Σχολή ΗΜΜΥ, Πολυτεχνείο Κρήτης
ΗΡΥ 418 - Αρχιτεκτονική Παράλληλων και Κατανεμημένων Υπολογιστών (Εαρινό εξάμηνο 2017-2018)
Διδάσκων : Ν. Αλαχιώτης
Ονόματα φοιτητών : Καλλέργης Γεώργιος, Σκυβαλάκης Κωνσταντίνος
Α.Μ. φοιτητών : 2013030054, 2013030034
Ημερονημία παράδοσης : 28 Μαρτίου 2018
Άσκηση 1

1 Περιγραφή Κώδικα

Στην 1η άσκηση μελετήσαμε διαφορετικούς τρόπους για να παραλληλοποιήσουμε έναν σειριακό κώδικα υπολογισμού Hamming αποστάσεων μεταξύ δυο strings κάθε φορά, από δυο διαφορετικά σετ με strings A και B. Σύμφωνα με την εκφώνηση δημιουργήσαμε 3 διαφορετικούς παράλληλους κώδικες με τη χρήση του OpenMP και 3 με τη χρήση των POSIX Threads. Γενικά, η υλοποίηση των OpenMP κωδίκων ήταν αρκετά ευκολότερη και πολύ μικρότερη σε έκταση απ΄ότι η αντίστοιχη υλοποίηση με pthreads καθώς το OpenMP API είναι ουσιαστικά ένα black box όπου δίνει τη δυνατότητα στον χρήστη να εκμεταλλευτεί την παραλληλία χωρίς ιδιαίτερο κόπο τις περισσότερες φορές. Παρακάτω θα αναφερθούμε αναλυτικότερα στις διαφορές μεταξύ computation-to-communication ratio συγκρίνοντας OpenMP και pthreads.

Όσον αφορά, τα σημεία που έπρεπε να γίνει ο παραλληλισμός, θεωρήσαμε πως ήταν αρκετά προφανές ότι θα γίνουν στα συγκεκριμένα σημεία στις for, κάτι που επιβεβαιώσαμε όταν βγήκε η ανακοίνωση. Επί της ουσίας ήταν το κομμάτι στο οποίο γινόταν όλοι οι υπολογισμοί και οι συγκρίσεις ανάμεσα στα strings των 2 arrays, επομένως αν θέλαμε μία ταχύτερη εκτέλεση θα την επιτυχάναμε μειώνοντας το χρόνο εκτέλεσης στα εν λόγω σημεία, είτε στο βρόγχο που συγκρίνει μια συμβολοσειρά του A με όλες του B, είτε μία του A με μία του B, είτε χαρακτήρες αυτών. Για αυτόν ακριβώς το λόγο επικεντρωθήκαμε σε αυτό το κομμάτι κώδικα.

Για το OpenMP, δώσαμε την εντολή pragma στον κώδικα που θέλουμε να παραλληλοποίησουμε. Πιο συγκεκριμένα, μαζί με το παραπάνω χρησιμοποίησαμε το for όπως αναφέρεται και στο documentation για περιπτώσεις όπως η δική μας, το num_threads για να δώσουμε τον αριθμό των threads που θέλουμε, reduction για να δείξουμε την μεταβλητή το μέγεθος της οποίας αυξάνεται (dist) και private για την μεταβλητή του εσωτερικού loop, όπου χρειάστηκε. Αναζητώντας στο documentation τρόπους για να κάνουμε βέλτιστη την υλοποίηση μας, εκτός των παραπάνω δοκιμάσαμε να εφαρμόσουμε και schedule με διάφορα ορίσματα όπως auto, dynamic και static. Ωστόσο, δεν στάθηκαν ικανά να μεταβάλλουν σημαντικά το execution time για να συμπεριληφθούν στην υλοποίησή μας.

Όσον αφορά τα pthreads, έπρεπε να διαχειριστούμε εμείς τα threads. Για τη δημιουργία αυτών, καλούσαμε την pthread_create στην οποία περνούσαμε σαν όρισμα το thread, το όνομα του task που έπρεπε να εκτελέσει και το struct με τα ορίσματα της συνάρτησης. Δίναμε το struct καθώς η συγκεκριμένη συνάρτηση παίρνει ένα μόνο όρισμα, ωστόσο εμείς έπρεπε να περνάμε αρκετά περισσότερα για την επιτυχή διεκπεραίωση του task. Όσον αφορά τον όγκο δουλειάς που έπρεπε να φέρει εις πέρας κάθε thread, καθοριζόταν από το πόσα threads έχουμε και πόση δουλειά συνολικά, επομένως το κάθε ένα αναλάμβανε ένα 'μπλοκ' εργασιών ανάλογα με το id του . Πιο συγκεκριμένα, αν έπρεπε να γίνουν 160 υπολογισμοί από 4 νήματα, το δεύτερο νήμα θα έπρεπε να κάνει τους υπολογισμούς από 40-79. Επιλέξαμε η pthread_create να μην γυρνάει κάτι καθώς γράφουμε στο ηαμμ που τον έχουμε ορίσει ως global double pointer. Στη συνέχεια, γίνονται join τα threads με την pthread_join.

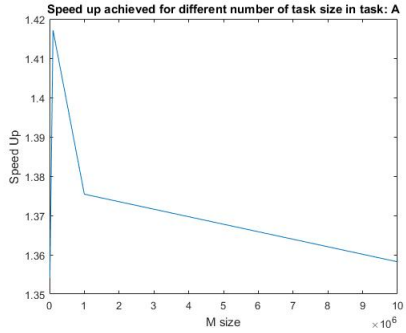
Τέλος, δημιουργήσαμε και ένα run_script.sh σε bash shell για να κάνουμε compile και να τρέχουμε τους κώδικες μας ταυτόχρονα συγκρίνοντας έτσι κάθε φορά τα αποτελέσματα τους. Μέσα σε αυτό το script δίνουμε τις τιμές των παραμέτρων του προγράμματος οι οποίες οι εξής: $m, n, l, num\_threads, seed$ και $print\_flag$. Αν και προφανής, η χρήση των παραμέτρων αυτών περιγράφεται εν συντομία σε σχόλια μέσα στο run_script.sh.

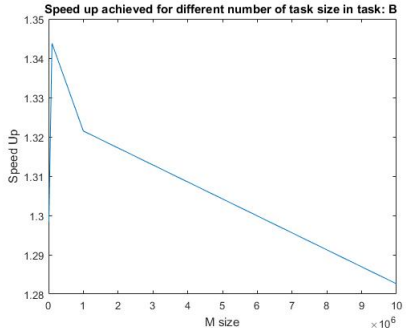
2 Screenshots και Γραφικές αποτελεσμάτων

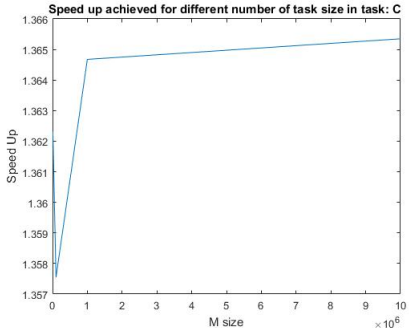
2.1 Run του κώδικα για m=1000 n=1000 l=100 με 2 threads.

```
You chose to use seed=1, m=1000, n=1000, l=100, print=0, number of threads=2
############################################################
Serial Code Run
Time: 0.362277 seconds
############################################################
OpenMP Parallel Code Run (a)
Time: 0.912088 seconds
############################################################
OpenMP Parallel Code Run (b)
Time: 0.226611 seconds
############################################################
OpenMP Parallel Code Run (c)
Time: 0.236111 seconds
############################################################
Pthreads Parallel Code Run (a)
Time: 18.184762 seconds
############################################################
Pthreads Parallel Code Run (b)
Time: 0.219501 seconds
############################################################
Pthreads Parallel Code Run (c)
Time: 0.195557 seconds
```

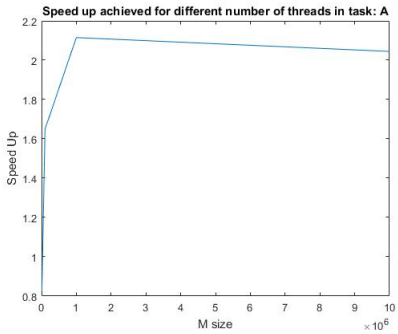
2.2 Μεταβολή του speed up συναρτήσει του task size για το OMP

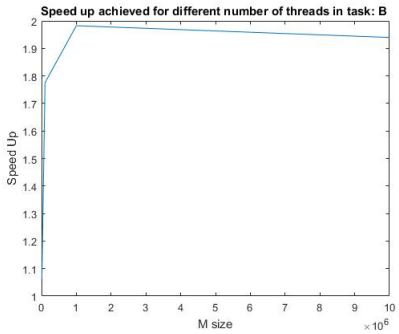


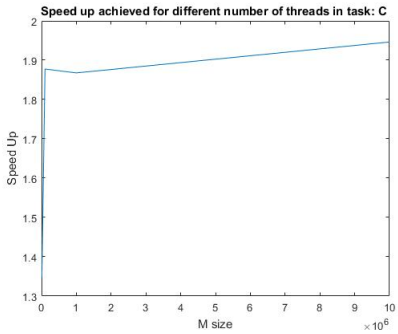




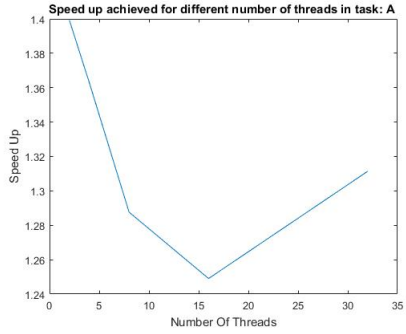
2.3 Μεταβολή του speed up συναρτήσει του task size για το PTHREADS

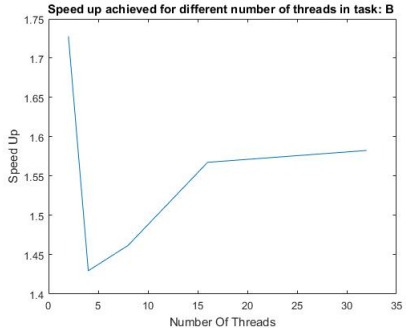


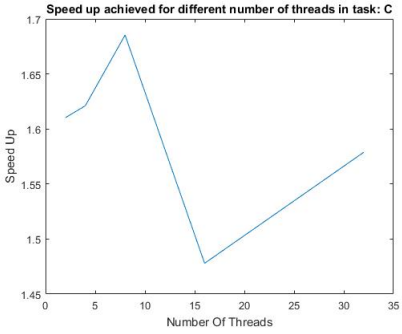




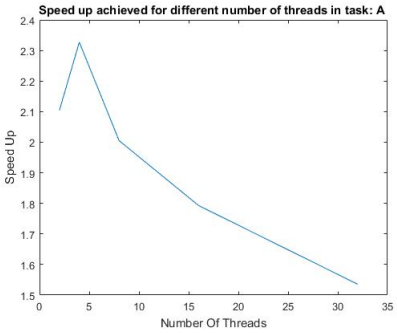
2.4 Μεταβολή του speed up συναρτήσει του αριθμού των threads για το OMP

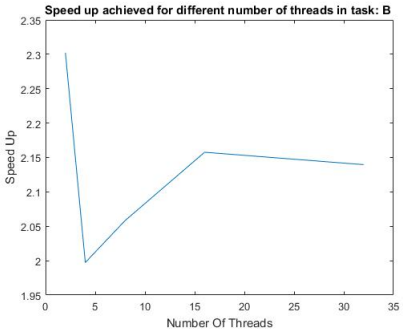


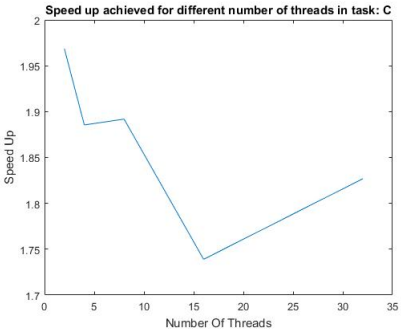




2.5 Μεταβολή του speed up συναρτήσει του αριθμού των threads για το PTHREADS







3 Σχολιασμός Αποτελεσμάτων

Αρχικά, θα πρέπει να διευκρινήσουμε ότι κάθε ένας από τους τρεις παράλληλους κώδικες, είτε με χρήση OpenMP είτε με χρήση pthreads εκμεταλλεύεται σε διαφορετικά σημεία του σειριακού κώδικα τον παραλληλισμό των tasks. Ως αποτέλεσμα, για διαφορετικές τιμές εισόδου (m, n, l) να έχουμε διαφορετική αποδοτικότητα σε κάθε περίπτωση. Αναλυτικότερα, στην περίπτωση του task c, τα νήματα πρέπει για κάθε string από τα m συνολικά του σετ A να κάνουν ένα μεγάλο αριθμό από υπολογισμούς συγκρίνοντας το με όλο το δεύτερο σετ B, χαρακτήρα προς χαρακτήρα, επομένως όπως γίνεται αντιληπτό, είναι coarse grained. Αντίθετα, το άλλο άκρο αποτελεί το granularity που αναμένουμε από το task a. Στο εν λόγω task, ο όγκος των υπολογισμών που πρέπει να γίνουν σε αυτό είναι σημαντικά μικρότερος, αφού έχουμε μόνο τη σύγκριση των χαρακτήρων στις 2 συμβολοσειρές που το καθιστά fine grained. Τέλος, μία ενδιάμεση κατάσταση αποτελεί το task b στο οποίο σίγουρα αναμένουμε καλύτερο granularity από το task a αλλά χειρότερο από το task c.

Επιπλέον, πέρα από την αρχική εκτίμηση σημαντικό ρόλο διαδραματίζει και το task size. Αυτό που παρατηρούμε είναι ότι κάθε παράλληλος κώδικας εκμεταλλεύεται προφανώς σε διαφορετικό σημείο την παραλληλία και έτσι δεν είναι απόλυτα δυνατό να συγκρίνουμε με ίδιες εισόδους όλους τους παράλληλους κώδικες μεταξύ τους. Επομένως, οφείλουμε να εξετάσουμε σε κάθε περίπτωση τους αντίστοιχους OpenMP και pthreads κώδικες με τον σειριακό και να υπολογίσουμε το speedup, αποσκοπώντας κάθε φορά να εκμεταλλευτούμε διαφορετικό σημείο παραλληλίας. Λόγου χάρη, ο κώδικας του (α) εκμεταλλεύεται την παραλληλία στον έλεγχο των χαρακτήρων δυο strings που σημαίνει ότι άμα δώσουμε $m = 100$, $n = 100$ και $l = 10$ κατά πάσα πιθανότητα δε θα δούμε κανένα κέρδος και μάλιστα ίσως γίνει και χειρότερο αν για τον έλεγχο μόνο 10 χαρακτήρων χρησιμοποιήσουμε OpenMP και pthreads. Αντιθέτως, στην ίδια περίπτωση αν δώσουμε και $l = 1000000$ τότε θα δούμε σίγουρα διαφορά στην εκτέλεση γιατί πλέον ο παραλληλισμός μας θα έχει νόημα καθώς θα έχουμε αυξήσει επαρκώς το task size και θα μπορέσουμε να το εκμεταλλευτούμε κατάλληλα για το μοίρασμα του. Παρόμοιες περιπτώσεις έχουμε στα (β) και (γ) ερωτήματα όπου και εκεί χρειαζόμαστε μεγάλο πλήθος συμβολοσειρών στα σετ A και B αντίστοιχα για να δούμε κάποιο ουσιώδες κέρδος στον παραλληλισμό μας. Όπως φαίνεται και στα screenshots που παρατίθενται παρακάτω για το παραδειγμα που δώσαμε προηγουμένως, βλέπουμε πως για σταθερά m=n=100, για μικρό l=100, το σειριακό είναι αρκετά πιο γρήγορο από το παράλληλο! Αντίθετα, για μεγάλο task size, l=1000000 έχουμε καλύτερο speed up, κατά 0.6 στο OpenMP και 1.2 φορές καλύτερο στα pthreads.

Συνεχίζοντας, καταλυτικό ρόλο παίζει ο αριθμός των threads που χρησιμοποιούνται σ΄ ένα πρόγραμμα. Αυτό που αναμένουμε είναι ότι με περισσότερα νήματα θα έχουμε ταχύτερη εκτέλεση του παράλληλου κώδικα σε σχέση με το σειριακό. Όπως παρατηρείται στα παραπάνω γραφήματα, αυξάνοντας τον αριθμό τους, βελτιώνεται το speed up και ανάλογα με το τασκ διαφοροποιείται το peak του. Από εκείνο το σημείο και μετά, όσο αυξάνει ο αριθμός των threads, μειώνεται η απόδοση του παραλληλισμού. Για παράδειγμα, για το omp, στα task A, C μεγιστοποιείται στα 16 νήματα, ενώ στο task B στα 4. Αυτό οφείλεται στο γεγονός ότι περισσότερα νήματα απαιτούν περισσότερο communication μεταβάλλοντας το communication to computation ratio και εισάγει περισσότερα overheads.

Όσον αφορά τη διαφορά του speed-up στο pthreads και στο OpenMP, αυτή έγκειται σε πολλούς παράγοντες. Ένας από αυτούς είναι ότι το OpenMP δημιουργεί τόσα threads όσα και οι πυρήνες αν δε δώσουμε εμείς το όρισμα που αναφέραμε παραπάνω. Για αυτό όταν αυξάνεται σημαντικά ο αριθμός των threads παρατηρούμε μια σημαντική, αρνητική επίδραση στο speed up πέρα από τους λόγους που εξηγήσαμε παραπάνω (communication to computation ratio) . Αντίθετα, παρά το ότι επηρεάζεται, επίσης αρνητικά, η επίδοση των pthreads, δεν είναι τόσο έντονη όσο στο πρώτο όπως φαίνεται και στους πίνακες. Επίσης, λόγω αυτής της ιδιότητας του OpenMP παρατηρήσαμε από τις εκτελέσεις χωρίς να δώσουμε αριθμό νημάτων ότι είναι κάπως σταθερό το speed-up του, με μια μικρή διακύμανση, διαφορά που οφείλεται κυρίως σε άλλους παράγοντες όπως στην επίδραση άλλων προγραμμάτων που τρέχουν στο ίδιο διαστημα και χρησιμοποιούν τους ίδιους πόρους. Τέλος, στη διαφορά αυτών των 2 συμβάλλει και το ότι το OpenMP είναι higher level συγκριτικά με τα pthreads, και βέβαια ότι το δεύτερο είναι "thread based" ενώ το άλλο "task based". Να τονίσουμε κλείνοντας πως σιγουρευτήκαμε ότι τα αποτελέσματα των παράλληλων κωδίκων είναι σωστά καθώς τα ελέγξαμε με το αποτέλεσμα του σειριακού κώδικα για μικρές τιμές των παραμέτρων εκτυπώνοντας τον πίνακα με τις αποστάσεις κάθε φορά.

Για αρκετά μεγάλες τιμές των παραμέτρων ίσως χρειαστεί να θέσουμε το stack size σε unlimited ως εξής: ulimit -s unlimited, μιας και η default τιμή του είναι 8192 bytes και εμείς χρειαζόμαστε αρκετά περισσότερα καθώς δημιουργούμε τους πίνακες με τα strings.

```
george@george-Lenovo-G580:~/Desktop/Distributed-and-Parallel-Systems$ ./run_script.sh 1 100 100 100 0 2
You chose to use seed=1, m=100, n=100, l=100, print=0, number of threads=2
############################################################
Serial Code Run
Time: 0.004831 seconds
############################################################
OpenMP Parallel Code Run (a)
Time: 0.011358 seconds
############################################################
OpenMP Parallel Code Run (b)
Time: 0.003836 seconds
############################################################
OpenMP Parallel Code Run (c)
Time: 0.003586 seconds
############################################################
Pthreads Parallel Code Run (a)
Time: 0.182413 seconds
############################################################
Pthreads Parallel Code Run (b)
Time: 0.004928 seconds
############################################################
Pthreads Parallel Code Run (c)
Time: 0.003843 seconds
```

```
george@george-Lenovo-G580:~/Desktop/Distributed-and-Parallel-Systems$ ./run_script.sh 1 100 100 1000000 0 2
You chose to use seed=1, m=100, n=100, l=1000000, print=0, number of threads=2
############################################################
Serial Code Run
Time: 37.440025 seconds
############################################################
OpenMP Parallel Code Run (a)
Time: 26.515912 seconds
############################################################
OpenMP Parallel Code Run (b)
Time: 24.357339 seconds
############################################################
OpenMP Parallel Code Run (c)
Time: 24.485634 seconds
############################################################
Pthreads Parallel Code Run (a)
Time: 17.337043 seconds
############################################################
Pthreads Parallel Code Run (b)
Time: 17.094196 seconds
############################################################
Pthreads Parallel Code Run (c)
Time: 19.336334 seconds
```