ΑΝΑΦΟΡΑ ΑΣΚΗΣΗΣ 1 ΗΡΥ418

Μανώλης Πετράκος

Δημήτριος Καραμπάσογλου

Περιγραφή κώδικα.

Σε όλες τις υλοποιήσεις οι πίνακες δημιουργούνται δυναμικά. Οι πίνακες των Strings αρχικοποιούνται με τυχαίους χαρακτήρες και ο πίνακας για την αποθήκευση των αποτελεσμάτων μεγέθους m*n αρχικοποιείται με μηδενικά. Ο λόγος που δημιουργούνται δυναμικά(malloc) και όχι στατικά(int diffs[m][n]) είναι ότι το μέγεθος που μπορούν να πάρουν οι πίνακες είναι πολύ μεγάλο και ξεπερνά το μέγεθος του stack του προγράμματος. Η λύση σε αυτό το πρόβλημα είναι είτε να δημιουργηθούν δυναμικά είτε να αυξήσουμε το μέγεθος του stack μέσω της setrlimit(). Ο πρώτος τρόπος αποδείχθηκε αποδοτικότερος και επομένως επιλέχθηκε. Με σκοπό όλες οι υλοποιήσεις να έχουν τα ίδια αποτελέσματα και να επαληθεύεται η ορθότητα τους, χρησιμοποιείται η srand(0) ώστε ο ψευδοτυχαίος αλγόριθμος να δημιουργεί κάθε φορά τα ίδια strings. Τα strings αποτελούνται από δέκα διαφορετικά ψηφία και οι πιθανότητα να βρεθεί διαφορετικό ψηφίο είναι 9/10.

Η σειριακή υλοποίηση δέχεται ως είσοδο τα πλήθη(m, n) και το μέγεθος(l) των Strings. Δημιουργεί τους πίνακες και με τρεις εμφολευμένες for() συγκρίνεται κάθε string από το πρώτο σύνολο με κάθε ένα από το δεύτερο, ψηφίο προς ψηφίο. Για κάθε διαφορετικό ψηφίο αυξάνεται η τιμή του αντίστοιχου κελιού στον πίνακα των αποτελεσμάτων και αφού ολοκληρωθεί η σύγκριση των δυο λέξεων προσθέτεται η τιμή του κελιού στη συνολική απόσταση. Χρονομετρείται μόνο το κομμάτι κώδικα που αποτελείται από τις for() και τέλος εμφανίζονται τα αποτελέσματα.

<u>OpenMP</u>

Για τις υλοποιήσεις με την χρήση OpenMP χρειάζεται ως επιπλέον είσοδος και ο αριθμός των threads. Προσθέτοντας στον σειριακό κώδικα τα κατάλληλα directives, ο compiler τον μετατρέπει σε παράλληλο. Τα threads πλην του master δημιουργούνται με την χρήση του *#pragma omp parallel* και τρέχουν μόνο μέσα στα brackets του. Ανάλογα το μέγεθος του task της υλοποίησης χρησιμοποιείται το directive *#pragma omp for* γύρω από τις for() του σειριακού κώδικα ώστε να μοιραστούν οι επαναλήψεις μεταξύ των threads. Σε όλες τις υλοποιήσεις χρησιμοποιούνται οι ρήτρες *schedule(static),* γιατί ο χρόνος των επαναλήψεων σε κάθε μία είναι σχετικά σταθερός, και *nowait* επειδή δεν υπάρχει λόγος να συγχρονιστούν τα threads στο τέλος των loops. Σε όλες τις περιπτώσεις που χρειάζεται επικοινωνία αυτή αποτελείται από λίγες και απλές πράξεις, έτσι το directive *#pragma omp critical* μπορεί να αντικατασταθεί με το *#pragma omp atomic* το οποίο υλοποιείται στο hardware και είναι γρηγορότερο.

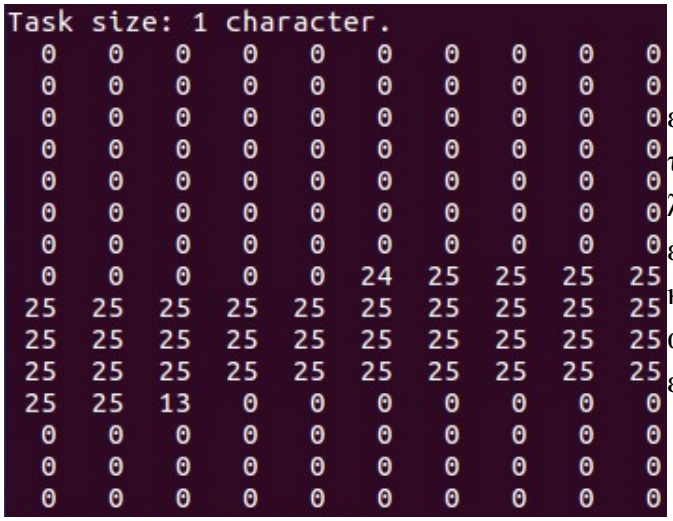
Αν η άθροιση της συνολικής απόστασης hamming γίνει όπως στο σειριακό κώδικα, για κάθε σύγκριση λέξεων κάθε thread πρέπει να επεξεργαστεί το συνολικό αποτέλεσμα, το οποίο είναι κοινό μεταξύ τους. Όταν δύο ή παραπάνω νήματα προσπαθούν να αλλάξουν μια κοινή μεταβλητή δημιουργείται race condition το οποίο για να αντιμετωπιστεί χρειάζεται επικοινωνία μεταξύ τους. Με σκοπό αυτή να μειωθεί, δημιουργείται στο κάθε thread μια τοπική μεταβλητή(threadDiff), η οποία κρατάει το άθροισμα όλων των αποστάσεων που έχει βρει, και με το πέρας των επαναλήψεων προσθέτεται στη συνολική απόσταση. Ως αποτέλεσμα γίνεται μόνο μία φορά ανεξαρτήτως του αριθμού των επαναλήψεων.

Για την υλοποίηση με μέγεθος task μία γραμμή, το directive *#pragma omp for* τοποθετείται πάνω από την πρώτη for() με σκοπό να μοιραστεί στα threads. Το καθένα παίρνει περίπου γραμμές προς threads επαναλήψεις. Η συγκεκριμένη υλοποίηση πάσχει από πιθανό load imbalance λόγο του ότι το task είναι μεγάλο. Για παράδειγμα μια ακραία περίπτωση είναι το σύνολο Α να αποτελείται μόνο από ένα string και το σύνολο Β από πολλά. Έτσι στον πίνακα αποτελεσμάτων θα δημιουργηθεί μόνο μια γραμμή και όλη η επεξεργασία θα γίνει από ένα νήμα, ανεξάρτητα από τον αριθμό τους. Στην περίπτωση αυτή η απόδοση θα είναι ίδια ή χειρότερη από το σειριακό αλγόριθμο. Η επικοινωνία είναι η ελάχιστη δυνατή, δηλαδή η άθροιση των επιμερούς αποτελεσμάτων.

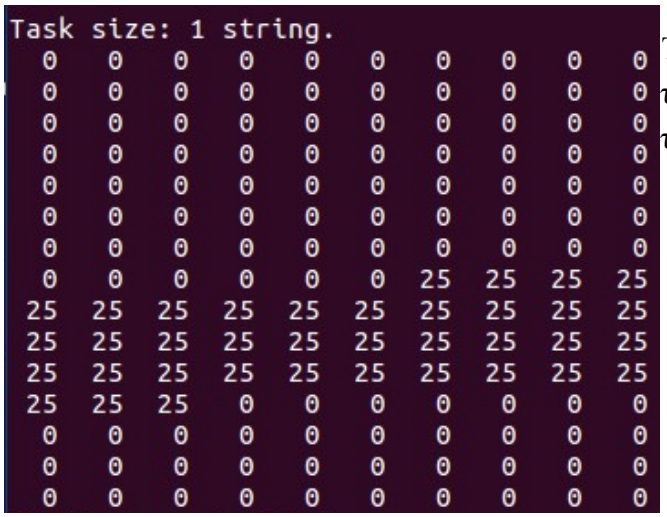
Για την υλοποίηση με μέγεθος task ένα κελί, το directive *#pragma omp for* τοποθετείται στο ίδιο σημείο αλλά προστίθεται η ρήτρα *collapse(2).* Τα πρώτα δυο for() ενόνονται και μοιράζονται. Οι επαναλήψεις που αναλογούν σε κάθε thread είναι γραμμές επί στύλες δια τον αριθμό των threads. Υπάρχει πιθανότητα load imbalance αν υπάρχουν πολύ λίγα strings με μεγάλο μέγεθος. Για παράδειγμα, αν υπάρχει ένα string σε κάθε ομάδα, θα δουλέψει μόνο ένα νήμα και τα υπόλοιπα δεν θα κάνουν τίποτα. Η επικοινωνία μεταξύ των threads είναι ίδια με την προηγούμενη υλοποίηση.

Στην υλοποίηση με μέγεθος task ένα χαρακτήρα το directive *#pragma omp for* τοποθετείται πάλι στο ίδιο σημείο αλλά αυτή τη φορά με την ρήτρα *collapse(3).* Με αυτόν τον τρόπο ενώνονται και μοιράζονται όλες οι for() και οι επαναλήψεις που αναλογούν σε κάθε thread είναι γραμμές επί στήλες επί τον αριθμό των ψηφίων δια τον αριθμό των threads. Με αυτόν τον τρόπο δεν υπάρχει πιθανότητα load imbalance, καθώς το κάθε νήμα συγκρίνει κάθε φορά ένα ζευγάρι χαρακτήρων από ένα οποιοδήποτε ζευγάρι string. Δημιουργείται όμως προβληματική συμπεριφορά σε πολύ μεγάλο πλήθος δεδομένων, δηλαδή δεν εκτελούνται όλες οι επαναλήψεις. Το πρόβλημα αντιμετωπίζεται προσθέτοντας τη ρήτρα *ordered,* η οποία αλλάζει τη σειρά των επαναλήψεων και εξασφαλίζει την εύρυθμη λειτουργία του αλγόριθμου. Μια άλλη λύση είναι να μετακινηθεί το directive πριν την τελευταία for() και να μοιράζει μόνο αυτήν, αλλά ο αλγόριθμος θα είναι πολύ ευαίσθητος σε load imbalance αν οι λέξεις έχουν λίγα ψηφία. Τέλος επειδή το κάθε κελί πιθανώς επεξεργάζεται από παραπάνω από ένα νήμα υπάρχει ανάγκη για επικοινωνία και δεν είναι δυνατό να αθροιστούν οι επιμέρους αποστάσεις από αυτά. Αντίθετα με τις άλλες υλοποιήσεις αυξάνονται κάθε φορά που εντοπίζεται διαφορετικό ψηφίο.

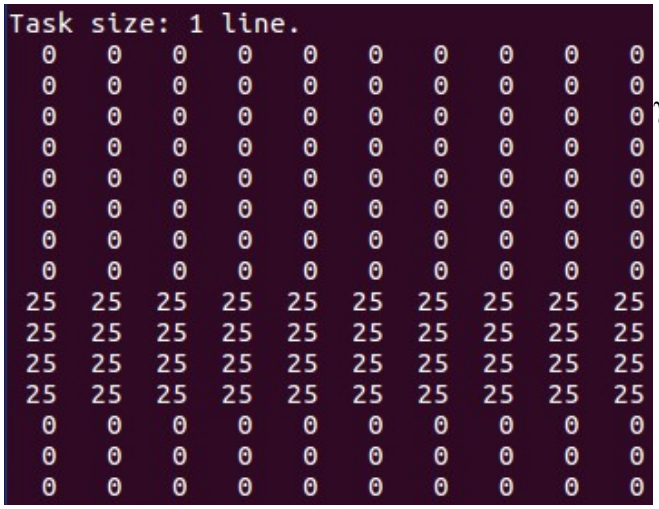
Για τις παρακάτω εικόνες έχει μετατραπεί ο κώδικας ώστε να δουλεύει μόνο το thread με tid = 2, να βρίσκει πάντα διαφορά και να εμφανίζεται ο πίνακας των αποτελεσμάτων. Με αυτόν τον τρόπο φαίνεται ο διαμοιρασμός των tasks. Το μέγεθος είναι 15*10*25 και τα νήματα τέσσερα.



Στη δίπλα εικόνα φαίνεται ότι το thread επεξεργάζεται περίπου το 1/4 των tasks. Αφού το μέγεθος τους είναι ένας χαρακτήρας μερικές λέξεις μοιράζονται μεταξύ των νημάτων και δεν έχουν ολοκληρωθεί από αυτό. Το πολύ δυο κελιά μοιράζεται με άλλα νήματα, με αποτέλεσμα η παραπάνω επικοινωνία να είναι ελάχιστη.



Το thread πάλι επεξεργάζεται περίπου το 1/4 των tasks. Αφού το μέγεθος τους είναι μία λέξη, τις επεξεργάζεται ολόκληρες.



Αυτή τη φορά το μέγεθος των tasks είναι μία γραμμή και τις επεξεργάζεται ολόκληρες.

Pthreads

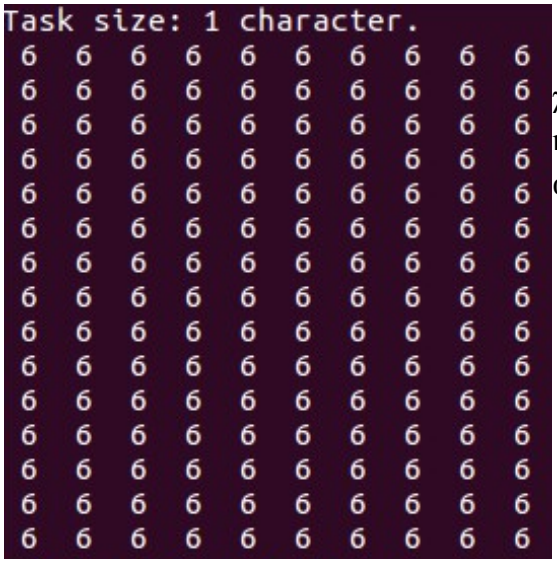
Για τις υλοποιήσεις με την χρήση Pthreads χρειάζεται και πάλι ως επιπλέον είσοδος ο αριθμός των threads. Χρειάζεται ένα struct με τις απαραίτητες πληροφορίες για κάθε thread, όπως οι διευθύνσεις και τα μεγέθη των πινάκων, για να μπορέσουν να υπολογιστούν τα tasks που τους αναλογούν και να δουλέψουν πάνω στους ίδιους πίνακες. Επίσης, μέσα σε μία for() το master thread δημιουργεί όλα τα struct, τα γεμίζει με τις κατάλληλες τιμές και στέλνει με την *pthread_create()* τα νέα νήματα στη συνάρτηση που γίνονται οι υπολογισμοί. Στη συνέχεια πάει και το αρχικό thread στη συνάρτηση υπολογισμού και αφού τελειώσει περιμένει τα υπόλοιπα με την *pthread_join()*. Επειδή το master πριν ξεκινήσει πρέπει να δημιουργήσει και να στείλει τα υπόλοιπα, δεν υπάρχει νόημα να δημιουργηθεί το struct του πρώτο. Για αυτόν τον λόγο η for() είναι ανάποδη, δηλαδή ξεκινάει απο το τελευταίο νήμα και τελειώνει με το πρώτο. Μετά από πολλές μετρήσεις, η ανάποδη υλοποίηση αποδείχτηκε ελαφρά πιο γρήγορη και επιλέχτηκε. Χρησιμοποιούνται οι ίδιες τρεις for() με την σειριακή υλοποίηση, με τη διαφορά ότι τα όρια τους αλλάζουν ανάλογα την περίπτωση. Ακόμα, όλη η επικοινωνία αποτελείται από απλές προσθέσεις και μπορούν να αποφευχθούν τα mutex χρησιμοποιώντας ατομικές εντολές, συγκεκριμένα την *__atomic_fetch_add()*. Στο τέλος της συνάρτησης υπολογισμού "σκοτώνουμε" τα καινούργια threads. Χρονομετρείται ολόκληρη η διαδικασία.

Στην υλοποίηση με μέγεθος task ένα χαρακτήρα, αντίθετα με τις επόμενες, το κάθε thread δεν υπολογίζει ποιες συγκρίσεις θα κάνει. Θα πάρει μέρος στον υπολογισμό όλων των κελιών, ξεκινώντας από το ψηφίο στην ίδια θέση με την τιμή του tid του και με βήμα το πλήθος των νημάτων. Τα κελιά υπολογίζονται από παραπάνω από ένα thread και οι πράξεις γίνονται ατομικά με αποτέλεσμα να αυξηθεί η επικοινωνία. Για τον ίδιο λόγο, το τοπικό άθροισμα πρέπει να αυξάνεται σε κάθε διαφορετικό ψηφίο. Δημιουργείται load imbalance αν ο αριθμός των ψηφίων δεν διαιρείται απόλυτα με τον αριθμό των νημάτων και στην ακραία περίπτωση που υπάρχει μόνο ένα ψηφίο σε κάθε string, επειδή δουλεύει μόνο ένα thread.

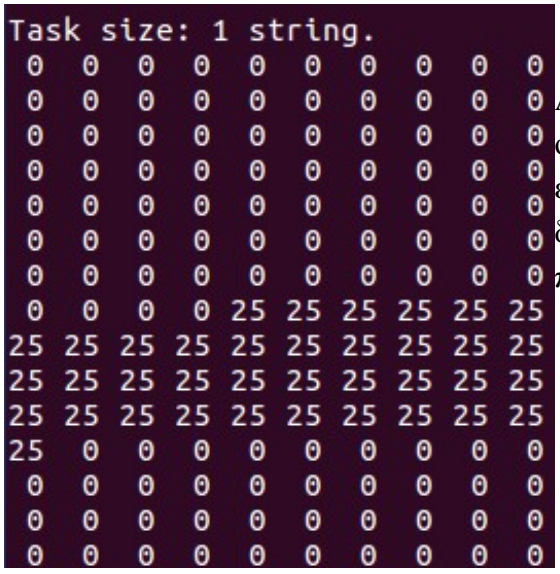
Για την υλοποίηση με μέγεθος task ένα κελί, υπολογίζεται από το κάθε thread ποιο είναι το αρχικό και το τελευταίο κελί καθώς και η αρχική γραμμή. Ξεκινώντας από αυτή τη γραμμή, το νήμα αγνοεί τα κελιά πριν από το αρχικό, αν αυτά υπάρχουν, και συνεχίζει μέχρι το τελευταίο. Τα tasks μοιράζονται διαιρώντας το πλήθος τους με τον αριθμό των νημάτων, αν η διαίρεση δεν είναι τέλεια αυτά που θα περισσέψουν ανατέθοντε στο τελευταίο thread. Τα νήματα επειδή δημιουργούνται και στέλνονται ανάποδα έχει ένα μικρό χρονικό περιθώριο να τελειώσει την παραπάνω δουλειά πριν τα υπόλοιπα ξεκινήσουν. Ακόμα μειώνεται η επικοινωνία προστίθοντας τις αποστάσεις των λέξεων σε μια τοπική μεταβλητή η οποία στο τέλος προστίθεται στη συνολική απόσταση. Και αυτή η υλοποίηση αντιμετωπίζει τα ίδια προβλήματα με του OpenMP.

Για την υλοποίηση με μέγεθος task μία γραμμή, το κάθε νήμα υπολογίζει ποιά είναι η πρώτη και η τελευταία γραμμή που θα επεξεργαστεί. Τα tasks μοιράζονται διαιρώντας το πλήθος τους με τον αριθμό των νημάτων και στρογγυλοποιώντας προς τα πάνω. Επειδή το μέγεθος του task είναι μεγάλο, δεν είναι βέλτιστο να πάρει αυτά που περισσεύουν το τελευταίο νήμα. Όπως και στην προηγούμενη υλοποίηση, οι αποστάσεις που υπολογίστηκαν προστίθενται σε μια τοπική μεταβλητή και στο τέλος αυτή προστίθεται στη συνολική απόσταση για την μείωση της επικοινωνίας. Η υλοποίηση αυτή αντιμετωπίζει τα ίδια προβλήματα με του OpenMP.

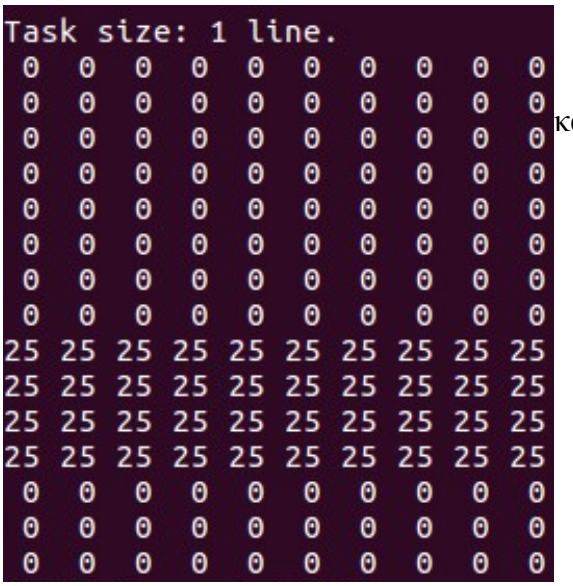
Ο κώδικας έχει μετατραπεί αντίστοιχα για τα Pthreads. Το μέγεθος είναι 15*10*25 και τα νήματα τέσσερα. Δουλεύει μόνο το thread με tid = 2.



Το νήμα επεξεργάζεται περίπου το 1/4 κάθε κελιού. Οι χαρακτήρες που επεξεργάζεται είναι στις θέσεις 2, 6, 10 κλπ. Μοιράζεται όλα τα κελιά με όλα τα νήματα με αποτέλεσμα να υπάρχει πολύ επικοινωνία μεταξύ τους.



Το thread πάλι επεξεργάζεται περίπου το 1/4 των tasks. Αφού το μέγεθος τους είναι μία λέξη, τις επεξεργάζεται ολόκληρες. Υπάρχει μια μικρή διαφορά στο πια κελιά έχουν επεξεργαστεί σε σχέση με το OpenMP επειδή η διαίρεση που μοιράζει τα task δεν είναι τέλια. Αυτά που περισσεύουν έχουν ανατεθεί στο τελευταίο νήμα.

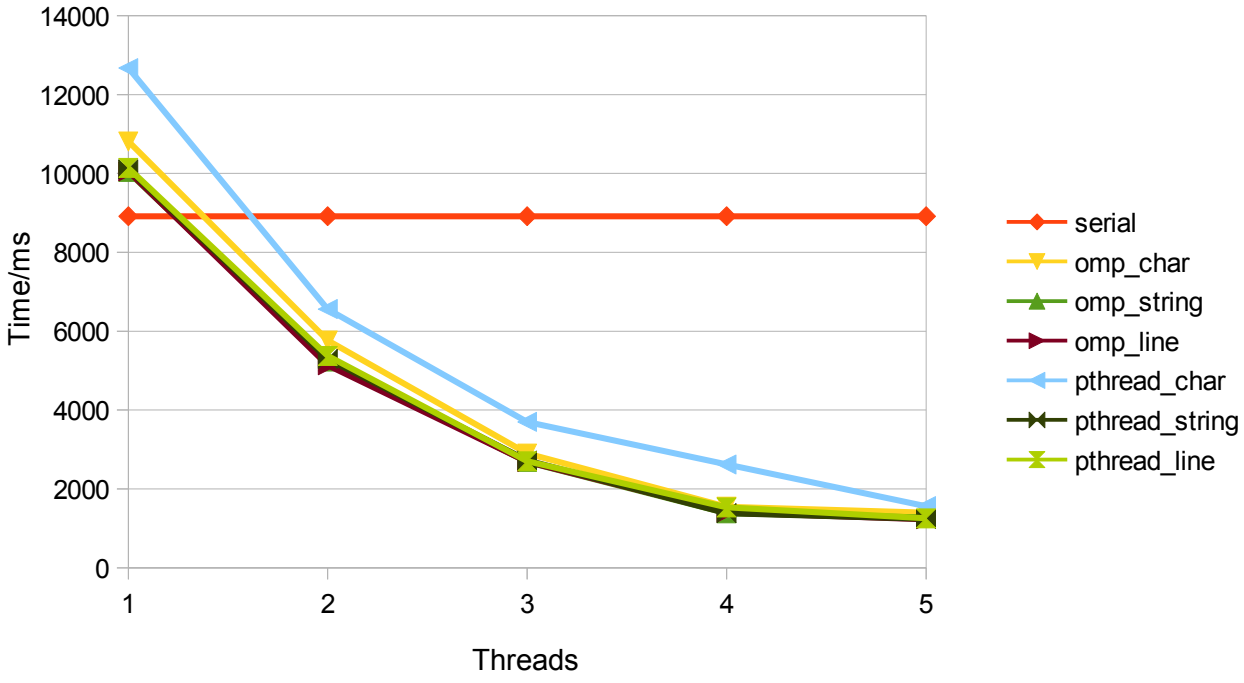


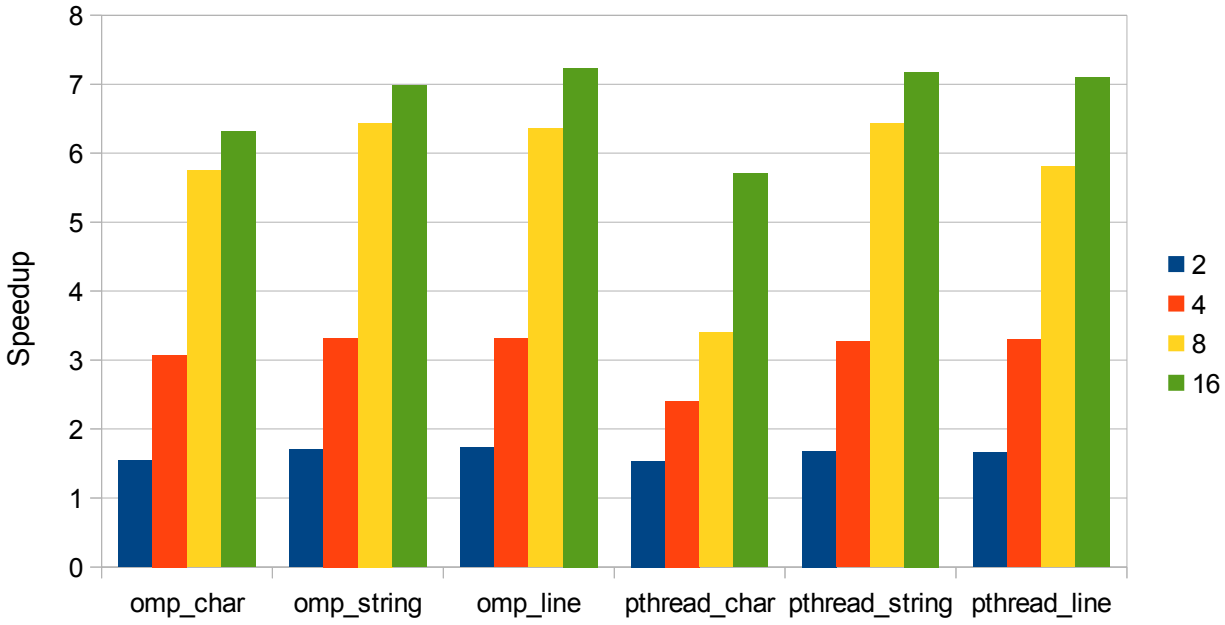
Αυτή τη φορά το μέγεθος των tasks είναι μία γραμμή και τις επεξεργάζεται ολόκληρες.

Αποτελέσματα

Ο υπολογιστής όπου γίνονται οι μετρήσεις έχει 8 πυρήνες.

Το παρακάτω γράφημα δείχνει τους χρόνους υπολογισμού(milisecond) για κάθε υλοποίηση με εισόδους: m = 1000, n = 1000, l = 1000 .



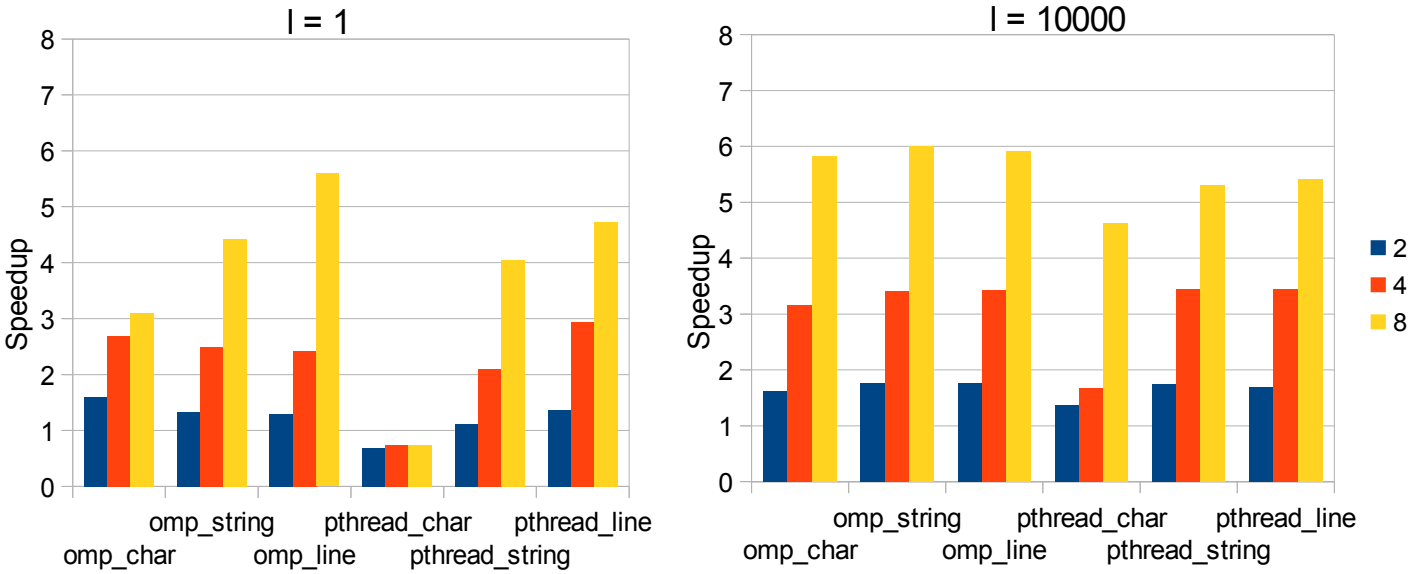


Για ένα απλό dataset όλοι οι αλγόριθμοι αποδίδουν παρόμοια, με εξαίρεση τον pthread_char λόγο της υψηλής επικοινωνίας μεταξύ των threads του. Η αύξηση του speedup από 8 σε 16 πυρήνες οφείλεται στο hyperthreading και ότι μοιράζονται περισσότερο τα δεδομένα. Γενικά τα αποτελέσματα είναι αναμενόμενα, περίπου στα 3/4 του max speedup. Επίσης για 1 thread όλοι οι παράλληλοι αλγόριθμοι είναι πιο αργοί επειδή πληρόνουν το overhead για την δημιουργία των νημάτων κλπ χωρίς να μπορούν να εκμεταλλευτούν την παραλληλία.

Για να παρατηρήσουμε την συμπεριφορά των αλγορίθμων σε διαφορετικές δομές δεδομένων έχουμε τα παρακάτω γραφήματα speedup.

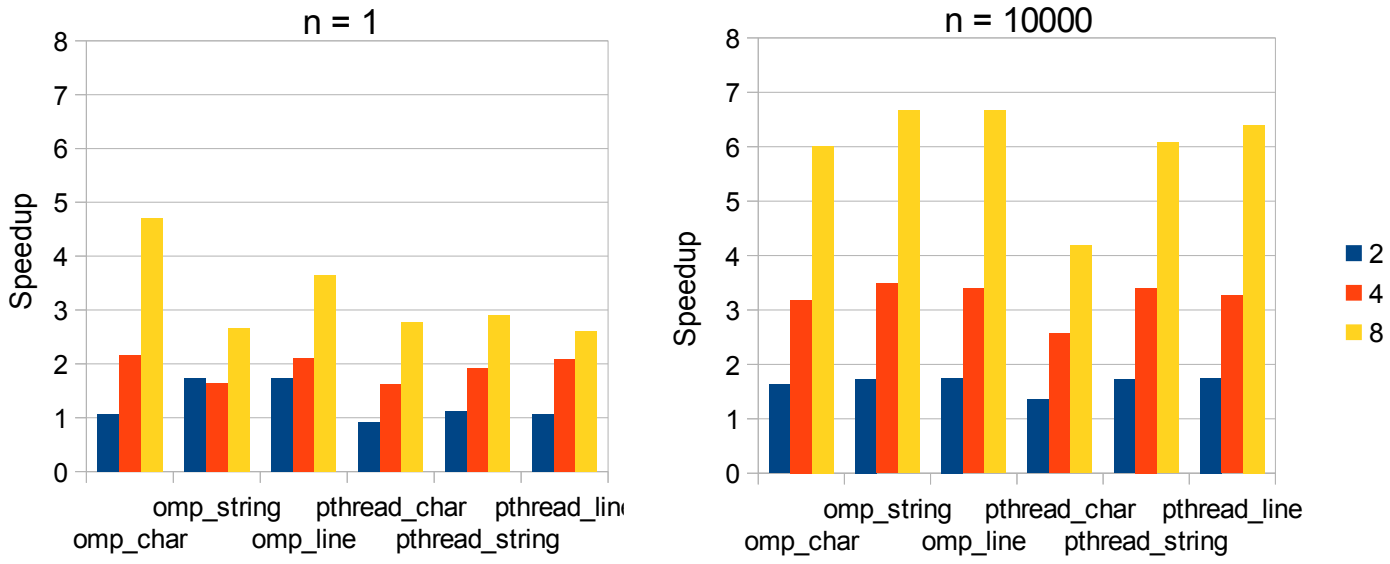
Γενικά το speedup είναι μικρότερο στα αριστερά γραφήματα επειδή το πλήθος των συγκρίσεων είναι μικρότερο και το overhead πιο σημαντικό. Το overhead αποτελείται από τον χρόνο που χρειάζονται τα νήματα για να δημιουργηθούν, να ξεκινήσουν και να υπολογίσουν ποια είναι τα tasks τους.

1) M = 1000, N = 1000, L μεταβαλλόμενο



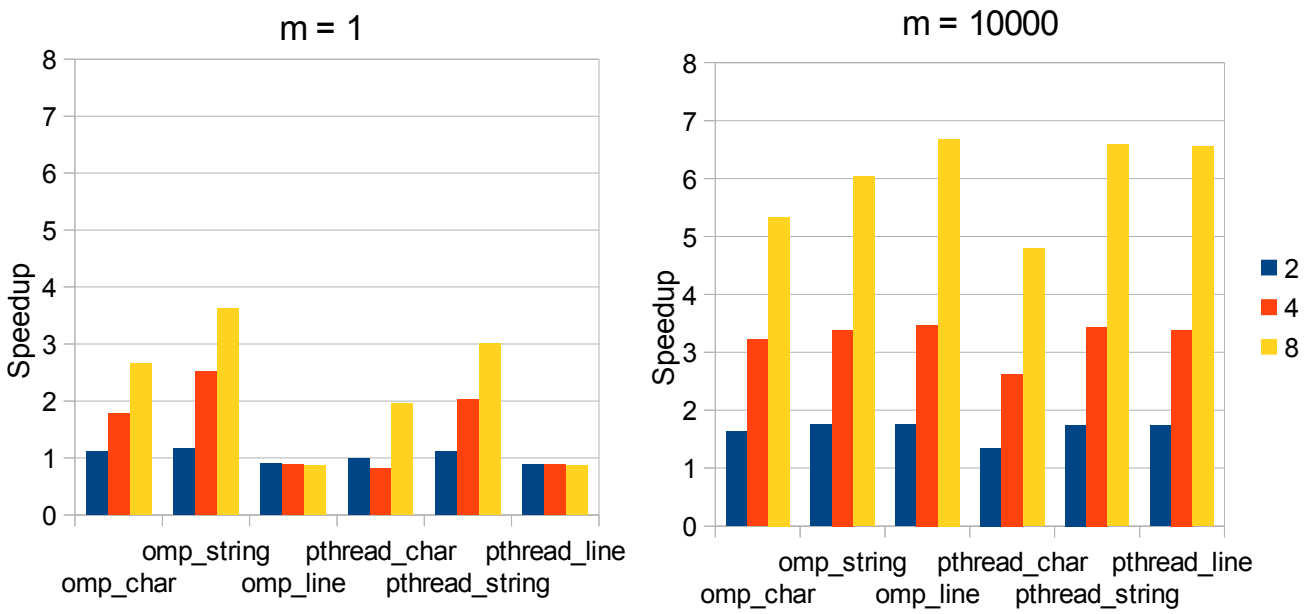
Παρατηρούμε ότι ο pthread_char αλγόριθμος είναι πιο αργός και από το σειριακό όταν το μέγεθος των λέξεων είναι πολύ μικρό. Τα tasks μοιράζονται στατικά, δουλεύει μόνο ένα thread και με το overhead που υπάρχει για να δημιουργήσει και να στείλει το νήμα, γίνεται ακόμα πιο αργός. Αντίθετα στον omp_char τα thread διαλέγουν χαρακτήρες από οποιαδήποτε λέξη και σε οποιαδήποτε θέση με αποτέλεσμα να μην παρουσιάζουν τέτοιο πρόβλημα.

2) M = 1000, L = 1000, N μεταβαλλόμενο



Παρατηρούμε ότι το πλήθος των γραμμών δεν επηρεάζει το speedup ιδιαίτερα. Αυτό που επηρεάζει τις μετρήσεις είναι ότι ο αριθμός των πράξεων είναι μικρότερος στο αριστερό γράφημα.

3) N = 1000, L = 1000, M μεταβαλλόμενο



Βλέπουμε ότι μειώνοντας τον αριθμό των στηλών οι αλγόριθμοι που έχουν ως task μια στήλη επηρεάζονται σημαντικά και γίνονται πολύ αργοί.

Γενικά από τα γραφήματα συμπεραίνουμε ότι η παράλληλοι αλγόριθμοι γίνονται πιο αποτελεσματικοί όσο ο όγκος των δεδομένων μεγαλώνει. Επίσης η απόδοση τους εξαρτάται από τη δομή των δεδομένων. Σύμφωνα με τις μετρήσεις μας, την καλύτερη επίδοση έχουν οι αλγόριθμοι με μέγεθος task ίσο με ένα string, ανεξαρτήτως αν είναι υλοποιημένοι με OpenMP ή Pthreads. Έχουν το καλύτερο computation to communication ratio καθώς η μόνη επικοινωνία που χρειάζεται είναι η άθροιση των μερικών αθροισμάτων που έχουν τα νήματα και το task είναι αρκετά μικρό ώστε να μην δημιουργείται load imbalance.

Σημείωση: Δεν φτιάξαμε διαγράμματα για την περίπτωση όπου φαίνεται η αδυναμία των αλγορίθμων με μέγεθος task μία λέξη(m = 1, n = 1, l = 10000) γιατί ο αριθμός των πράξεων είναι πολύ μικρός και όλοι οι αλγόριθμοι έχουν κακή επίδοση με αποτελέσματα να βγαίνουν σωστά συμπεράσματα. Επίσης, και να μεγαλώσει το l περισσότερο, κανένας αλγόριθμος δεν θα αποδώσει γιατί οι line και string θα έχουν μόνο ένα task και οι char θα έχουν τεράστια επικοινωνία.

Τέλος δεν συμπεριλάβαμε τα διαγράμματα με εισόδους m = 10000, n = 10000, l = 10000 διότι τα αποτελέσματα του speedup είναι ίδια με την πρώτη περίπτωση και δεν έχει νόημα να συμπεριληφθούν.

Συμπεράσματα και παρατηρήσεις.

Το OpenMP API είναι εύκολο και γρήγορο στην εκμάθηση, στην χρήση και την συντήρηση του κώδικα. Ο τρόπος γραφής του είναι πολύ ξεκάθαρος αλλά δεν υπάρχουν ιδιαίτερα περιθώρια για βελτιστοποιήσεις λόγω περιορισμών. Επειδή δεν γνωρίζουμε πως συμπεριφέρεται στο background, μπορούν να υπάρξουν προβλήματα για τα οποία δεν ευθύνεται άμεσα ο κώδικας μας. Για παράδειγμα, ο αλγόριθμος omp_char λειτουργούσε σωστά μέχρι ένα μέγεθος δεδομένων και μόλις έφταναν ένα συγκεκριμένο νούμερο η ρήτρα collapse() δεν συμπεριφερόταν σωστά.

Αντίθετα τα Pthreads δεν έχουν την ίδια αυτοματοποίηση και είναι πιο χρονοβόρα στη δημιουργία του κώδικα. Για παράδειγμα, τα struct με τις πληροφορίες που χρειάζεται κάθε νήμα πρέπει να φτιαχτούν και να συμπληρωθούν από τον προγραμματιστή. Τα πλεονεκτήματά τους είναι η δυνατότητα αλλαγής και βελτίωσης λεπτομερειών, καθώς και γνωρίζουμε πως ακριβώς είναι και λειτουργεί ο κώδικας.