Παράλληλα Συστήματα Εργασία 2: OpenMP

Άσκηση 2.1

Υλοποίηση: Εμμανουήλ-Ταξιάρχης Οζίνης (sdi2300147)

Περιγραφή προβλήματος

Το πρόβλημα μπορεί να χωριστεί στα εξής σημαντικά υποπροβλήματα:

- Την διατύπωση και υλοποίηση του σειριακού αλγορίθμου για πολλαπλασιασμό πολυωνύμων.
- Την **παραλληλοποίηση** του αλγορίθμου σε ικανοποιητικό βαθμό, δηλαδή με τέτοιο τρόπο που το παράλληλο τμήμα του είναι σημαντικά μεγαλύτερο από το μη παραλληλοποιημένο.

Περιγραφή λύσης

Για την οργάνωση της λύσης, αρχικά ορίζονται οι βοηθητικές συναρτήσεις:

- `parse_args()` : Διαβάζει τα command line arguments με κατάλληλους ελέγχους.
- `elapsed()` : Επιστρέφει την χρονική διαφορά σε δευτερόλεπτα (έως ακρίβεια nsec) από το start στο end. Σημειώνεται ότι για τις χρονικές μετρήσεις χρησιμοποιείται η δομή struct timespec της *C11*.
- `generate_random_coef()` : Δίνει τυχαίες μη μηδενικές ακέραιες τιμές σε ένα πολυωνυμο, όπου ο κάθε συντελεστής έχει απόλυτη τιμή το πολύ `MAX_ABS_COEFFICIENT_VALUE` .

Έχει οριστεί δομή `Polynomial` και για την διαχείρησή της υπάρχουν οι ακόλουθες συναρτήσεις:

- `pol_init()` : Δεσμεύει δομή Polynomial χρησιμοποιώντας ήδη δεσμευμένους συντελεστές.
- `pol_destroy()` : Αποδεσμεύει το πολυώνυμο και τους συντελεστές.
- `pol_print()` : Τυπώνει ένα πολυώνυμο, για λόγους debugging
- `pol_equals()` : Ελέγχει αν δύο πολυώνυμα είναι ίσα.
- `pol_add()` : Προσθέτει δύο πολυώνυμα. Τα πολυώνυμα και το αποτέλεσμα πρέπει να είναι ήδη δεσμευμένα. Αυτό γίνεται για λόγους ταχύτητας στον πολλαπλασιασμό.
- `pol_multiply()` : Ο σειριακός αλγόριθμος πολλαπλασιασμού πολυωνύμων. Δεσμεύει ο ίδιος το αποτέλεσμα. Η υλοποίησή του επεξηγείται αργότερα.
- `pol_multiply_threaded()` : Ο παραλληλοποιημένος αλγόριθμος πολλαπλασιασμού. Είναι υπεύθυνος για την δημιουργία και "συλλογή" των νημάτων, και επομένως όλα τα `#pragma omp` γίνονται εσωτερικά σε αυτή την συνάρτηση. Καλείται κατευθείαν από την `main()` και δεσμεύει ο ίδιος το αποτέλεσμα. Η υλοποίησή του επεξηγείται αργότερα. Σημειώνεται ότι σε περίπτωση μη υποστήριξης του OpenMP, αυτή η συνάρτηση εμφανίζει μήνυμα σφάλματος και επιστρέφει `NULL` .

Για την υλοποίηση του σειριακού αλγορίθμου, έχει γίνει η εξής προσέγγιση:
Έστω ότι πολλαπλασιάζονται τα `pol1` και `pol2` . Τότε, διατηρώντας ένα άθροισμα αρχικοποιημένο σε 0, **για κάθε όρο** του `pol1` πολλαπλασιάζουμε αυτό τον όρο με **ολόκληρο** το `pol2` , πολλαπλασιάζοντας ή μηδενίζοντας τους κατάλληλους συντελεστές και προσαρμόζοντας τις δυνάμεις. Στο τέλος κάθε επανάληψης προσθέτουμε το προσωρινό αποτέλεσμα στο άθροισμα. Έτσι στο τέλος του αλγορίθμου το άθροισμα είναι το αποτέλεσμα του πολλαπλασιασμού.

Για την **παραλληλοποίηση** του σειριακού αλγορίθμου έχει γίνει η εξής προσέγγιση: Παρατηρείται ότι ο σειριακός αλγόριθμος αποτελείται από δύο εμφωλευμένους βρόχους. Η παραλληλοποίηση επιλέχθηκε να γίνει στον εξωτερικό βρόχο επειδή έτσι το μοναδικό κρίσιμο σημείο είναι η πρόσθεση πολυωνύμων και η εγγραφή στην κοινόχρηστη μεταβλητή του αθροίσματος. Αυτό επίσης μπορεί να βελτιωθεί αν οι προσθέσεις των "τοπικών" επαναλήψεων γίνουν σε τοπικά αθροίσματα για κάθε νήμα, και το κρίσιμο σημείο περιλαμβάνει μόνο μία πρόσθεση πολυωνύμων. Ο λόγος που δεν επιλέχθηκε ο εσωτερικός βρόχος για παραλληλοποίηση είναι ότι εκτός της επιμέρους παραλληλοποίησης και της πρόσθεσης, θα απαιτούνταν **φράγματα** πριν και μετά τον αλγόριθμο της πρόσθεσης. Αυτό θα γινόταν γιατί το τμήμα κώδικα που πολλαπλασιάζει έναν όρο με ένα πολυώνυμο έχει διαφορετικό αριθμό επαναλήψεων από αυτές της πρόσθεσης, πράγμα που σημαίνει ότι (για την πλήρη αξιοποίηση των δεδομένων νημάτων) ο διαμοιρασμός τους θα γινόταν με διαφορετικά νήματα να χρησιμοποιούν αποτελέσματα άλλων νημάτων, το οποίο χωρίς φράγματα θα εμφάνιζε συνθήκες ανταγωνισμού.

Στην συνάρτηση `pol_multiply_threaded()` , για να γίνει η πρόσθεση του τοπικού αθροίσματος κάθε νήματος στο διαμοιραζόμενο συνολικό άθροισμα, χρησιμοποιείται το `#pragma omp critical` . Με την χρήση του εξασφαλίζεται ο σωστός συγχρονισμός των νημάτων έτσι ώστε να μην υπάρχουν συνθήκες ανταγωνισμού σε διαμοιραζόμενα δεδομένα. Σε πολλές περιπτώσεις, στον αλγόριθμο τα νήματα γράφουν σε "κοινούς" πίνακες. Όμως τότε δεν χρειάζεται συγχρονισμός γιατί οι πίνακες αυτοί έχουν ξεχωριστές θέσεις για κάθε νήμα και άρα στην πράξη κάθε νήμα ποτέ δεν προσπαθεί να γράψει σε θέση που δεν του ανήκει. Σημειώνεται ότι στον βρόχο που παραλληλοποιείται με OpenMP έχει χρησιμοποιηθεί `#pragma omp for schedule(static, 1) nowait` , όπου το `schedule()` χρησιμοποιείται για την ανάθεση των επαναλήψεων κυκλικά (πιο ομοιόμορφη κατανομή φόρτου), ενώ με την χρήση του `nowait` ακυρώνεται το έμμεσο φράγμα μετά το loop, επιτρέποντας ένα thread να συνεχίσει προσπαθώντας να εκτελέσει το critical section (το οποίο μπορεί να εκτελείται παράλληλα με non-critical sections).

Σημειώνεται ότι στην συνάρτηση `pol_multiply_threaded()` γίνονται στην αρχή αρκετές δεσμεύσεις μνήμης, έτσι ώστε να αποφευχθούν όσο το δυνατότερο δεσμεύσεις μέσα σε βρόχους του αλγορίθμου. Ωστόσο αυτές οι δεσμεύσεις θεωρούνται τμήμα του αλγορίθμου όσον αφορά τις μετρήσεις, επειδή ο σειριακός αλγόριθμος δεν έχει κατι αντίστοιχο, και επομένως αυτό πρέπει να θεωρηθεί overhead του παράλληλου αλγορίθμου. Στην πράξη αυτές οι δεσμεύσεις προκύπτει ότι είναι τάξης μικρότερης του 1 *μs*. Επισης σημειώνεται ότι ο κώδικας για την δέσμευση αυτών των μεταβλητών έχει αλλάξει σε ένα βαθμό σε σχέση με το αντίστοιχο κώδικα της άσκησης 1.1 της εργασίας 1. Αυτή η αλλαγή έχει γίνει για λόγους καλύτερης οργάνωσης του κώδικα και δεν έχει επιρροή στον χρόνο εκτέλεσης, που σημαίνει ότι η σύγκριση της άσκησης 1.1 με την 2.1 μπορεί να γίνει με συνεπή τρόπο για τον χρόνο εκτέλεσης.

Πειραματικά αποτελέσματα

- **N**: Βαθμός των πολυωνύμων
- **Threads**: Αριθμός νημάτων στον παράλληλο αλγόριθμο
- Σε όλα τα παρακάτω αποτελέσματα οι χρόνοι είναι σε *seconds* και δίνεται ακρίβεια *microsecond*.
- Αποφεύγεται η χρήση του N = *10^6* γιατί στο συγκεκριμένο σύστημα μια τέτοια εκτέλεση ολοκληρώνεται σε μη πρακτικό αριθμό λεπτών, καθώς η πολυπλοκότητα είναι τετραγωνική.
- Ο μέγιστος αριθμός από threads που χρησιμοποιείται στα πειράματα είναι **15** και όχι 16. Αυτό συμβαίνει λόγω της naive προσέγγισης που έχει γίνει στην άσκηση 1.1 με pthreads, επειδή το main thread που δημιουργεί τα pthreads δεν εκτελεί καμία "χρήσιμη" εργασία, αλλά απλώς περιμένει τα υπόλοιπα threads. Επίσης αυτό σημαίνει ότι τα πειράματα με 16 threads στην άσκηση 1.1 τεχνικά χρησιμοποιούσαν 17 threads, δηλαδή ένα παραπάνω νήμα που δεν εκτελούνταν πραγματικά παράλληλα με όλα τα υπόλοιπα. Αντίθετα, η άσκηση 2.1 με OpenMP αξιοποιεί όλα τα threads, συμπεριλαμβανομένου του main thread, και μπορεί πράγματι να τρέξει τον αλγόριθμο με 16 hardware threads. Για να γίνει η σύγκριση σωστά όμως, εδώ θα χρησιμοποιηθούν έως 15 threads και για τις δύο υλοποιήσεις.

Ακολουθούν οι μετρήσεις των πειραμάτων για την υλοποίηση με **OpenMP**.

Σειριακοί χρόνοι εκτέλεσης για κάθε πείραμα με συγκεκριμένο N και Threads.

N\Threads	**1**	**2**	**4**	**8**	**15**
1000	0.000873	0.000865	0.000955	0.000871	0.000862
10000	0.092004	0.095119	0.092414	0.089482	0.088982
100000	9.148084	9.210692	9.344938	9.392035	9.282453

Παράλληλοι χρόνοι εκτέλεσης για κάθε πείραμα με συγκεκριμένο N και Threads, σε αντιστοιχία με τον παραπάνω πίνακα.

N\Threads	**1**	**2**	**4**	**8**	**15**
1000	0.000913	0.000642	0.000471	0.006442	0.004524
10000	0.084958	0.045485	0.039771	0.030588	0.042729
100000	8.215920	4.235178	2.362443	1.588373	1.272459

Speedup παράλληλης εκτέλεσης σε σχέση με την σειριακή, σε αντιστοιχία με τους παραπάνω πίνακες.

N\Threads	**1**	**2**	**4**	**8**	**15**
1000	0.955665	1.348402	2.027056	0.135245	0.190595
10000	1.082935	2.091205	2.323688	2.925387	2.082486
100000	1.113458	2.174806	3.955625	5.912993	7.294895

Efficiency σε σχέση με το *speed up* και τον αριθμό των νημάτων, σε αντιστοιχία με τους παραπάνω πινακες.

N\Threads	**1**	**2**	**4**	**8**	**15**
1000	0.955665	0.674201	0.506764	0.016905	0.012706
10000	1.082935	1.045602	0.580922	0.365673	0.138832
100000	1.113458	1.087403	0.988906	0.739124	0.486326

Σχολιασμός αποτελεσμάτων

Από τους παραπάνω πίνακες συμπεραίνουμε τα εξής:

- Γενικά υπάρχει αισθητή επιτάχυνση, η οποία είναι πολύ σημαντικότερη για μεγαλύτερες τιμές του *N*. Για *N*=1000 για παράδειγμα, η επιτάχυνση είναι χαμηλότερη από 1.5 κατά μέσο όρο, ενώ για *N*=10000 είναι ήδη κοντά στο 3 κατά μέσο όρο.
- Μέχρι και τα 4-8 νήματα η επιτάχυνση είναι αρκετά αποδοτική, ενώ στα 16 παρατηρούμε ότι η αποδοτικότητα είναι ελάχιστη. Αυτό προφανώς συμβαίνει επειδή το μέγεθος των δεδομένων δεν είναι αρκετά μεγάλο για την πραγματική αξιοποίηση 15 νημάτων. Πιθανότατα, για αρκετά μεγαλύτερα *N* (τα οποία δεν παρατίθονται για λόγους ταχύτητας), η αποδοτικότητα των 15 νημάτων θα αυξανόταν.
- Για 1 και 2 νήματα παρατηρούμε μερικές αποδοτικότητες λίγο μεγαλύτερες του 1, το οποίο στην θεωρία δεν θα έπρεπε να συμβαίνει. Εφόσον οι αλγόριθμοι έχουν ο καθένας διαφορετικά overheads, αλλά και διαφορετική χρήση της μνήμης (ίσως και της cache), είναι ίσως λογικό (για σχετικα μεγάλα *N*) να υπάρχει λίγος θόρυβος, που έχει αυτό το αποτέλεσμα.

Σύγκριση με τα αποτελέσματα της άσκησης 1.1

Ακολουθούν οι μετρήσεις των πειραμάτων για την υλοποίηση με **pthreads**.

Speedup παράλληλης εκτέλεσης σε σχέση με την σειριακή.

N\Threads	1	2	4	8	15
1000	0.791199	1.239546	1.288399	1.229979	0.969816
10000	1.103307	2.099371	3.287014	3.734003	4.312176
100000	1.118343	2.216025	3.914853	6.188193	7.613948

Efficiency σε σχέση με το *speed up* και τον αριθμό των νημάτων.

N\Threads	1	2	4	8	15
1000	0.791199	0.619773	0.322100	0.153747	0.064654
10000	1.103307	1.049686	0.821754	0.466750	0.287478
100000	1.118343	1.108013	0.978713	0.773524	0.507597

Παρατηρούμε ότι μέχρι και τα 4 threads οι δύο υλοποιήσεις έχουν περίπου τα ίδια speedups, και δεν φαίνεται καμία από τις δύο να είναι απολύτως αποδοτικότερη από την άλλη. Ωστόσο για 8-15 threads η υλοποίηση με **pthreads** είναι φανερά **αποδοτικότερη**. Αυτό είναι αναμενόμενο γιατί η υλοποίηση με OpenMP είναι περισσότερο high level και έχει αναπόφευκτα overheads που προκαλούν αισθητές καθυστερήσεις σε σχεση με τα low level pthreads.

Για τα πειράματα χρησιμοποιήθηκε σύστημα με τα εξής χαρακτηριστικά:

- **Μοντέλο laptop:** MSI Katana GF66 12UC
- **Μοντέλο CPU:** 12th Gen Intel(R) Core(TM) i7-12650H
- **Logical processors:** 16
- **OS:** WSL2 (Ubuntu) πάνω σε Windows11
- **C compiler:** gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0

Σημείωση: Η hardware μνήμη RAM είναι 16GB όμως το WSL την περιορίζει σε ~8GB. Παρόλα αυτά **τονίζεται ότι το WSL δεν περιορίζει την χρήση των hardware threads**.

Το πρόγραμμα της άσκησης εκτελείται με σωστό τρόπο στα συστήματα linux του εργαστηρίου.

Άσκηση 2.2

Χαρακτηριστικά υλοποίησης

- Υλοποίηση προγράμματος από: Κωνσταντίνο Γεώργιο Βλαζάκη (sdi2300017)
- Λειτουργικό σύστημα όπου εκτελέστηκαν τα προγράμματα: Linux Mint 22.2 Cinnamon
- CPU: AMD Ryzen 5 7600 6-Core Processor
- GCC version: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0

Παραδοχές:

- Όλες οι ακέραιες τιμές σε πίνακες και διανύσματα βρίσκονται στο διάστημα [0, 100], ώστε να είναι ευανάγνωστα τα δεδομένα.
- Το πρόγραμμα δεν μπορεί να λάβει τιμή διάστασης του πίνακα μεγαλύτερη από 10^4, καθώς καταλήγει σε υπερχείλιση (overflow) ακεραίων.

Υλοποιήσεις

main: Κύριο πρόγραμμα προς εκτέλεση:

- Ξεκινάει θέτοντας τα αναγκαία δεδομένα για την εκτέλεση του προγράμματος. Αυτά αποτελούν τα ορίσματα από τη γραμμή εντολών (Διάσταση πίνακα, ποσοστό μηδενικών στοιχείων πίνακα, επαναλήψεις πολλαπλασιασμού πίνακα και αριθμό νημάτων προς εκτέλεση), καθώς και τα παράγωγά τους (συνολικό αριθμό τιμών που θα περιέχει ο πίνακας, αριθμό μηδενικών στοιχείων στον πίνακα). Επιπλέον, υπολογίζει τον αριθμό μη μηδενικών στοιχείων που θα υπάρχουν στον πίνακα, για τη δημιουργία και χρήση της Compressed Sparse Row (CSR) αναπαράστασης αραιού πίνακα.
- Δημιουργεί έναν δυσδιάστατο πίνακα χρησιμοποιώντας δυναμική δέσμευση μνήμης (με αριθμό γραμμών "dimension" και αριθμό στηλών "dimension", καθώς είναι τετραγωνικός) και θέτει όλα τα στοιχεία του σε τυχαίους ακέραιους αριθμούς. Αφότου γεμίσει τον πίνακα με ακέραιους, μηδενίζει τόσους από αυτούς όσους έχει ορίσει ο χρήστης, σε τυχαίες θέσεις του πίνακα. Έτσι παράγεται ένας αραιός πίνακας.
- Δημιουργεί ένα διάνυσμα χρησιμοποιώντας δυναμική δέσμευση μνήμης (με αριθμό γραμμών "dimension"), και θέτει όλα τα στοιχεία του σε τυχαίους ακέραιους αριθμούς.
- Πολλαπλασιάζει σειριακά τον πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης (με κάθε επανάληψη να λαμβάνει ως διάνυσμα εισόδου το διάνυσμα εξόδου της προηγούμενης), και μετράει τον χρόνο εκτέλεσης όλων των επαναλήψεων.
- Παράγει σειριακά την CSR αναπαράσταση του πίνακα (μέσω της σειριακής συνάρτησης CSR_create) και μετράει τον χρόνο εκτέλεσής της.
- Πολλαπλασιάζει σειριακά την CSR αναπαράσταση του πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης (με κάθε επανάληψη να λαμβάνει ως διάνυσμα εισόδου το διάνυσμα εξόδου της προηγούμενης), και μετράει τον χρόνο εκτέλεσης όλων των επαναλήψεων.
- Παράγει παράλληλα την CSR αναπαράσταση του πίνακα (μέσω της παράλληλης συνάρτησης CSR_create_omp) και μετράει τον χρόνο εκτέλεσής της.
- Πολλαπλασιάζει παράλληλα τον πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης και μετράει τον χρόνο εκτέλεσης.
- Πολλαπλασιάζει παράλληλα την CSR αναπαράσταση του πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης και μετράει τον χρόνο εκτέλεσης.
- Τέλος, συλλέγει όλα τα δεδομένα που είναι σημαντικά για τον χρήστη (τις παραμέτρους του προγράμματος και τους χρόνους εκτέλεσης της κάθε διαδικασίας) σε αντίστοιχες λίστες, και τα αποθηκεύει σε ένα εξωτερικό αρχείο με όνομα "test_data.txt".

matrixlib: Βιβλιοθήκη συναρτήσεων που σχετίζονται με πίνακες και λίστες:

- Δομή **CSR_t**: Χρησιμοποιείται για την αποθήκευση της μορφής CSR ενός πίνακα. Αποτελείται από μία λίστα ακεραίων για τις μη μηδενικές τιμές του αρχικού πίνακα (**val_array**), μία λίστα ακεραίων για τις στήλες όπου βρίσκονται αυτές οι τιμές (**col_array**), και μία λίστα "δεικτών" στις θέσεις της "**col_array**" όπου βρίσκονται τα πρώτα μη μηδενικά στοιχεία κάθε γραμμής του αρχικού (αραιού) πίνακα (**start_idx**).
- **CSR_create**: Δημιουργεί την CSR αναπαράσταση ενός δυσδιάστατου πίνακα ακεραίων. Λαμβάνει έναν δυσδιάστατο πίνακα ακεραίων, τους αριθμούς στηλών και γραμμών του πίνακα, καθώς και τον αριθμό μη μηδενικών στοιχείων. Αρχικοποιεί τις λίστες μίας δομής CSR_t, και προσθέτει τον αριθμό των μη μηδενικών στοιχείων στην τελευταία θέση της λίστας "**start_idx**". Διατρέχει σειριακά κάθε γραμμή του αρχικού πίνακα, όπου αναθέτει τιμή στο αντίστοιχο στοιχείο της λίστας "**start_idx**" και διατρέχει κάθε στήλη του αρχικού πίνακα ώστε να προσθέσει τις μη μηδενικές τιμές του στη λίστα "**val_array**" και τις αντίστοιχες στήλες τους στην "**col_array**". Τέλος, επιστρέφει τη συμπληρωμένη δομή CSR_t.
- **mat_vec**: Πολλαπλασιάζει έναν δυσδιάστατο πίνακα ακεραίων με ένα διάνυσμα. Λαμβάνει τον αρχικό πίνακα ακεραίων, το διάνυσμα με το οποίο αυτός θα πολλαπλασιαστεί, και τους αριθμούς γραμμών και στηλών του πίνακα. Αρχικοποιεί το διάνυσμα αποτελέσματος. Διατρέχει σειριακά κάθε γραμμή και στήλη του πίνακα και αθροίζει τα γινόμενα των στοιχείων κάθε γραμμής με τα στοιχεία κάθε στήλης του διανύσματος. Τέλος, επιστρέφει το συμπληρωμένο διάνυσμα αποτελέσματος.
- **CSR_mat_vec**: Εφαρμόζει τον πολλαπλασιασμό δυσδιάστατου πίνακα ακεραίων με διάνυσμα, χρησιμοποιώντας την CSR αναπαράσταση του πίνακα. Λαμβάνει τη δομή CSR_t που περιέχει την CSR αναπαράσταση, το διάνυσμα με το οποίο θα πολλαπλασιαστεί και τη διάσταση που αντιστοιχεί στον πίνακα και το διάνυσμα. Αρχικοποιεί το διάνυσμα αποτελέσματος. Διατρέχει σειριακά τις γραμμές, βρίσκει το πρώτο μη μηδενικό στοιχείο κάθε γραμμής του αρχικού πίνακα, καθώς και της ακριβώς επόμενης γραμμής. Διατρέχει όλα τα μη μηδενικά στοιχεία ανάμεσα σε αυτά που έχει βρει και αθροίζει τα γινόμενα τους με τα στοιχεία του διανύσματος. Τέλος, επιστρέφει το συμπληρωμένο διάνυσμα αποτελέσματος.
- **print_matrix**: Εκτυπώνει έναν δυσδιάστατο πίνακα ακεραίων στην κονσόλα. Λαμβάνει τον δυσδιάστατο πίνακα και τους αριθμούς γραμμών και στηλών του.
- **print_CSR**: Εκτυπώνει μία αναπαράσταση CSR στην κονσόλα. Λαμβάνει τη δομή CSR_t και τον αριθμό γραμμών του αρχικού πίνακα.
- **CSR_create_omp**: Εκτελεί την ίδια λειτουργία με την CSR_create. Εδώ η υλοποίηση έχει αλλαχτεί ώστε να επιτραπεί η παραλληλοποίησή της. Χρησιμοποιεί τρεις ξεχωριστούς βρόχους for, ώστε να αφαιρεθεί η εξάρτηση που υπήρχε (κατά την εκτέλεση της CSR_create) πάνω στη μεταβλητή "**list_idx**". Ο πρώτος εκτελεί μία απλή μέτρηση των μη μηδενικών στοιχείων κάθε γραμμής του αρχικού πίνακα, όπου κάθε αριθμός αποθηκεύεται στην επόμενη θέση από αυτή που αντιστοιχεί στη γραμμή του. Ο δεύτερος αθροίζει κάθε αριθμό (μετά από την πρώτη θέση, καθώς πρέπει να παραμείνει "0") με τον ακριβώς προηγούμενό του. Πλέον, η λίστα περιέχει "δείκτες" στις ακριβείς θέσεις όπου υπάρχουν τα πρώτα μη μηδενικά στοιχεία κάθε γραμμής του αρχικού πίνακα. Ο τρίτος βρόχος διατρέχει τις λίστες "**val_array**" και "**col_array**", χρησιμοποιώντας τη συμπληρωμένη λίστα "**start_idx**", και αποθηκεύει όλα τα μη μηδενικά στοιχεία και τις αντίστοιχες στήλες όπου βρίσκονται.

- **mat_vec_omp**: Εκτελεί την ίδια λειτουργία με την "**mat_vec**", αλλά ο βρόχος που διατρέχει τις γραμμές του αρχικού πίνακα έχει παραλληλοποιηθεί μέσω της εντολής "**pragma omp parallel for**", με αριθμό νημάτων ορισμένο στο "**main.c**" από παράμετρο γραμμής εντολών. Οι μεταβλητές "**i**" και "**j**" του βρόχου έχουν τεθεί ως ιδιωτικές (private) για κάθε νήμα, καθώς αποτελούν εξάρτηση για την πρόσβαση στον πίνακα και τα διανύσματα στις πράξεις.
- **CSR_mat_vec_omp**: Εκτελεί την ίδια λειτουργία με την "**CSR_mat_vec**", αλλά ο βρόχος που διατρέχει τις γραμμές έχει παραλληλοποιηθεί μέσω της εντολής "**pragma omp parallel for**". Οι μεταβλητές "**i**", "**j**", "**row_start**" και "**row_end**" του βρόχου έχουν τεθεί ως ιδιωτικές για κάθε νήμα.
- **compare_array**: Συγκρίνει όλα τα στοιχεία δύο λιστών ανά θέση και επιστρέφει τον αριθμό των στοιχείων που δεν ταιριάζουν.

Αποτελέσματα εκτελέσεων

- Προϋποθέσεις:
  - Ο χρόνος κάθε διαδικασίας μετριέται σε δευτερόλεπτα.
  - Κάθε εκτέλεση έχει επαναληφθεί 5 φορές ώστε να μετρηθεί ο μέσος όρος του χρόνου κάθε διαδικασίας.
  - Τα δεδομένα αυτά λήφθηκαν μέσω του bash script "**test_script.sh**" και της εντολής "**make test**".
  - Κάθε γραμμή περιγράφει τους χρόνους εκτέλεσης μίας διαδικασίας, κάθε στήλη την παράμετρο που έχει αλλάξει κατά τη συγκεκριμένη εκτέλεση.
  - Οι παράμετροι που αλλάζουν είναι οι εξής:
    - D -> Διάσταση πίνακα (αποτελεί αριθμό γραμμών και στηλών για τετραγωνικό πίνακα).
    - Z -> Ποσοστό μηδενικών στοιχείων στον πίνακα.
    - R -> Επαναλήψεις πολλαπλασιασμού πίνακα-διανύσματος.
    - T -> Αριθμός νημάτων.
  - Οι προεπιλεγμένες (Default) παράμετροι είναι:
    - D = 1000.
    - Z = 60.
    - R = 5.
    - T = 4.
- Σειριακή εκτέλεση συναρτήσεων/διαδικασιών:

Function/Parameters:	**Default**	**D=10000**	**Z=0**	**Z=99**	**R=1**	**R=20**
Serial Product	0.038701	3.211010	0.032476	0.031889	0.006420	0.128637
Serial CSR Creation	0.022590	1.620603	0.012938	0.003824	0.018679	0.019010
Serial CSR Product	0.010396	1.772760	0.021668	0.000286	0.001722	0.034366

- Σχόλια:
  - Η αλλαγή της διάστασης πίνακα από 1000 σε 10000 παράγει μεγάλη διαφορά στον υπολογισμένο χρόνο εκτέλεσης, σε σύγκριση με κάθε άλλη αλλαγή.
  - Υπάρχει σημαντική επιτάχυνση στον πολλαπλασιασμό πίνακα-διανύσματος, όταν συγκρίνουμε την κοινή σειριακή υλοποίηση με την CSR σειριακή υλοποίηση.
    - Για παράδειγμα, με τις προεπιλεγμένες παραμέτρους, το σειριακό γινόμενο CSR-διανύσματος έχει χρόνο εκτέλεσης 3.7~ γρηγορότερο από το κοινό σειριακό γινόμενο (αραιού πίνακα με διάνυσμα).
    - Η επιτάχυνση αυτή αυξάνεται όσο μεγαλώνει το ποσοστό μηδενικών στον αραιό πίνακα, εφόσον οι λίστες της CSR αναπαράστασης καταλήγουν μικρότερες.
- Παράλληλη εκτέλεση συναρτήσεων/διαδικασιών:

Function/Parameters:	**Default**	**D=10000**	**Z=0**	**Z=99**	**R=1**	**R=20**	**T=8**	**T=16**
Parallel Product	0.003955	0.342185	0.003574	0.003639	0.000677	0.014089	0.003390	0.002982
Parallel CSR Creation	0.003148	0.179188	0.001796	0.000764	0.002996	0.003135	0.001943	0.002004
Parallel CSR Product	0.005874	1.002648	0.011930	0.000201	0.000994	0.021651	0.004507	0.004648

- Επιταχύνσεις από σειριακή σε παράλληλη εκτέλεση συναρτήσεων/διαδικασιών:

Function/Parameters:	**Default**	**D=10000**	**Z=0**	**Z=99**	**R=1**	**R=20**	**T=8**	**T=16**
Speedup of Product	9.785335	9.383842	9.086738	8.763122	9.483013	9.130314	11.416224	12.978203
Speedup of CSR Creation	7.175985	9.044149	7.203786	5.005236	6.234646	6.063796	11.626351	11.272455
Speedup of CSR Product	1.769833	1.768078	1.816261	1.422886	1.732394	1.587271	2.306634	2.236661

- Σχόλια:
  - Η αλλαγή της διάστασης πίνακα από 1000 σε 10000 εξακολουθεί να παράγει μεγάλη διαφορά στον υπολογισμένο χρόνο εκτέλεσης, σε σύγκριση με κάθε άλλη αλλαγή.
  - Στην παράλληλη εκτέλεση, ο πολλαπλασιασμός CSR-διανύσματος φέρνει χειρότερα αποτελέσματα από την κοινή υλοποίηση. Η κοινή υλοποίηση λαμβάνει υψηλή επιτάχυνση μόλις εκτελεστεί με πολλαπλά νήματα, καθώς οι υπολογισμοί είναι απλοί, και η πρόσβαση στη μνήμη cache του επεξεργαστή από κάθε thread είναι αποτελεσματική. Αντιθέτως, στην CSR υλοποίηση κάθε νήμα δεν θα έχει ισότιμο αριθμό μη μηδενικών τιμών να λάβει υπόψη του, άρα κάποια νήματα θα καθυστερήσουν τη συνολική παράλληλη εκτέλεση λόγω υψηλότερου φόρτου.

Άσκηση 2.3

Υλοποίηση: Εμμανουήλ-Ταξιάρχης Οζίνης (sdi2300147)

Περιγραφή προβλήματος

Το πρόβλημα που δίνεται περιλαμβάνει κυρίως τα εξής υποπροβλήματα:

- Υλοποίηση σειριακού αλγορίθμου mergesort (συνηθισμένη αναδρομική υλοποίηση).
- Υλοποίηση παράλληλου αλγορίθμου mergesort με βάση την αναδρομική λογική, με την βοήθεια των tasks. Πρέπει να υπάρχει κατάλληλος συγχρονισμός, και τα tasks δεν πρέπει να δημιουργούνται ανεξέλεγκτα στην ταξινόμηση μεγάλων arrays για να μην επιδρούν αρνητικά στην εκτέλεση.

Περιγραφή λύσης

Για την οργάνωση της λύσης, αρχικά ορίζονται οι βοηθητικές συναρτήσεις:

- `parse_args()` : Διαβάζει τα command line arguments με κατάλληλους ελέγχους. Διευκρινίζεται ότι η χρήση είναι η εξής:

  > $ ./sort <array length> {serial|parallel <thread count>}

  Δηλαδή το thread count δίνεται μόνο όταν επιλεχθεί η επιλογή `parallel` .
- `elapsed()` : Επιστρέφει την χρονική διαφορά σε δευτερόλεπτα (έως ακρίβεια nsec) από το start στο end. Σημειώνεται ότι για τις χρονικές μετρήσεις χρησιμοποιείται η δομή struct timespec της *C11*.
- `create_random_arr()` : Δημιουργεί ένα array με τυχαίες μη αρνητικές ακέραιες τιμές και μέγιστη δυνατή τιμή `MAX_ARRAY_VALUE` .
- `is_sorted()` : Ελέγχει αν ένα `int` array είναι ταξινομημένο κατά αύξουσα σειρά.

Τόσο για την σειριακή όσο και για την παράλληλη υλοποίηση της mergesort, χρησιμοποιείται η ίδια συνάρτηση `merge()` , η οποία συγχωνεύει τα (ήδη ταξινομημένα) τμήματα `[lower, mid]` και `[mid + 1, higher]` . Προφανώς η χρήση της σειριακής `merge()` στην παράλληλη υλοποίηση δεν είναι ιδανική, και αυτό το ζήτημα αναλύεται αργότερα.

Η σειριακή υλοποίηση γίνεται στην συνάρτηση `mergesort()` , η οποία καλεί αναδρομικά δύο φορές τον εαυτό της για κάθε μισό του array που ορίζεται από τα `lower` και `higher` indexes (το `higher` περιλαμβάνεται). Μετά τις αναδρομικές κλήσεις γίνεται `merge()` μεταξύ αυτών των δύο μισών για να υπάρξει ταξινόμηση στο `[lower, higher]` . Η ρίζα της αναδρομής έχει `lower = 0` και `higher = n - 1` , όπου `n` το length του array.

Η παράλληλη υλοποίηση γίνεται από την `mergesort_threaded()` και την `mergesort_threaded_internal()` , η οποία καλείται μέσα στην πρώτη.

Η `mergesort_threaded()` αναλαμβάνει την δημιουργία (και καταστροφή) των νημάτων (με `#pragma omp parallel num_threads(thread_count) ...`) και αναθέτει την top level κλήση της `mergesort_threaded_internal()` σε ένα μόνο νήμα (με `#pragma omp single`).

Η `mergesort_threaded_internal()` υποθέτει ότι καλείται από μόνο ένα νήμα. Για να υπάρξει παραλληλοποίηση, οι αναδρομικές κλήσεις `mergesort_threaded_internal()` δεν γίνονται απευθείας από το αρχικό νήμα, αλλά ορίζονται ως tasks για οποιοδήποτε διαθέσιμο νήμα με την χρήση του `#pragma omp task ...` . Για να γίνει σωστά το `merge()` από το αρχικό νήμα, αυτό πρέπει να περιμένει να ολοκληρωθούν τα tasks που ανέλαβαν τις αναδρομικές κλήσεις. Αυτό γίνεται με `#pragma omp taskwait` πριν από την κλήση της `merge()` .

Με τον παραπάνω παράλληλο αλγόριθμο, κάθε νήμα που αναλαμβάνει ένα task,αναγκάζεται να δημιουργήσει και το ίδιο νέα tasks. Αν η δημιουργία νέων tasks γίνει ανεξέλεγκτα τότε γρήγορα θα κυριαρχήσει η συνιστώσα των overheads για κάθε task, καθώς κάθε thread θα αναγκάζεται να δημιουργεί και να αναλαμβάνει ένα υπερβολικά μεγάλο πλήθος από tasks, με αποτέλεσμα να επιβαρύνεται ο χρόνος εκτέλεσης. Για να λυθεί αυτό, τα tasks ορίζονται μόνο μέχρι συγκεκριμένο threshold στο βάθος της αναδρομής, ενώ μετά το threshold οι πιο βαθιές αναδρομικές κλήσεις αναλαμβάνονται απευθείας από τα αντίστοιχα αρχικά threads, χωρίς δημιουργία νέων tasks. Έτσι η παραλληλία γίνεται περισσότερο coarse-grained και τα tasks παραμένουν σε πλήθος εύκολα διαχειρίσιμο από τα hardware threads. Για να γίνουν όλα τα παραπάνω χρησιμοποιείται το `if()` clause στο κάθε `#pragma omp task` , ενώ η `mergesort_threaded_internal()` λαμβάνει παράμετρο `depth` που αυξάνεται σε κάθε αναδρομική κλήση. Στο σύστημα που έγιναν οι μετρήσεις, η δημιουργία νέων tasks

σταματά στο `depth = 4`. Σημειώνεται ότι στους ορισμούς των tasks χρησιμοποιείται και το `firstprivate()` clause για τον συνεπή ορισμό των τιμών που θα έχουν οι μεταβλητές που χρησιμοποιούνται στην εκτέλεση των tasks.

Στην παράλληλη υλοποίηση χρησιμοποιήθηκε η ίδια σειριακή `merge()` με την σειριακή υλοποίηση. Αυτή η επιλογή έγινε για τους εξής λόγους:

- Η παραλληλοποίηση της "in-place" λογικής της `merge()`, ιδιαίτερα για παραμετρικό αριθμό νημάτων, θα ήταν αρκετά σύνθετη λόγω της σειριακής ανατοποθέτησης των ταξινομημένων στοιχείων. Δηλαδή κάθε νήμα δεν θα γνώριζε αν η επόμενη κενή θέση πρέπει να δωθεί σε δικό του στοιχείο ή σε στοιχείο άλλου νήματος, εκτός και αν γινόταν κάποια προεπεξεργασία.
- Αν γινόταν παραλληλοποίηση την `merge()` τότε θα προέκυπτε ένα ακόμα ζήτημα: Στην τρέχουσα υλοποίηση μόνο ένα thread μπορεί να μπει μέσα σε οποιαδήποτε κλήση της `mergesort_threaded_internal()`. Επομένως για την υποστήριξη μιας πολυνηματικής `merge()` θα έπρεπε πολλά threads να μπαίνουν στην ίδια `mergesort_threaded_internal()`, αλλά όχι όλα, γιατί κάποια θα ήταν αποδοτικό να είναι συνεχώς stand-by για να αναλάβουν νέα tasks. Δηλάδη μια τέτοιου είδους σχεδίαση θα απαιτούσε τον ορισμό δύο νοητών ομάδων threads, όπου η μία εστιάζει στην εκτέλεση και δημιουργία tasks και η άλλη εστιάζει στην παράλληλη εκτέλεση της `merge()`.

Επειδή δεν έγινε παραλληλοποίηση στην `merge()`, όπως είναι αναμενόμενο, η παράλληλη εκτέλεση του αλγορίθμου αν και έχει επιτάχυνση, αυτή η επιτάχυνση είναι σημαντικά μικρότερη από την ιδανική, ειδικά για μεγάλο πλήθος threads. Αυτό θα φανεί και στα πειραματικά αποτελέσματα.

Πειραματικά αποτελέσματα

- **N**: Βαθμός των πολυωνύμων
- **Threads**: Αριθμός νημάτων στον παράλληλο αλγόριθμο
- Σε όλα τα παρακάτω αποτελέσματα οι χρόνοι είναι σε *seconds* και δίνεται ακρίβεια *microsecond*.
- Αποφεύγεται η χρήση του N = *10^8* γιατί στο συγκεκριμένο σύστημα μια τέτοια εκτέλεση ολοκληρώνεται σε μη πρακτικό αριθμό λεπτών.

Σειριακοί χρόνοι εκτέλεσης για κάθε πείραμα με συγκεκριμένο N. Για κάθε N ο σειριακός χρόνος μετράται στην αρχή, και όχι κάθε φορά για διαφορετικό αριθμό νημάτων.

N	**t**
100000	0.007267
1000000	0.070821
10000000	0.729270

Παράλληλοι χρόνοι εκτέλεσης για κάθε πείραμα με συγκεκριμένο N και Threads.

N\Threads	**1**	**2**	**4**	**8**	**16**	
100000	0.008896	0.004802	0.004589	0.004242	0.008070	
1000000	0.088091	0.046557	0.038014	0.030236	0.024493	
10000000	0.907519	0.476589	0.374667	0.278249	0.205092	

Speedup παράλληλης εκτέλεσης σε σχέση με την σειριακή, σε αντιστοιχία με τους παραπάνω πίνακες.

N\Threads	**1**	**2**	**4**	**8**	**16**
100000	0.816912	1.513380	1.583538	1.712964	0.900471
1000000	0.803958	1.521181	1.863043	2.342244	2.891460
10000000	0.803586	1.530187	1.946446	2.620924	3.555821

Efficiency σε σχέση με το *speed up* και τον αριθμό των νημάτων, σε αντιστοιχία με τους παραπάνω πινακες.

N\Threads	**1**	**2**	**4**	**8**	**16**
100000	0.816912	0.756690	0.395884	0.214120	0.056279
1000000	0.803958	0.760590	0.465761	0.292780	0.180716
10000000	0.803586	0.765093	0.486611	0.327616	0.222239

Σχολιασμός αποτελεσμάτων

Από τους παραπάνω πίνακες συμπεραίνουμε τα εξής:

- Υπάρχει επιτάχυνση, αλλά γρήγορα περιορίζεται και παραμένει μικρότερη από 4, ακόμα και για 16 threads.
- Επομένως οι αποδοτικότητες, αν και ξεκινούν ικανοποιητικές (~0.8), γρήγορα μειώνονται και παραμένουν πολύ χαμηλές ακόμα και για μεγάλα N. Για παράδειγμα στα 4 threads οι αποδοτικότητες είναι ήδη χαμηλότερες από 0.5.
- Οι χαμηλές αποδοτικότητες οφείλονται κυρίως στην χρήση σειριακής `merge()` στην παράλληλη υλοποίηση. Λόγω του νόμου του Amdahl, είναι εμφανές ότι αυτό το σειριακό τμήμα εκτέλεσης θέτει όρια στην παραλληλοποίηση συνολικά. Βέβαια στην πράξη οι κλήσεις στην `merge()` δεν γίνονται αυστηρά σειριακά, αλλά υπάρχουν ορισμένες επικαλύψεις λόγω της χρήσης των tasks. Προφανώς όμως αυτές οι επικαλύψεις δεν είναι αρκετά σημαντικές, όπως φαίνεται από τα αποτελέσματα.
- Ένας ακόμα λόγος για τις χαμηλές αποδοτικότητες είναι η χρήση των `taskwait`, που λειτουργούν σαν μία μορφή barriers και αναγκάζουν κάθε εκτέλεση μιας `merge()` να γίνεται αυστηρά **μετά** από τις κλήσεις `merge()` του επόμενου βάθους. Αυτό σημαίνει ότι ακόμα και αν όλες οι κλήσεις `merge()` γίνονταν παράλληλα με όσες έχουν το **ίδιο βάθος**, οι κλήσεις `merge()` με διαφορετικά βάθη δεν μπορούν να κληθούν ποτέ ταυτόχρονα, αλλά πρέπει να εκτελεστεί κάθε βάθος μετά από το άλλο. Αυτός ο περιορισμός είναι ακόμα πιο δύσκολο να αντιμετωπιστεί γιατί αυτή είναι η φύση του αλγορίθμου mergesort.

Για τα πειράματα χρησιμοποιήθηκε σύστημα με τα εξής χαρακτηριστικά:

- **Μοντέλο laptop:** MSI Katana GF66 12UC
- **Μοντέλο CPU:** 12th Gen Intel(R) Core(TM) i7-12650H
- **Logical processors:** 16
- **OS:** WSL2 (Ubuntu) πάνω σε Windows11
- **C compiler:** gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0

Σημείωση: Η hardware μνήμη RAM είναι 16GB όμως το WSL την περιορίζει σε ~8GB. Παρόλα αυτά **τονίζεται ότι το WSL δεν περιορίζει την χρήση των hardware threads**.

Το πρόγραμμα της άσκησης εκτελείται με σωστό τρόπο στα συστήματα linux του εργαστηρίου.