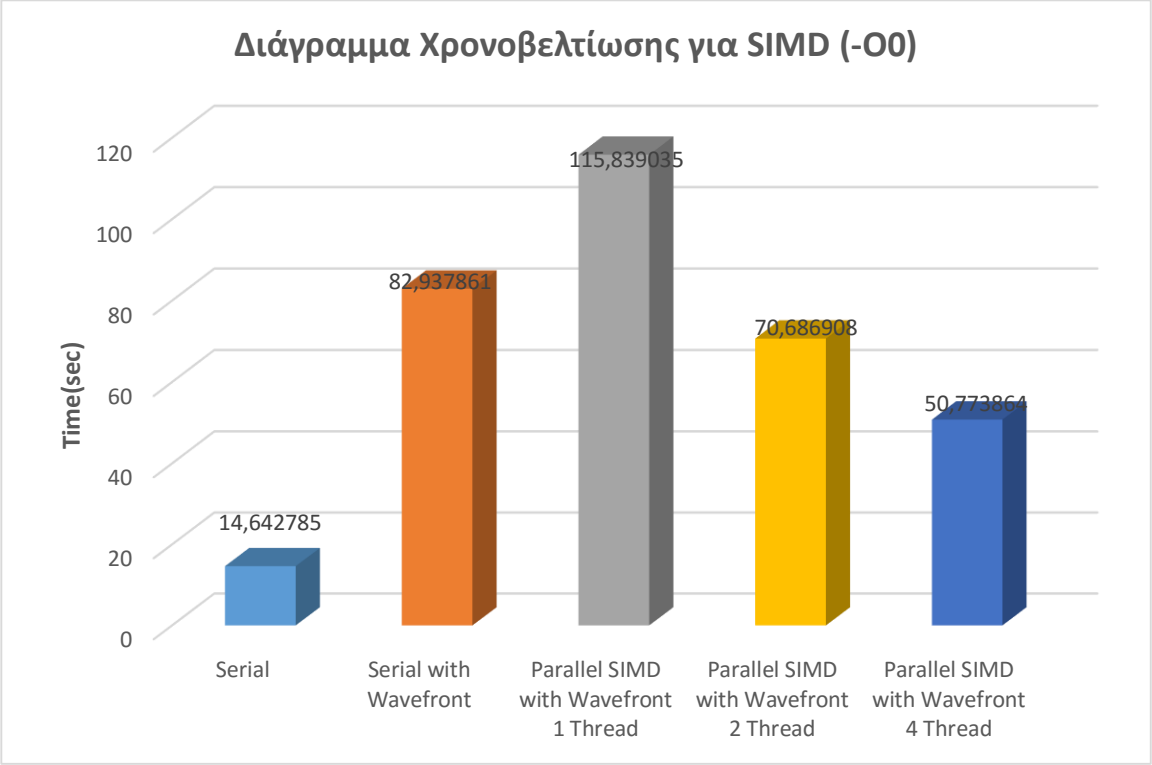



Διάγραμμα 5.2 : Χρονοβελτίωση με –O3

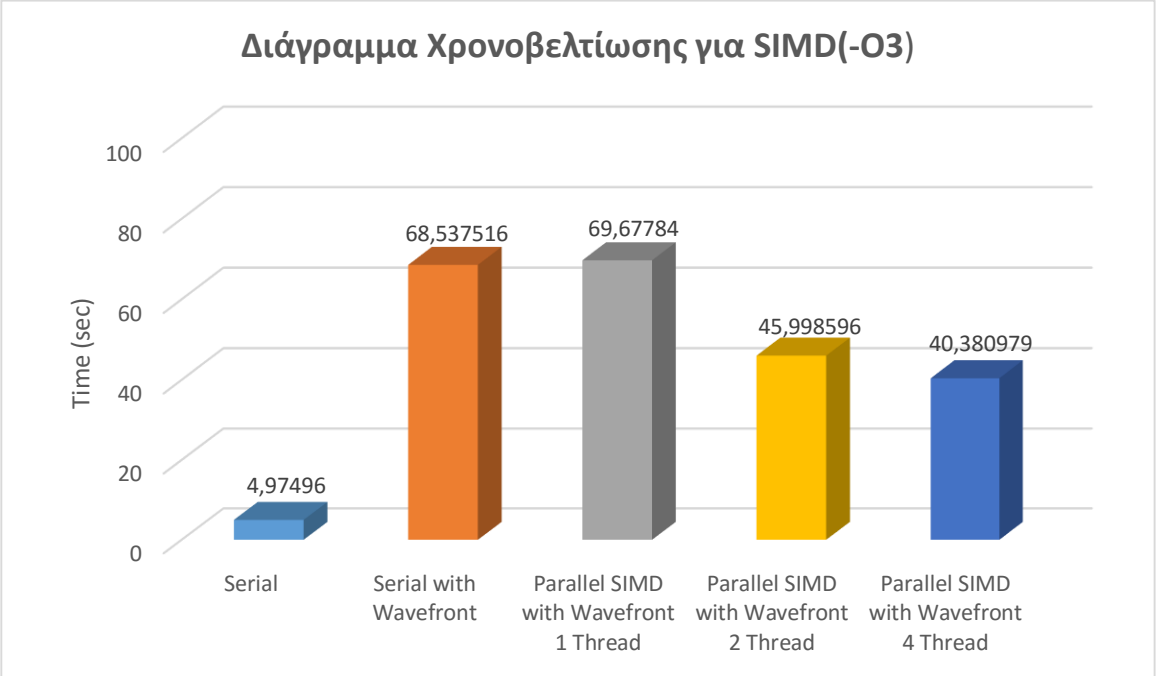
| Threads | Optimization | Time (sec) |
|---|---|---|
| Serial | -O0 | 14.642785 |
| Serial with Wavefront | -O0 | 82.937861 |
| Parallel SIMD with Wavefront 1 Thread | -O0 | 115.839035 |
| Parallel SIMD with Wavefront 2 Thread | -O0 | 70.686908 |
| Parallel SIMD with Wavefront 4 Thread | -O0 | 50.773864 |
| | | |
| Serial | -O3 | 4.97496 |
| Serial with Wavefront | -O3 | 68.537516 |
| Parallel SIMD with Wavefront 1 Thread | -O3 | 69.67784 |
| Parallel SIMD with Wavefront 2 Thread | -O3 | 45.998596 |
| Parallel SIMD with Wavefront 4 Thread | -O3 | 40.380979 |

Πίνακας 5.2: Χρόνοι εκτέλεσης Σειριακού και Παραλληλοποιημένου με εντολές SIMD Προγράμματος

Από τον παραπάνω πίνακα παρατηρούμε ότι σε όλες τις περιπτώσεις οι χρόνοι που προκύπτουν από πρόγραμμα που περιέχει εντολές SIMD είναι αρκετά παρόμοιοι με αυτούς που προκύπτουν από απλή παραλληλοποίηση. Αυτό δεν είναι αναμενόμενο μιας και είχαμε προβλέψει ότι η χρήση SIMD εντολών θα μείωνε δραματικά τον χρόνο. Εικάζουμε ότι ο λόγος που συμβαίνει αυτό είναι ότι υπάρχουν δεδομένα τα οποία δεν έχουν ευθυγραμμιστεί σωστά στην μνήμη ή ότι η επιλογή SIMD εντολών που κάναμε δεν επέτρεψε τη βέλτιστη διανυσματοποίηση,

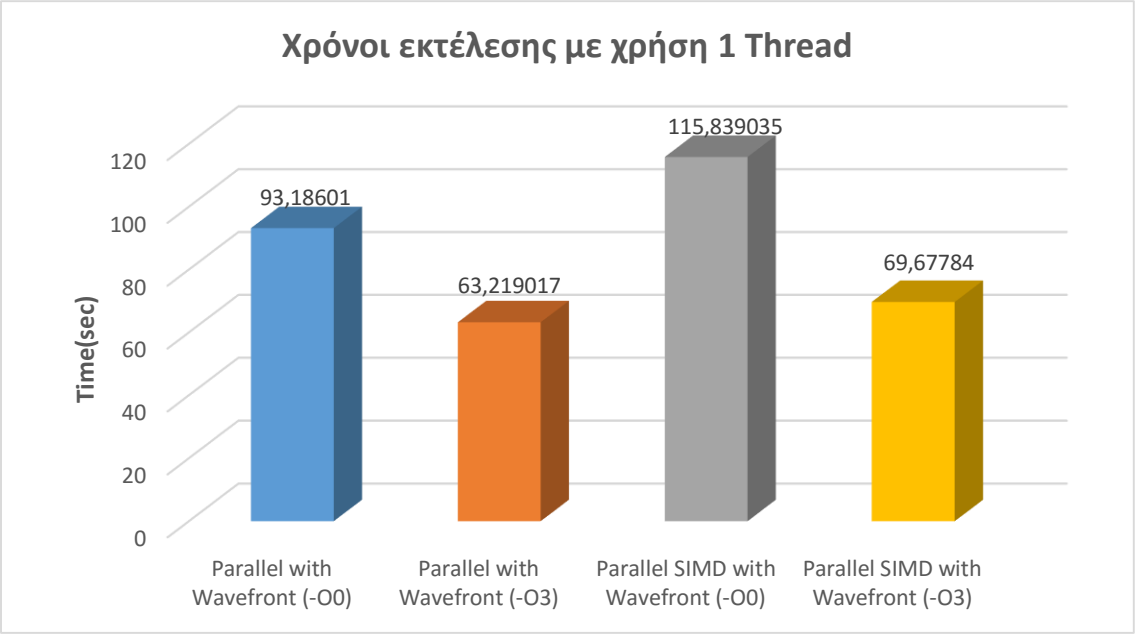



Διάγραμμα 5.3 : Χρονοβελτίωση για SIMD με –O0

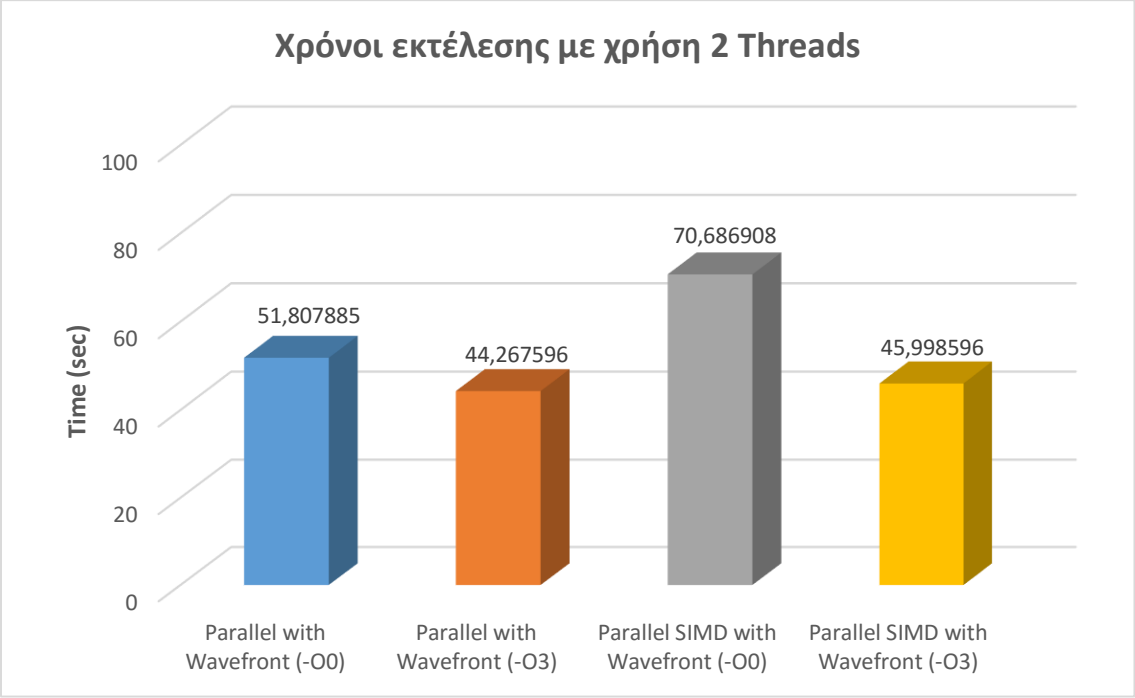



Διάγραμμα 5.4 : Χρονοβελτίωση για SIMD με –O3

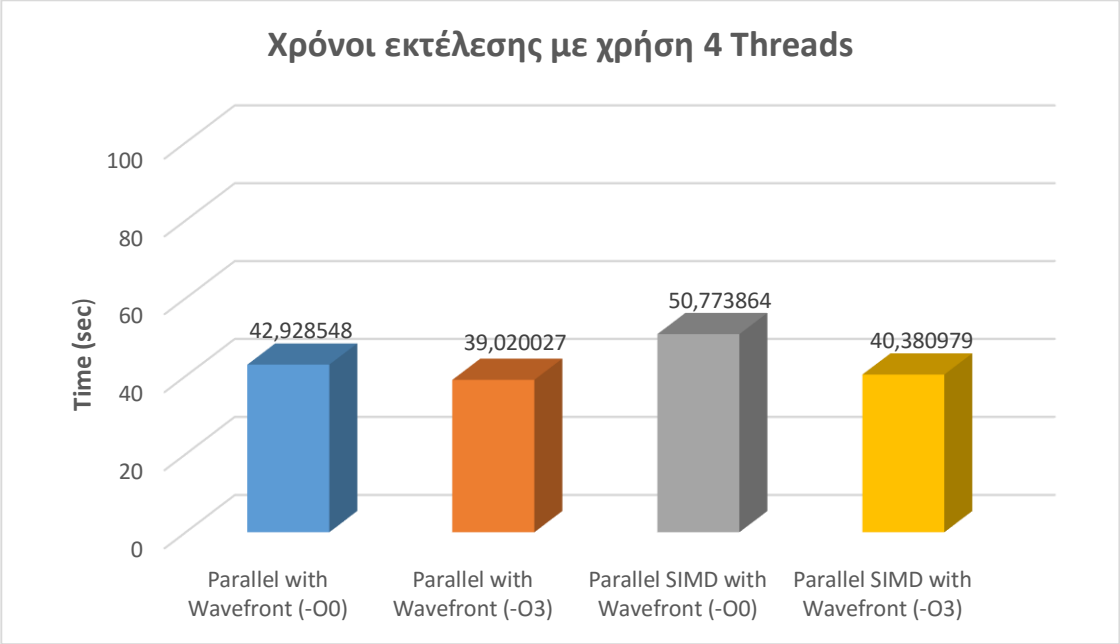
Παρακάτω παρατίθενται οι μετρήσεις που λάβαμε για διαφορετικές τεχνικές και διαφορετικά επίπεδα optimization ανάλογα με τον αριθμό των threads που τα εκτελούσαν:




Διάγραμμα 5.5 : Χρόνοι Εκτέλεσης με 1 Thread




Διάγραμμα 5.6 : Χρόνοι Εκτέλεσης με 2 Threads




Διάγραμμα 5.7 : Χρόνοι Εκτέλεσης με 4 Threads

## 6. Επισυναπτόμενα Αρχεία

Στην αναφορά μας επισυνάπτουμε τα αρχεία που δημιουργήθηκαν στα πλαίσια της εργασίας. Τα αρχεία αυτά είναι:

i. *SmithWaterman.cpp* : Το αρχείο που δόθηκε στην εκφώνηση της άσκησης
ii. *SmithWaterman_wavefront.cpp* : Η σειριακή έκδοση του αρχείου της εκφώνησης αλλά με χρήση του αλγορίθμου Wavefront
iii. *SmithWaterman_parallel.cpp* : Η παραλληλοποιημένη έκδοση του αρχείου Smith_Waterman_wavefront.cpp
iv. *SmithWaterman_parallel_simd.cpp* : Η παραλληλοποιημένη έκδοση του αρχείου Smith_Waterman_wavefront.cpp αλλά και με προστιθέμενες simd εντολές.
v. *run.sh* : bash script που χρησιμοποιείται για την αξιοπιστία των αποτελεσμάτων (ίδια ευθυγράμμιση εξόδων προγραμμάτων)

## 7. Βιβλιογραφία

[1] The Scalasxa Development Team, “Scalasca User Guide”, Version 2.3 , April 2016 , http://www.scalasca.org/software/scalasca-2.x/download.html

[2] OpenMP Architecture Review Board, “OpenMP Application Program Interface,” Version 4.5, November 2015 , http://www.openmp.org

[3] I. E. Βενέτης , *Chapter06-Data Dependencies – Vectorization* , Απρίλιος 2016 σελ. 29

[4] X. Tian, B. de Supinski , *Explicit Vector Programming with OpenMP 4.0 SIMD Extensions* , Primeur Magazine , November 2014

[5] G. Zitzlsberger,*C++ SIMD parallelism with Intel Cilk Plus and OpenMP 4.0* , Intel Corporation , December 2014

[6] Intel Core i5 4200 Specifications, http://ark.intel.com/products/75459/Intel-Core-i5-4200U-Processor-3M-Cache-up-to-2_60-GHz

[7] J. Deslippe, H. He, H. Wasserman , Woo-Sun Yang, *OpenMP and Vectorization Training Introduction ,2015,* σελ 33-34