Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Βρουβάκης Γιάννης : 2014030122
Αλέξανδρος Μιχαήλ : 2014030077

Αρχιτεκτονική Παράλληλων Υπολογιστών HPY418

Ως Θέμα μελέτης είχαμε την αξιοποίηση της παραλληλίας σε κώδικα C, έτσι ώστε να καταφέρουμε να διαχειριζόμαστε μεγάλο όγκο δεδομένων αισθητά πιο γρήγορα.

Το είδος παραλληλίας που χρησιμοποιήσαμε ήταν επιπέδου μνήμης SAS(Shared Address Space) με τη βοήθεια των OpenMP & Pthreads.

Ο κώδικας που υλοποιήσαμε ζητούσε τη κατασκευή δύο συνόλων συμβολοσειρών, σε ένα δεδομένο μήκος, και το άθροισμα της απόστασης Hamming κάθε σειράς με τις υπόλοιπες. Ενδεικτικά μεγέθη συνόλων είναι τα 10,100,1000,10000 καθώς για το μέγεθος των σειρών 10,100,1000 .

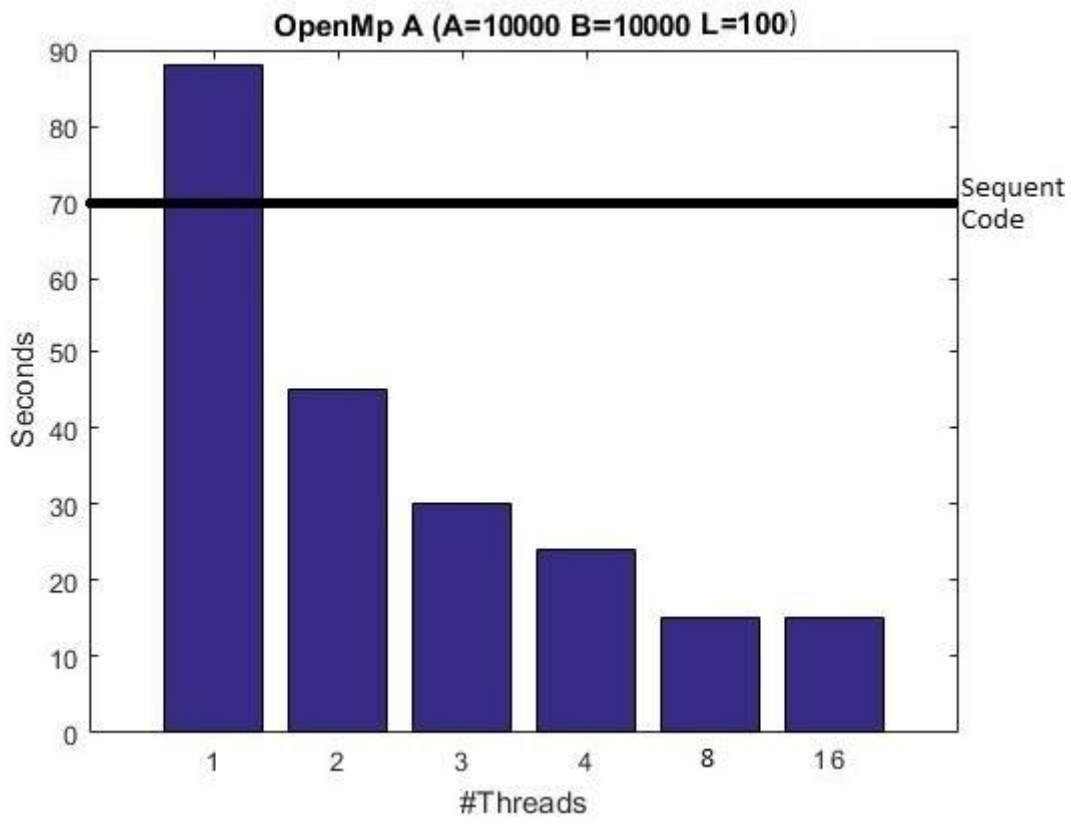
Η πλατφόρμα υπολογισμών μας χρησιμοποιούσε τον επεξεργαστή της intel i7 στους 4 πυρήνες, παρακάτω περιγράφονται παρατηρήσεις και αποτελέσματα από τις δοκιμές μας.

Αρχικά να προσθέσουμε πως ο σειριακός κώδικας στις τιμές A=1000,B=1000,L=1000 είχε εκτέλεση στα 7sec, A=10000,B=10000,L=100 εκτέλεση στα 70sec, που αποτέλεσαν τις επί το πλείστον εισόδους για τις δοκιμές μας.

Η διαδικασία που πρέπει να ακολουθούμε κάθε φορά που χρησιμοποιούμε παραλληλία, είναι να εντοπίζουμε το σημείο που θα πραγματοποιηθεί, έτσι στο δικό μας παράδειγμα επιλέξαμε το task να αποτελείται από :

- A) Μέρος των σειρών σε ένα ζευγάρι.
- B) Ένα ζευγάρι σειρών.
- C) Μία σειρά ενός συνόλου με το δεύτερο σύνολο.

Έτσι τα αποτελέσματα που λάβαμε σε κάθε περίπτωση έχουν ως εξής :

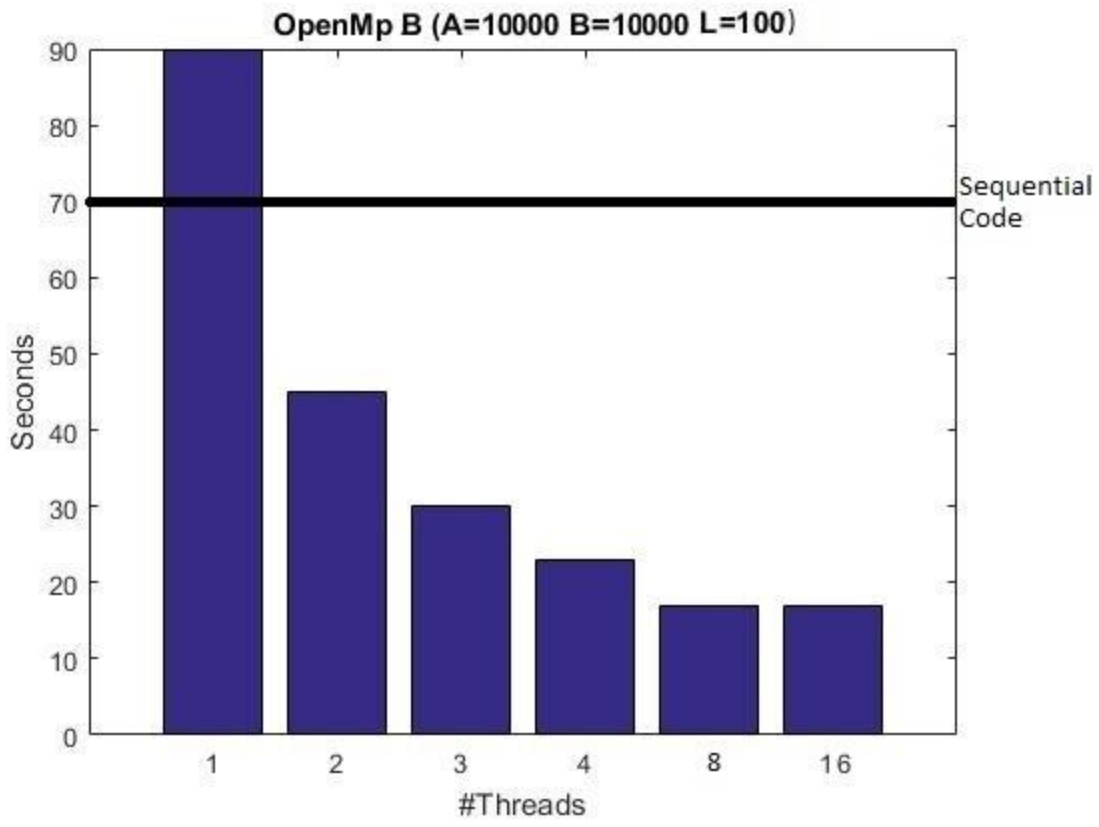


Τα Αποτελέσματα είναι αναμενόμενα καθώς παρατηρούμε πως όσο αυξάνεται ο αριθμός των thread τόσο μειώνεται ο χρόνος εκτέλεσης, **μεγάλα tasks** με ισοδύναμο workload. Ωστόσο αξιοσημείωτο είναι να προσθέσουμε πως η σειριακή εκτέλεση ενώ θεωρητικά ταυτίζεται με εκείνη της παράλληλης για τη χρήση ενός thread, η παράλληλη τείνει να είναι πιο αργή, το οποίο βασίζουμε στο γεγονός πως χρειάζεται χρόνο για την ανάθεση εργασίας σε αυτό το ένα thread.

Ενδεικτικά το Speedup που εμφανίζεται σε κάθε περίπτωση είναι :

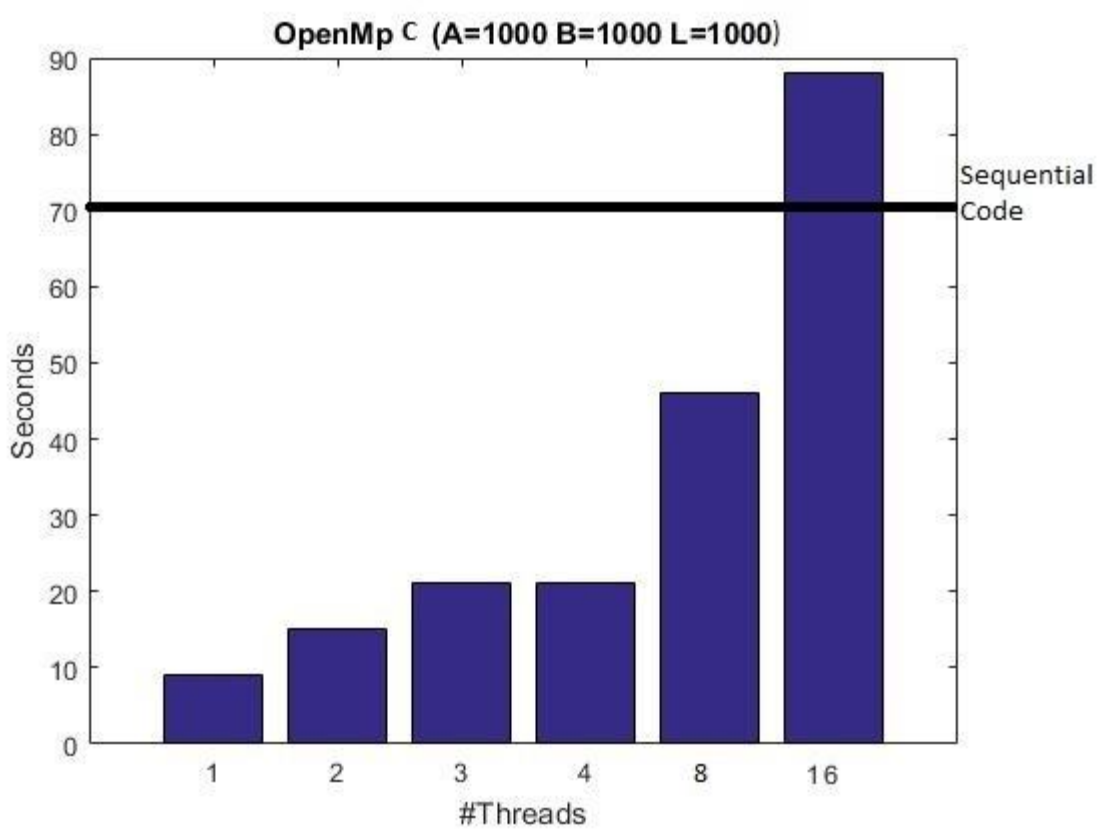
- ***Speedup(2 thread) = 70/45 = 1.55***
- ***Speedup(3 thread) = 70/30 = 2.33***
- ***Speedup(4 thread) = 70/24 = 2.91***
- ***Speedup(8 thread) = 70/15 = 4.66***
- ***Speedup(16 thread) = 70/15 = 4.66***

Και εδώ φτάνουμε στο σημείο που βρίσκουμε το όριο μας, καθώς όσο και να αυξήσουμε τα threads, το speedup ανεβαίνει πολύ αργά.



Όμοια αποτελέσματα με τη προηγούμενη περίπτωση εμφανίζονται και εδώ, το speedup είναι εμφανές και όριο εμφανίζεται ξανά στα 16 threads, οι γενικές παρατηρήσεις ταυτίζονται με αυτές της περίπτωσης Α.

- ***Speedup (2 thread) = 70/45 = 1.55***
- ***Speedup (3 thread) = 70/30 = 2.33***
- ***Speedup (4 thread) = 70/23 = 3.04***
- ***Speedup (8 thread) = 70/17 = 4.11***
- ***Speedup (16 thread) = 70/17 = 4.11***

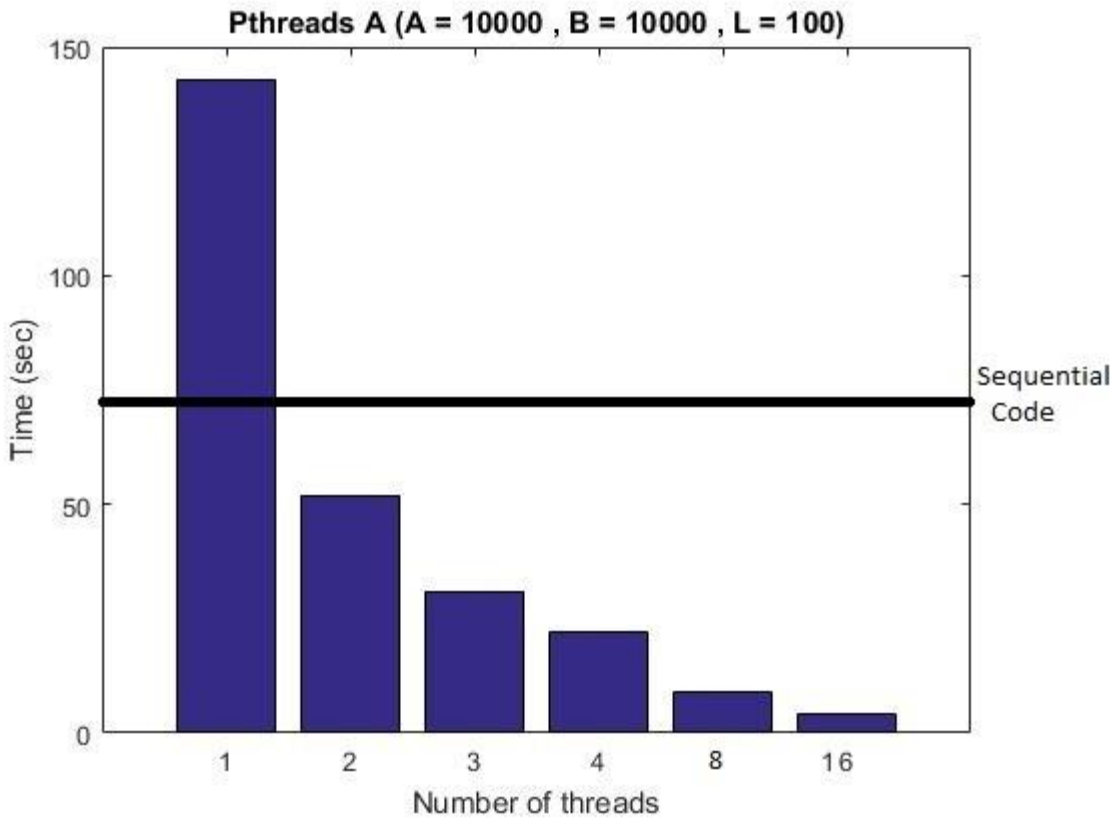


Το αποτέλεσμα που λαμβάνουμε σε αυτή τη περίπτωση δεν είναι το επιθυμητό, καθώς περιμέναμε πως θα υπήρχε βελτίωση στην εκτέλεση παρατηρούμε το αντίθετο, αφού φαίνεται πως τα threads μεταξύ τους δεν επικοινωνούν καλά, με αποτέλεσμα όσο περισσότερα threads χρησιμοποιούμε τόσο μειώνεται η απόδοση του προγράμματος μας.

- ***Speedup (2 threads) = 7/15 = 0.466***
- ***Speedup (3 threads) = 7/21 = 0.333***
- ***Speedup (4 threads) = 7/21 = 0.333***
- ***Speedup (8 threads) = 7/46 = 0.156***
- ***Speedup (16 threads) = 7/88 = 0.079***

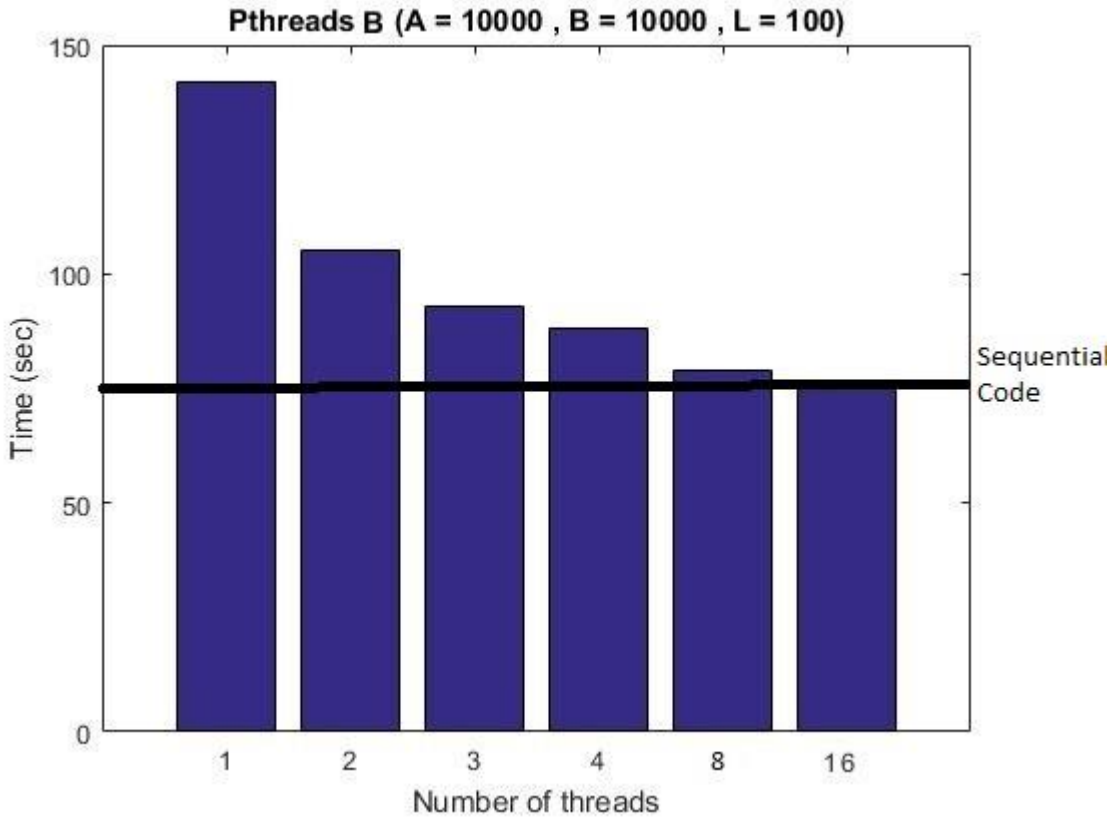
Καθόλου ικανοποιητικά αποτελέσματα, η δημιουργία των threads σε συνδυασμό με το κακό communication ή/και το workload δεν επιφέρουν αποδεκτά αποτελέσματα.

Τώρα έχουμε τα αποτελέσματα από τη χρήση των Posix Threads :



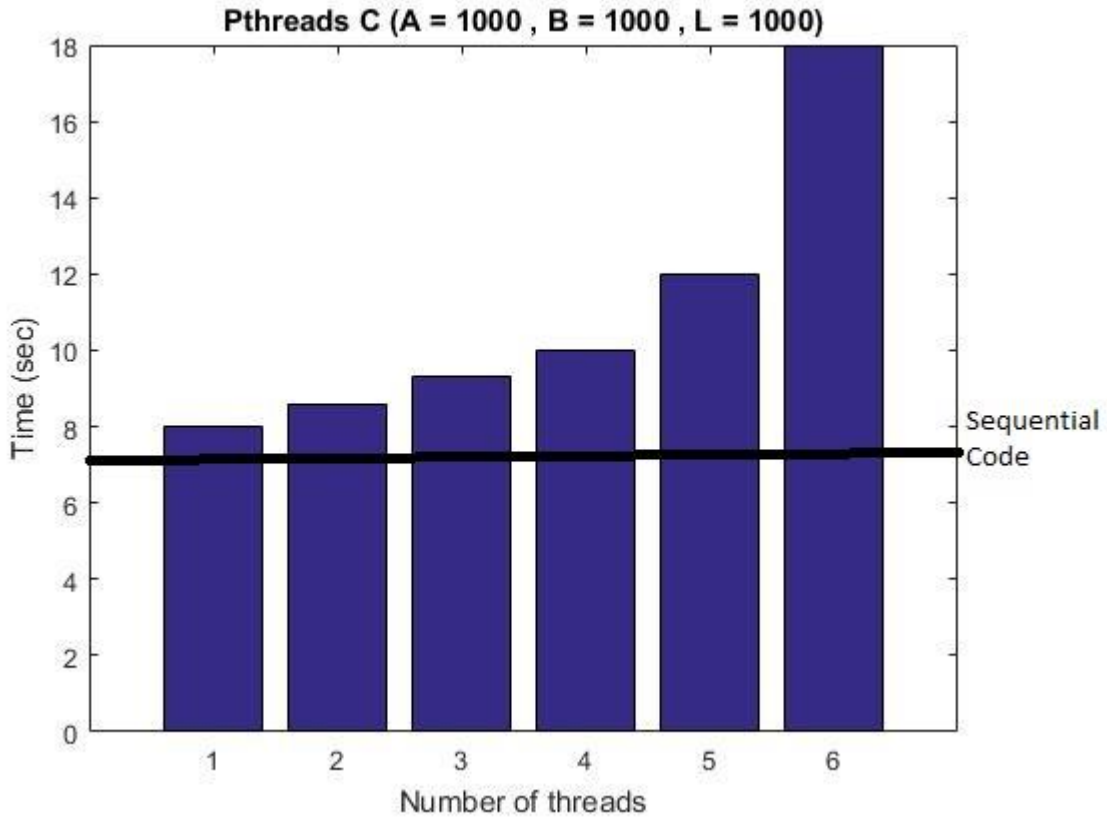
Παρατηρούμε εδώ, πως εμφανίζεται καλύτερο speedup από τη χρήση του OpenMP όπου είναι λογικό καθώς χρησιμοποιώντας την low level API Posix Threads, καταφέραμε να μοιράσουμε το workload όπως εμείς θέλαμε. Τα αποτελέσματα είναι ιδιαίτερα θετικά και το speedup είναι ανώτερο από εκείνο των OpenmMp επίσης ιδιαίτερη εντύπωση μας κάνει η περίπτωση των 16 threads όπου το speedup αυξάνεται δραματικά, ωστόσο αποτελεί λογικό speedup 16/1 που θα έπρεπε να έχουμε στη συγκεκριμένη περίπτωση, πράγμα που σημαίνει πως σε εκείνο το σημείο εξαφανίζεται το overhead και επωφελούμαστε από τη παραλληλία στο μέγιστο.

- ***Speedup (2 threads) = 70/52 = 1,34***
- ***Speedup (3 threads) = 70/31 = 2,25***
- ***Speedup (4 threads) = 70/22 = 3,18***
- ***Speedup (8 threads) = 70/9 = 7,7***
- ***Speedup (16 threads) = 70/5 = 14***



Για τη περίπτωση αυτή παρατηρούμε πως το speedup δε βελτιώνει καθώς βρίσκει όριο ελάχιστης υλοποίησης στα 16 threads το οποίο είναι πάλι πιο αργό από το σειριακό, η κατασκευή αλλά και το join των threads, σε συνδυασμό με άλλα drawbacks όπως communication και workload καθιστά τη παραλληλία περιττή.

- ***Speedup (2 threads) = 70/105 = 0.666***
- ***Speedup (3 threads) = 70/93 = 0.752***
- ***Speedup (4 threads) = 70/88 = 0.79***
- ***Speedup (8 threads) = 70/79 = 0.88***
- ***Speedup (16 threads) = 70/75 = 0.933***



Για το τελευταίο κομμάτι, παρατηρούμε πως έχω τα χειρότερα αποτελέσματα, το κόστος για τη δημιουργία είναι τεράστιο και σε συνδυασμό με το μεγάλο communication το overhead που δημιουργείται είναι αρκετό για να μην υπάρξει καμία βελτίωση αντιθέτως όσο αυξάνονται τα threads ο χρόνος εκτέλεσης αυξάνεται δραματικά.

- ***Speedup (2 threads) = 7/8,6 = 0.813***
- ***Speedup (3 threads) = 7/9.3 = 0.752***
- ***Speedup (4 threads) = 7/10 = 0.700***
- ***Speedup (8 threads) = 7/12 = 0.583***
- ***Speedup (16 threads) = 7/18 = 0.388***

**Για όλες τις γραφικές έγινε χρήση της MATLAB*

Συμπεράσματα:

Με βάση όλα τα παραπάνω, παρατηρούμε ότι ο παραλληλισμός είναι βελτίστως για μεγάλου μεγέθους task καθώς γνωρίζουμε πως το workload δεν αποτελεί πρόβλημα στο δικό μας κώδικα. Χρησιμοποιώντας το OpenMP παρατηρήσαμε θετικό speedup στις δυο πρώτες περιπτώσεις αλλά όχι όσο με τη χρήση των pthreads λόγο του ότι με την τεχνική αυτή διαχειριζόμασταν καλυτέρα το communication και την ανάθεση εργασίας.

Τέλος