ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμημα ΗΜΜΥ. Τομεάς Ηλεκτρονικής

Παράλληλα και Διανεμημένα Συστήματα

Εργασία 1: knnsearch

Συστήματα με κοινή μνήμη

Συντάκτης:
Χρήστος Χουτουρίδης
ΑΕΜ:8997
cchoutou@ece.auth.gr

Διδάσκων:
Νικόλαος Πιτσιάνης
nikos.pitsianis@eng.auth.gr

21/11/2024

1. Εισαγωγή

Η παρούσα εργασία αφορά τον παραλληλισμό συστημάτων με κοινή μνήμη. Το αντικείμενό είναι ο παραλληλισμός τους αλγορίθμου εύρεσης κοντινότερων γειτόνων **knnseach**. Στην εργασία αυτή υλοποιούμε τόσο μια σειριακή προσέγγιση όσο και μια που παραλληλοποιείται με **pthreads**, με **open-cilk** και με **open-mp**. Στην παραλληλοποιήσιμη έκδοση δεν μας ενδιαφέρει η ακρίβεια του αλγορίθμου, κάτι που μας δίνει την δυνατότητα να βελτιώσουμε τον χρόνο εκτέλεσης.

2. Παραδοτέα

Τα παραδοτέα της εργασίας αποτελούνται από:

- Την παρούσα αναφορά.
- Το σύνδεσμο με το αποθετήριο που περιέχει τον κώδικα για την παραγωγή των εκτελέσιμων, της αναφοράς και τις μετρήσεις.
- Τον σύνδεσμο με το image με το οποίο μεταγλωττίσαμε και εκτελέσαμε τις μετρήσεις.

3. Υλοποίηση

Πριν ξεκινήσουμε να αναλύουμε τον παραλληλισμό και τις μετρήσεις καλό θα ήταν να αναφερθούμε στην υλοποίηση. Για την παρούσα εργασία χρησιμοποιήσαμε τη γλώσσα **C++** και πέρα από τις βιβλιοθήκες που ζητούνται από την εκφώνηση (cilk/omp) έγινε και χρήση της **openblas**. Βοηθητικά έγινε επίσης χρήση του **bash** και της **matlab** κυρίως για την λήψη και οπτικοποίηση των αποτελεσμάτων. Στην παρούσα εργασία γίνεται ευρεία χρήση πινάκων τόσο για εισαγωγή και εξαγωγή των δεδομένων, όσο και για τους υπολογισμούς. Για το λόγο αυτό θεωρήσαμε ότι έπρεπε πρώτα να υλοποιήσουμε κάποιες αφαιρέσεις που θα μας επέτρεπαν να χειριστούμε ευκολότερα το πρόβλημα

3.1. Ο τύπος Matrix<>

Η πιο βασική αφαίρεση που υλοποιήσαμε είναι ο τύπος **Matrix<>**. Πρόκειται για μια αναπαράσταση πινάκων, είτε μονοδιάστατων είτε δισδιάστατων. Η υλοποίησή του έγινε με χρήση templates και έτσι μπορούμε να κατασκευάζουμε πίνακες οποιουδήποτε τύπου. Η αφαίρεση εσωτερικά κάνει χρίση μονοδιάστατης αποθήκευσης των δεδομένων με αποτέλεσμα να έχουμε την δυνατότητα να διαλέγουμε εμείς το ordering του πίνακα. Αν είναι δηλαδή column-major ή row-major. Στην αφαίρεση δημιουργούμε πίνακες στους οποίους τον έλεγχο της μνήμης τον έχουμε εσωτερικά στο αντικείμενο. Για την αποθήκευση χρησιμοποιούμε **std::vector** κάτι που μας δίνει ευελιξία στο μέγεθος αλλά και στη χρήση.

Η κύρια όμως λειτουργικότητα που καθιστά την αφαίρεση βολική για την συγκεκριμένη εργασία είναι η δυνατότητα να κατασκευάσουμε αντικείμενα των οποίων η μνήμη είναι και εξωτερικά του αντικειμένου. Να λειτουργήσει δηλαδή ο ίδιος ο τύπος ώς viewer, όπως για παράδειγμα το string view, χωρίς όμως να χρειάζεται να έχουμε διαφορετικό τύπο για αυτή τη δουλειά. Σε αυτή την περίπτωση στο εσωτερικό storage δεν αποθηκεύουμε παρά μόνο μια διεύθυνση στα δεδομένα του άλλου πίνακα. Μπορούμε όμως να χρησιμοποιήσουμε όλη την λειτουργικότητα του πίνακα. Αυτό μας δίνει τη δυνατότητα να δημιουργούμε αντικείμενα που "βλέπουν" σε ολόκληρους ή και σε **slices** από πίνακες.

Για παράδειγμα μπορούμε να γράψουμε:

```
Matrix<double> A(4,5);          // Πίνακας 4x5
Matrix<double> vA(A, 0, 2, 5);  // View στις 2 πρώτες γραμμές του A
for (int i=0 ; i<vA.rows() ; ++i)
    for (int j=0 ; j<vA.columns() ; ++j)
```

```
std::cout << A(i, j);
```

Στον παραπάνω παράδειγμα ο πίνακας με τα δεδομένα είναι ο **A**. ο **vA**, είναι ένα *slice* του A, που δείχνει μόνο στις 2 πρώτες γραμμές του A.

3.2. knnsearch version 0

Για το πρώτο σκέλος της άσκησης είχαμε να παρουσιάσουμε την knnsearch με σειριακό τρόπο. Για το σκοπό αυτό υλοποιήσαμε την **v0::knnsearch**. Η συνάρτηση αυτή κάνει χρήση του παραπάνω τύπου (Matrix) και επομένως είναι και ή ίδια template. Η λειτουργία της χωρίζεται σε 3 βασικά μέρη:

- Τον υπολογισμό του **πίνακα αποστάσεων**. Εδώ βρίσκεται ο κύριος υπολογιστικός όγκος της συνάρτησης, λόγω του πολλαπλασιασμού πινάκων που απαιτείται. Για το λόγο αυτό κάνουμε χρήση της openblas.
- Την **αντιστοίχηση των αποστάσεων** των σημείων **με τους δείκτες (indexes)** αυτών των αποστάσεων στον αρχικό πίνακα Corpus. Εδώ ουσιαστικά *"ζευγαρώνουμε"* τον κάθε δείκτη με τη απόσταση που του αναλογεί σε μια δομή **std::pair<>**. Αυτό το κάνουμε ώστε ταξινομώντας ένα διάνυσμα με τέτοια ζευγάρια ώς προς τη μικρότερη απόσταση, να ταξινομούνται αυτόματα και οι δείκτες. Γιαυτό το σκοπό κατασκευάζουμε για κάθε σημείο του Query, ένα διάνυσμα τέτοιων ζευγαριών.
- Την **μερική ταξινόμηση** του παραπάνω διανύσματος. Εδώ κάνουμε χρήση του αλγορίθμου quick-select. Ουσιαστικά ταξινομούμε μόνο ένα κομμάτι του διανύσματος όσα τα στοιχεία που θέλουμε (τον αριθμό των γειτόνων που ψάχνουμε). Ο αλγόριθμος αυτός είναι γραμμικός, περιορίζοντας έτσι λιγάκι τη "χασούρα". Για το σκοπό αυτό χρησιμοποιήσαμε την std::nth_element().

Στην υλοποίησή μας τα Corpus-Query είναι διαφορετικά, κάτι που δεν μας περιορίζει να λύσουμε το πρόβλημα όπου το Corpus == Query φυσικά.

3.3. knnsearch version 1

Για το δεύτερο σκέλος της εργασίας είχαμε να παρουσιάσουμε την knnsearch την οποία θα παραλληλοποιήσουμε. Για το σκοπό αυτό υλοποιήσαμε την **v1::knnsearch**. Η αρχική μας προσέγγιση ήταν **αναδρομική**. Ουσιαστικά χωρίζαμε τα Corpus-Queries σε slices και για καθένα από αυτά καλούσαμε ξανά τον εαυτό μας. Η αναδρομή τερματιζόταν όταν τα slices ήταν αρκετά μικρά ώστε να τρέξει ο σειριακός αλγόριθμος. Για να μην χάνονται όμως γείτονες η knnsearch έπρεπε να τρέχει για όλους τους συνδυασμούς. Το κάθε Corpus με τα δύο διαφορετικά Queries και το κάθε Query αντίστοιχα. Έπειτα συνενώναμε τα αποτελέσματα. Αν και αυτή η προσέγγιση ήταν η ακριβείς μας έδινε όμως τη δυνατότητα να "φτηνύνουμε" τον τρόπο με τον οποίο ενώναμε τα αποτελέσματα. Επίσης για την περίπτωση που το Query ήταν ίδιο με το Corpus, μας προσφέρεται η δυνατότητα να αποφύγουμε έναν από τους 4 συνδυασμούς.

Η προσέγγιση αυτή, αν και απλή, δεν βολεύει για παραλληλοποίηση, καθώς τα νήματα που πρέπει να δημιουργηθούν βρίσκονται μέσα στην αναδρομή, καθιστώντας δύσχρηστη(όχι αδύνατη) τη χρήση των εργαλείων της cilk και της openMP. Ακόμα και το "μέρος" στο οποίο θα έμπαινε η loop-α με τις join για την περίπτωση των pthreads ήταν περίπλοκο. Για το σκοπό αυτό **"ανοίξαμε" την αναδρομή** σε βρόχο επανάληψης. Σε αυτή την υλοποίηση τα Corpus-Queries, χωρίζονται σε ένα αριθμό από slices που μας δίνεται από την γραμμή εντολών. Για όλα αυτά σε ένα διπλό βρόχο καλούμε την σειριακή knnsearch. Ο διπλός βρόχος ουσιαστικά δημιουργεί όλους τους συνδυασμούς των slice από το Corpus με τα slices από το Query. Έτσι πάλι δεν χάνουμε γείτονες. Τα αποτελέσματα συνενώνονται όπου και κρατούνται οι κοντινότεροι γείτονες. Η διαδικασία επιλογής είναι ίδια με τη σειριακή. Γίνεται δηλαδή πάλι χρήση της quick-select. Για να μειώσουμε την ακρίβεια και εδώ μπορούμε να "φτωχύνουμε" το merge. Για το σκοπό αυτό έχουμε ένα όρισμα στη συνάρτηση το οποίο μας κόβει τον αριθμό των κοντινότερων γειτόνων. Σε αυτή την προσέγγιση υπάρχει ακόμα ένα ουσιαστικότερο όφελος. Ο χωρισμός σε slices μας δίνει την δυνατότητα να δημιουργήσουμε threads που δεν θα χρειαστεί ποτέ να χρειαστούν κομμάτια μνήμης από άλλα threads. Αυτό ο διαχωρισμός γίνεται στον έξω βρόχο επανάληψης

της knnsearch

```
#pragma omp parallel for
 for (size_t qi = 0; qi < num_slices; ++qi)
     worker_body (corpus_slices, query_slices, idx, dst, qi,
                  num_slices, corpus_slice_size, query_slice_size, k, m);
```

Για τα το παραπάνω snippet (που είναι και "καρδιά" του αλγόριθμου), ο κάθε worker δουλεύει σε δικό του corpus και query slice και αποθηκεύει τα δεδομένα σε δικό του slice πίσω στους idx και dst. Με αυτό τον τρόπο δεν χρειαζόμαστε κανέναν συγχρονισμό για τα threads.

3.4. Μετρήσεις

Για την παραγωγή των εκτελέσιμων χρησιμοποιήσαμε το προσωπικό μας(μου) laptop. Στον κατάλογο με τον πηγαίο κώδικα υπάρχει ότι είναι απαραίτητο για την μεταγλώττιση και εκτέλεση του κώδικα. Για τη μεταγλώττιση κάναμε χρήση του image hpcimage που βρίσκεται εδώ. Για την μεταγλώττιση και εκτέλεση μπορεί κάποιος να δώσει τις παρακάτω εντολές στο root κατάλογο "homework_1".

```
DOCK="docker run --rm -v <PATH_to_homework_1>:/usr/src/PDS_homework_1 \
     -w /usr/src/PDS_homework_1/ hoo2/hpcimage"
make -j IMAGE=hoo2/hpcimage v0
make -j IMAGE=hoo2/hpcimage v1
make -j IMAGE=hoo2/hpcimage v1_cilk
make -j IMAGE=hoo2/hpcimage v1_omp
make -j IMAGE=hoo2/hpcimage v1_pth
# run pthreads version with 4 slices
eval $DOCK ./out/knnsearch_v1_pth -c <path to hdf5> /test -k 100 -s 4 -t
```

4. Αποτελέσματα

Παρακάτω παραθέτουμε τα αποτελέσματα των μετρήσεων. Για αυτές:

- Εκτελέσαμε την κάθε έκδοση του προγράμματος της v1 για **διαφορετικό αριθμό από threads** για δύο διαφορετικούς πίνακες(all-to-all) ώστε να δούμε την συμπεριφορά. Τα threads μπορούν να περαστούν από τη γραμμή εντολών.
- Εκτελέσαμε την κάθε έκδοση του προγράμματος για δύο διαφορετικούς πίνακες(all-to-all) για **διαφορετική επιθυμητή ακρίβεια**.

Για αντιπαραβολή παραθέτουμε ότι ότι στο ίδιο μηχάνημα το οποίο πήραμε τις μετρήσεις η knnsearch έκδοση του **matlab** κάνει:

- sift: **3.223628 [sec]**
- mnist: **20.634987 [sec]**

4.1. Επίδραση των threads - OpenMP

Για την έκδοση με την openMP, χρησιμοποιώντας διαφορετικό αριθμό threads έχουμε:

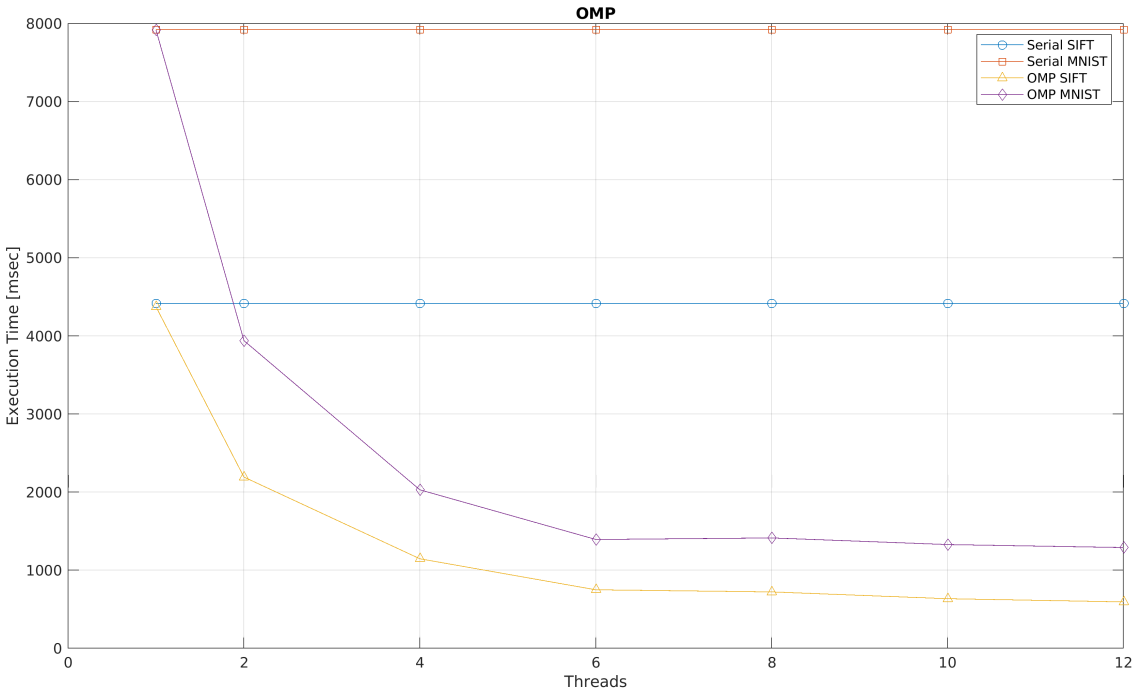

Figure 1: *Βελτιστοποίηση από την παραλληλοποίηση [openMP].*

4.2. Επίδραση των threads - openCilk

Για την έκδοση με την Cilk έχουμε:

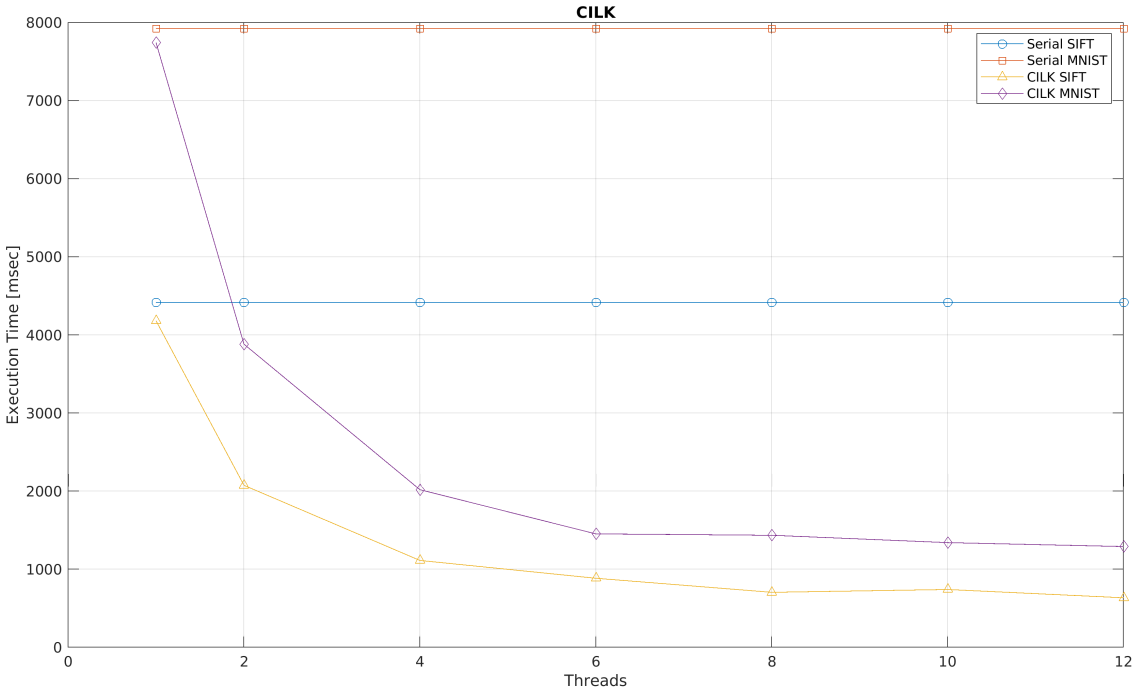

Figure 2: *Βελτιστοποίηση από την παραλληλοποίηση [open-cilk].*

4.3. Επίδραση των threads - pthreads

Για την έκδοση με τα pthreads έχουμε:

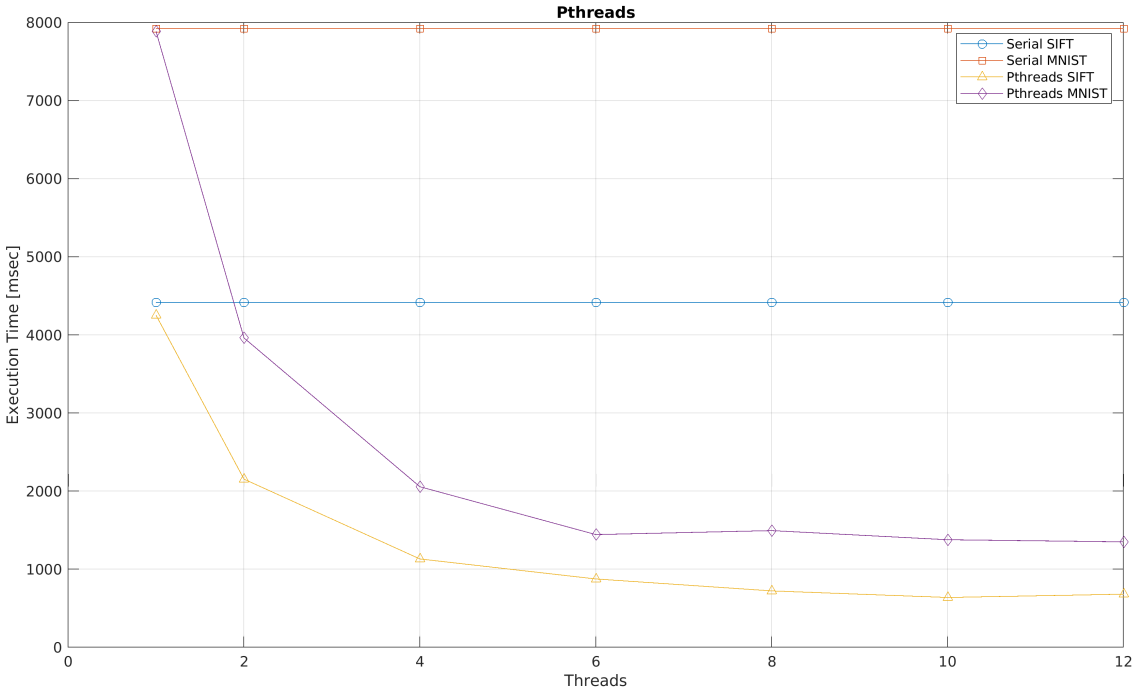

Figure 3: *Βελτιστοποίηση από την παραλληλοποίηση [pthreads].*

4.4. Επίδραση της ακρίβειας - OpenMP

Για την έκδοση με την openMP όσο αλλάζουμε την ακρίβεια έχουμε:

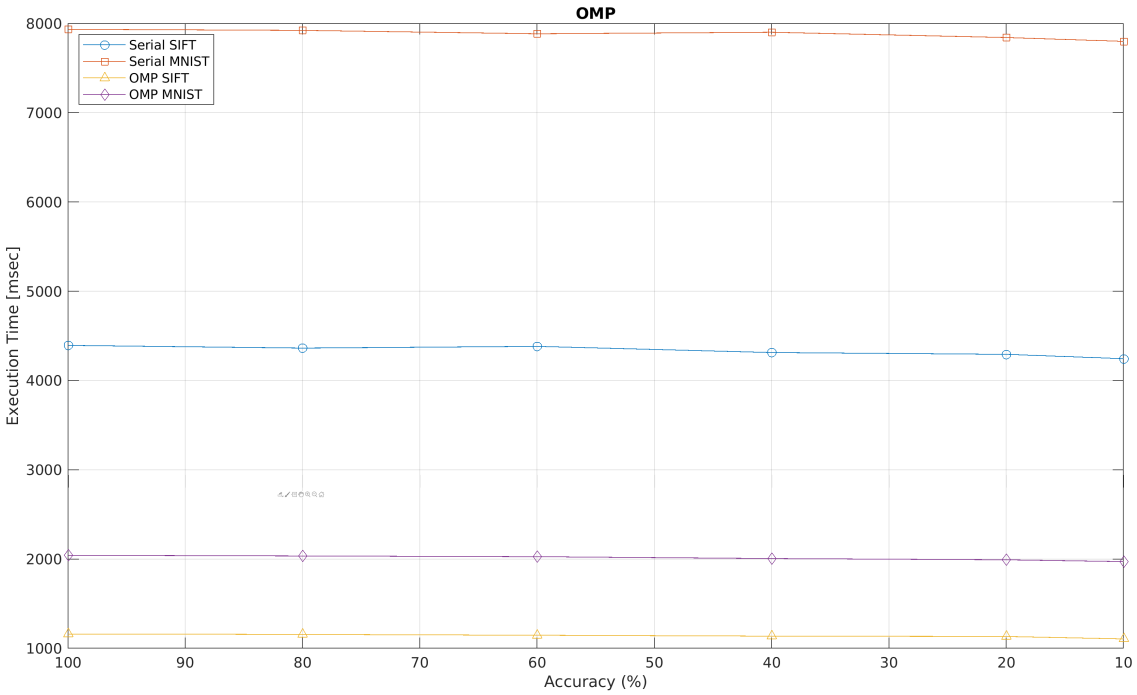

Figure 4: *Βελτιστοποίηση λόγο μειωμένης ακρίβειας [openMP].*

4.5. Επίδραση της ακρίβειας - openCilk

Για την έκδοση με την Cilk όσο αλλάζουμε την ακρίβεια έχουμε:

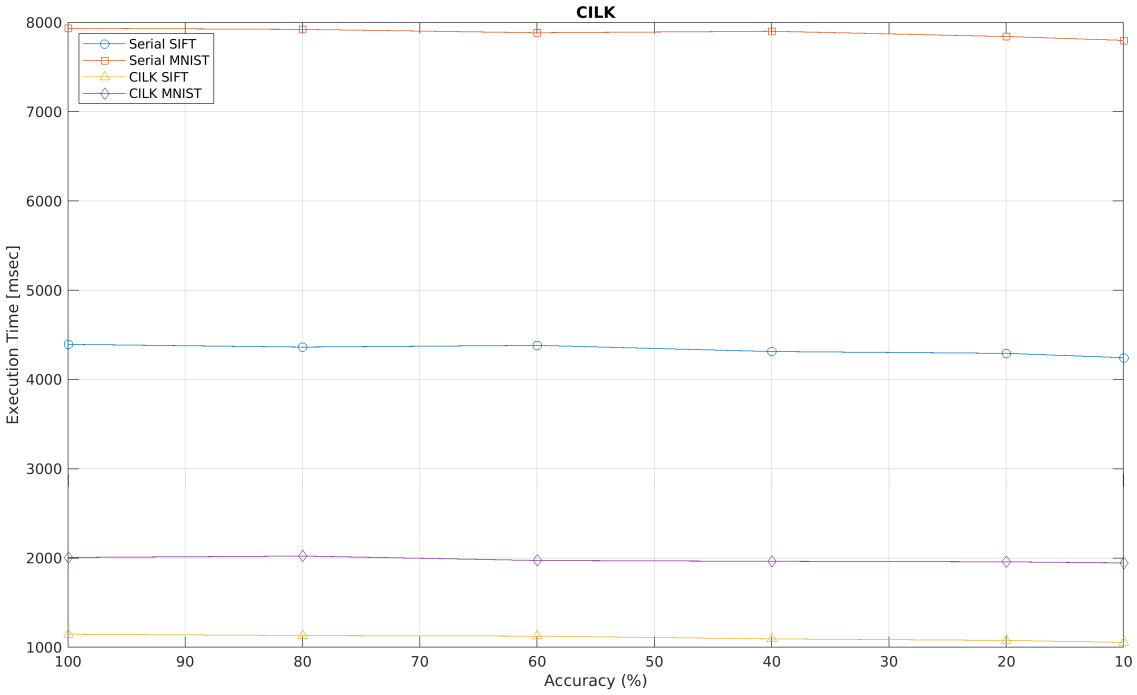

Figure 5: *Βελτιστοποίηση λόγο μειωμένης ακρίβειας [open-cilk].*

4.6. Επίδραση της ακρίβειας - pthreads

Για την έκδοση με τα pthreads όσο αλλάζουμε την ακρίβεια έχουμε:

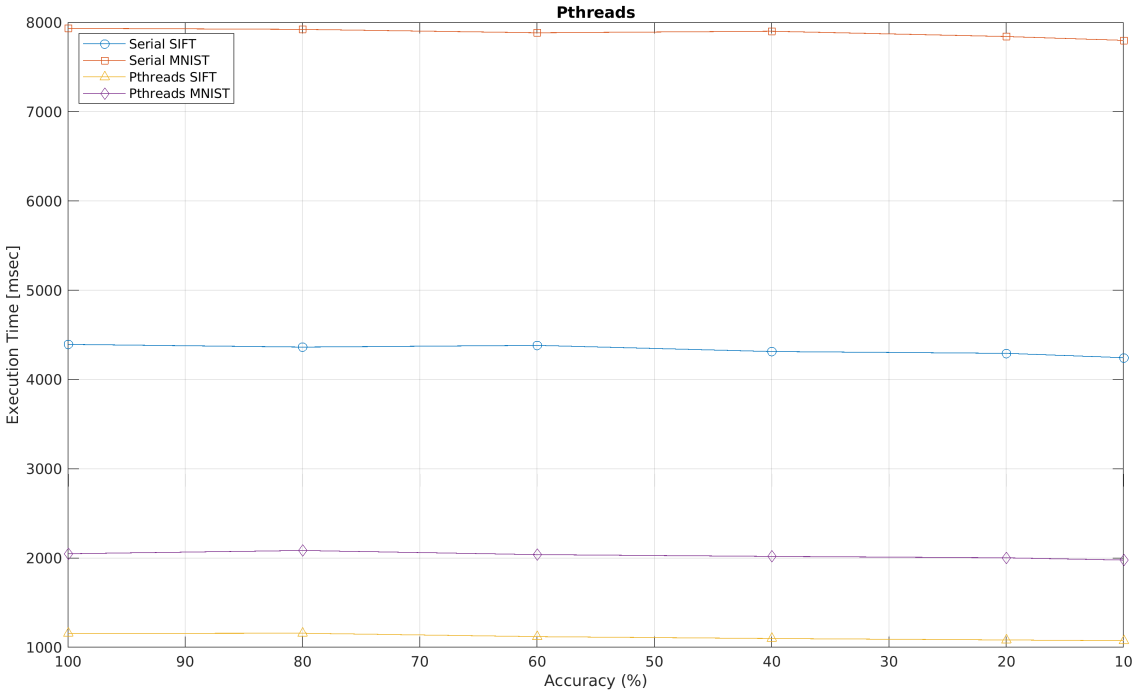

Figure 6: *Βελτιστοποίηση λόγο μειωμένης ακρίβειας [pthreads].*

5. Συμπεράσματα

Απ' ότι βλέπουμε παραπάνω, ενώ ο χρόνος βελτιώνεται καθώς χρησιμοποιούμε παραπάνω threads, αυτό δεν συμβαίνει όταν μειώνουμε την ακρίβεια. Τουλάχιστον όχι όσο θα θέλαμε. Φυσικά αυτό έχει να κάνει με την υλοποίηση μας, καθώς δεν επιλέξαμε καλή τεχνική συνένωσης. Στην υλοποίησή μας δεν έγινε πολύ επιθετική στρατηγική μείωσης της ακρίβειας των αποτελεσμάτων. Να πούμε επίσης ότι ο συγκεκριμένος αλγόριθμος, λόγο της ουσιαστικά "μη" επικοινωνίας που χρειάζονται τα threads, θα μπορούσε να χρησιμοποιηθεί και σε απομακρυσμένα συστήματα κάνοντας χρήση του MPI.