Μ127 - Παράλληλα Υπολογιστικά Συστήματα

Εργασία 1 – Προγραμματισμός με Pthreads & OpenMP

Ονοματεπώνυμο: Ρέσκος Νικόλαος *ΑΜ*: 7115112300030

►Υπολογιστικό σύστημα

Για την υλοποίηση της συγκεκριμένης εργασίας εκμεταλευτήκαμε το δίκτυο Υπολογιστών του Τμήματος και συγκεκριμένα τον υπολογιστή linux05.di.uoa.gr του δικτύου. Ο συγκεκριμένος υπολογιστής διαθέτει τετραπύρηνο επεξεργαστή Intel Core i5-6500 χρονισμένο στα 3.2 GHz, ενώ η συνολική L1 cache του είναι 256 kB, η L2 1MB και η L3 6MB. Η έκδοση του λειτουργικού συστήματος είναι η Ubuntu *20.04.6 LTS (Focal Fossa)*, ενώ η έκδοση του μεταγλωτιστή gcc είναι η 9.4.0.

Στο παραδοτέο συμπιεσμένο αρχείο έχουμε κάθε άσκηση σε ξεχωριστό φάκελο, όπου περιέχονται οι αντίστοιχοι πηγαίοι κώδικες, τα αρχεία Makefile, τα scripts που δημιουργήσαμε για την αυτοματοποίηση των πειραμάτων καθώς και τα python scripts που δημιουργήσαμε προκειμένου να διαχειριστούμε πιο εύκολα τα αποτελέσματα των πειραμάτων και να κατασκευάσουμε τους πίνακες και τα γραφήματα. Σε κάθε άσκηση πραγματοποιήσαμε πέντε φορές κάθε πείραμα, δηλαδή εκτελέσαμε το πρόγραμμα πέντε φορές για τον ίδιο αριθμό νημάτων και παραμέτρων εισόδου, ώστε να παρουσιάσουμε τους μέσους όρους των αποτελεσμάτων αυτών.

● Εργασία 1.1 (Pthreads & OpenMP)

Σε αυτήν την άσκηση επιδιώκουμε να εκτιμήσουμε την τιμή του π μέσω της μεθόδου “Monte Carlo”, με τρεις διαφορετικούς αλγορίθμους: i) σειριακό, ii) με χρήση Pthreads και iii) με χρήση OpenMP. Το πρόγραμμα που κατασκευάζουμε δέχεται ως είσοδο το πλήθος των τυχαίων μεταβλητών (πλήθος ρήψεων) και τον αριθμό των νημάτων και επιστρέφει το χρόνο εκτέλεσης του εκάστοτε αλγορίθμου.

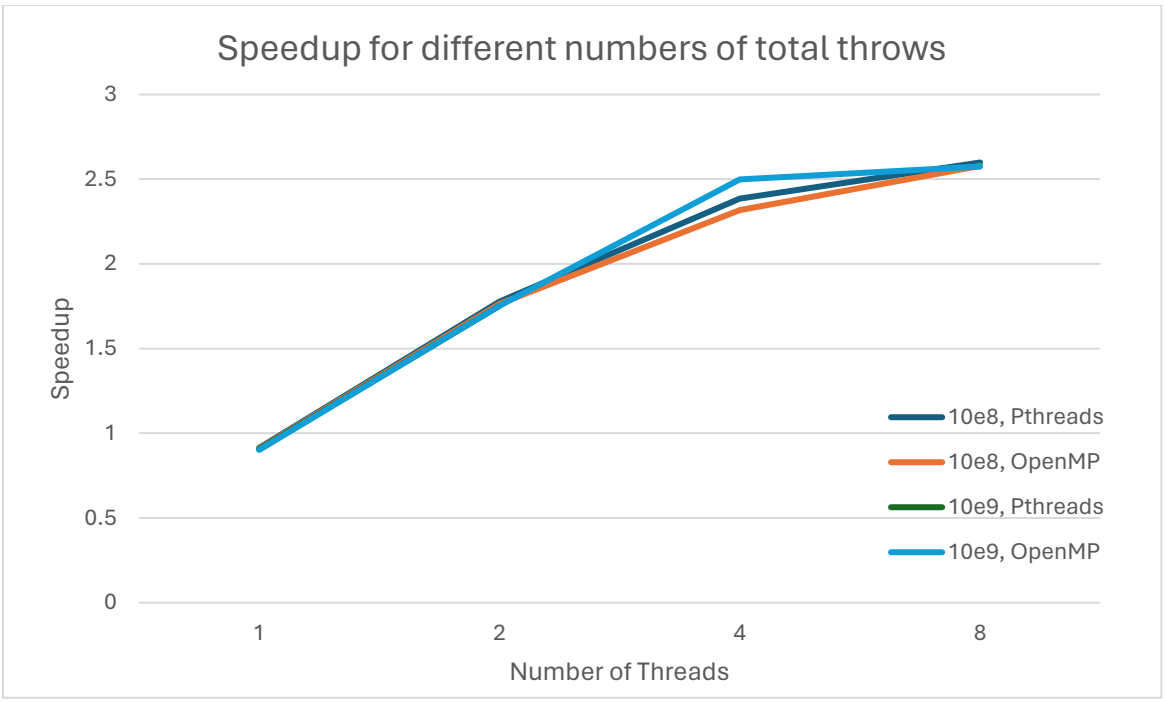
Η βασική ιδέα είναι ότι δημιουργούμε δύο τυχαίους αριθμούς στο διάστημα [-1,1], οι οποίοι αναπαριστούν τις συντεταγμένες ενός σημείου, και στη συνέχεια για κάθε τέτοιο σημείο, το οποίο αναπαριστά μία ρήψη, υπολογίζουμε αν η ευκλείδια απόσταση αυτού του σημείου είναι εντός του μοναδιαίου κύκλου. Με αυτόν τον τρόπο μπορούμε να εκτιμήσουμε το π, καθώς με την υπόθεση ότι τα σημεία μας είναι τυχαία και ακολουθούν μια ομοιόμορφη κατανομή, το ποσοστό αυτών που θα είναι εντός του κύκλου θα προσσεγγίζει τον λόγο των εμβαδών του μοναδιαίου κύκλου με το τετράγωνο στο οποίο είναι εγγεγραμμένος.

Ο σειριακός αλγόριθμος υλοποιείται μέσω της συνάρτησης double Serial(), η οποία αξιοποιεί την συνάρτηση rand() για τη δημιουργία ψευδοτυχαίων αριθμών. Η Serial() υπολογίζει το ποσοστό των ρήψεων που βρέθηκαν εντός του μοναδιαίου κύκλου και έπειτα υπολογίζει και επιστρέφει την προσέγγιση που προέκυψε για την τιμή του π. Ένα πρόβλημα που προέκυψε με τη rand() είναι ότι δεν είναι thread-safe, συνεπώς για τις υλοποιήσεις με Pthreads και OpenMP χρειαστήκαμε την συνάρτηση my_drand(), η οποία υλοποιείται στα δοσμένα αρχεία my_rand.c και my_rand.h. Ο λόγος που δεν χρησιμοποιήθηκε η ίδια και στον σειριακό αλγόριθμο, είναι πως η rand() επιτύγχανε καλύτερα αποτελέσματα από άποψη χρόνου.

Οι προσεγγίσεις με τη χρήση Pthreads και OpenMP υλοποιούνται από τις συναρτήσεις void* Parallel_Pth(void* rank) και long long int Parallel_OMP() αντίστοιχα. Πέραν της αναγκαιότητας χρησιμοποίησης μιας thread-safe συνάρτησης για την δημιουργία των ψευδοτυχαίων αριθμών, αξίζει να σημειωθεί πως χρειάστηκε επίσης να χρησιμοποιήσουμε συγχρονισμό. Η αναγκαιότητα για συγχρονισμό προκύπτει από το γεγονός ότι κάθε στιγμή θα πρέπει μόνο ένα thread να αλλάζει τις global μεταβλητές που περιέχουν το πλήθος των ρήψεων εντός του μοναδιαίου κύκλου σε κάθε προσέγγιση. Για τον σκοπό αυτό επιλέξαμε τη χρήση ενός mutex στην περίπτωση των Pthreads, ενώ όσον αφορά την υλοποίηση με OpenMP τη χρήση ενός reduction clause με reduction operator την πρόσθεση και reduction variable τη μεταβλητή που σχετίζεται με το συνολικό πλήθος των ρήψεων εντός του μοναδιαίου κύκλου. Παρακάτω παρουσιάζονται οι δύο πίνακες για διαφορετικό αριθμό συνολικών ρήψεων, οι αντίστιχοι μέσοι όροι των αποτελεσμάτων, καθώς και το γράφημα που προκύπτει από αυτά τα αποτελέσματα.

Total throws: 10^8					
Threads	**Algorithm**	**Avg Pi**	**Avg Elapsed Time(s)**	**Speedup**	**Efficiency**
1	Serial	3.14174484	2.2317438	-	-
1	Pthreads	3.1415964	2.4417468	0.913994768	0.913994768
1	OpenMP	3.1415964	2.4532972	0.909691578	0.909691578
2	Pthreads	3.14159528	1.2556718	1.777330509	0.888665255
2	OpenMP	3.14159528	1.2649474	1.764297709	0.882148855
4	Pthreads	3.14159536	0.93541942	2.385821539	0.596455385
4	OpenMP	3.14159536	0.96273986	2.31811717	0.579529293
8	Pthreads	3.14158144	0.85882864	2.598590331	0.324823791
8	OpenMP	3.14158144	0.86565338	2.578103259	0.322262907

Total throws: 10^9					
Threads	**Algorithm**	**Avg Pi**	**Avg Elapsed Time (s)**	**Speedup**	**Efficiency**
1	Serial	3.141576212	22.119002	-	-
1	Pthreads	3.1415923	24.418104	0.905844369	0.905844369
1	OpenMP	3.1415923	24.527798	0.901793223	0.901793223
2	Pthreads	3.141592304	12.558876	1.761224651	0.880612325
2	OpenMP	3.141592304	12.622668	1.752323835	0.876161918
4	Pthreads	3.141593824	8.6609188	2.553886315	0.638471579
4	OpenMP	3.141593824	8.8537264	2.498270333	0.624567583
8	Pthreads	3.141594528	8.5811008	2.577641554	0.322205194
8	OpenMP	3.141594528	8.5914984	2.574522041	0.321815255



Στο παραπάνω γράφημα παρουσιάζουμε τους μέσους όρους της επιτάχυνσης κάθε πειράματος. Ως επιτάχυνση ορίζεται ο λόγος $S = \frac{T_s}{T_p}$, όπου στον αριθμητή έχουμε το χρόνο εκτέλεσης της σειριακής προσέγγισης και στον παρονομαστή τον αντίστοιχο της παράλληλης.

Παρατηρούμε πως η αποδοτικότητα και των δύο παράλληλων αλγορίθμων υστερεί σε σχέση με τη σειριακή προσέγγιση στην περίπτωση που χρησιμοποιούμε μόνο ένα νήμα. Αυτό είναι απολύτως αναμενόμενο, καθώς στην περίπτωση του ενός νήματος δεν εκμεταλλευόμαστε τις δυνατότητες των παράλληλων αλγορίθμων, ενώ έχουμε το overhead των υλοποιήσεων τους. Ένας ακόμη πιθανός λόγος της παρατηρούμενης υστέρησης σχετίζεται με το γεγονός ότι χρησιμοποιούν διαφορετικό γεννήτορα ψευδοτυχαίων αριθμών, ο οποίος δεν είναι το ίδιο αποδοτικός χρονικά με την rand(), η οποία εμπεριέχεται στη σειριακή προσέγγιση. Η επιλογή αυτή έγινε ώστε να έχουμε μια thread-safe συνάρτηση στις παράλληλες υλοποιήσεις, ωστόσο αναμένουμε να είναι ένας παράγοντας που επηρεάζει την αποδοτικότητα των παράλληλων αλγορίθμων.

Καθώς αυξάνουμε τον αριθμό των νημάτων, παρατηρούμε αύξηση της επιτάχυνσης και για τα δύο διαφορετικά νούμερα συνολικών ρήψεων. Αυτή η επιτάχυνση είναι πανομοιότυπη στους αλγορίθμους των Pthreads και OpenMP για αριθμό ρήψεων 10^9, ωστόσο παρουσιάζεται μια ελαφρά υστέρηση της OpenMP υλοποίησης στην περίπτωση των 10^8 συνολικών ρήψεων. Σε κάθε περίπτωση η παρατηρούμενη επιτάχυνση απέχει κατά πολύ από την ιδεατή περίπτωση, όπου θα ήταν ίση με τον αριθμό των νημάτων που πραγματοποιούν την παραλληλοποίηση κάθε φορά. Αυτό οφείλεται κυρίως στο συγχρονισμό μέσω των mutex και reduction clause στις δύο υλοποιήσεις αντίστοιχα, καθώς με την επιστράτευση αυτών μετατρέπουμε ένα κομμάτι του παράλληλου κώδικα μας σε σειριακό. Έτσι λοιπόν, σύμφωνα με το Νόμο του Amdal, η επιτάχυνσή του εκάστοτε αλγορίθμου θα είναι περιορισμένη, ανεξάρτητα από τον διαθέσιμο αριθμό των πυρήνων.

Για σταθερό αριθμό νημάτων, παρατηρούμε πως πετυχαίνουμε καλύτερες επιταχύνσεις και στις δύο υλοποιήσεις όταν έχουμε μεγαλύτερο αριθμό συνολικών ρήψεων. Σύμφωνα με το Νόμο του Gustafson, αυτό θα μπορούσε να είναι μια ένδειξη ότι το σύστημα μας έχει επιθυμητό μέγεθος εισόδου πιο κοντά στο 10^9. Επίσης, το γεγονός πως η απόδοση είναι καλύτερη για δύο παρά για τέσσερα νήματα και στις δύο υλοποιήσεις, υποδηλώνει ότι η χρήση δύο επιπλέον πυρήνων δεν προσφέρει το ανάλογο όφελος στο πρόβλημα μας.

Στην περίπτωση των 8 νημάτων παρατηρούμε ότι η απόδοση πέφτει αρκετά περισσότερο σε σχέση με τα προηγούμενα πειράματα. Αυτό οφείλεται στο ότι το υπολογιστικό μας σύστημα διαθέτει τέσσερις πυρήνες, συνεπώς όταν έχουμε αριθμό νημάτων μεγαλύτερο από τέσσερα ο scheduler αναλαμβάνει να κάνει χρονοδρομολόγηση. Γενικά καθώς αυξάνουμε τον αριθμό των νημάτων πέραν από τον αριθμό των πυρήνων του υπολογιστικού μας συστήματος, εισέρχονται παράγοντες όπως συνθήκες ανταγωνισμού για κοινόχρηστους πόρους και ζητήματα που αφορούν την μεταφορά δεδομένων από και προς την μνήμη, συνεπώς η παρατηρούμενη πτώση στην απόδοση ήταν αναμενόμενη.

● **Εργασία 1.2 (Pthreads)**

Σε αυτήν την άσκηση κατασκευάσαμε ένα παράλληλο πρόγραμμα πολλαπλασιασμού δύο πινάκων με χρήση Pthreads, το οποίο δέχεται ως είσοδο τις διαστάσεις τους και τον αριθμό των νημάτων, τους αρχικοποιεί με τυχαίες τιμές μεταξύ 0 και 1, πραγματοποιεί τον πολλαπλασιαμό τους και τυπώνει τους χρόνους εκτέλεσης των αρχικοποιήσεων και του παράλληλου υπολογισμού.

Βασιζόμενοι στον δοσμένο κώδικα πολλαπλασιασμού πίνακα - διανύσματος, τροποποιήσαμε κατάλληλα τμήματα αυτού, ώστε να υλοποιεί πολλαπλασιαμό μεταξύ δύο πινάκων. Κατά τη διάρκεια αυτής της διαδικασίας δημιουργήσαμε και τη συνάρτηση void Transpose(double MT[], double M[], int rows, int cols), η οποία υπολογίζει τον ανάστροφο ενός πίνακα. Στόχος αυτής της προσέγγισης είναι η αποθήκευση των στοιχείων του δεύτερου πίνακα σε συνεχόμενες θέσεις μνήμης, ώστε να πετύχουμε τα βέλτιστα χρονικά αποτελέσματα. Έτσι λοιπόν πραγματοποιούμε έναν ιδιάζοντα πολλαπλασιασμό πινάκων, όπου οι γραμμές του πρώτου πίνακα πολλαπλασιάζονται με τις γραμμές του αναστρόφου του δεύτερου. Όλη αυτή η προσέγγιση έχει ως στόχο την ελαχιστοποίηση των cache misses.

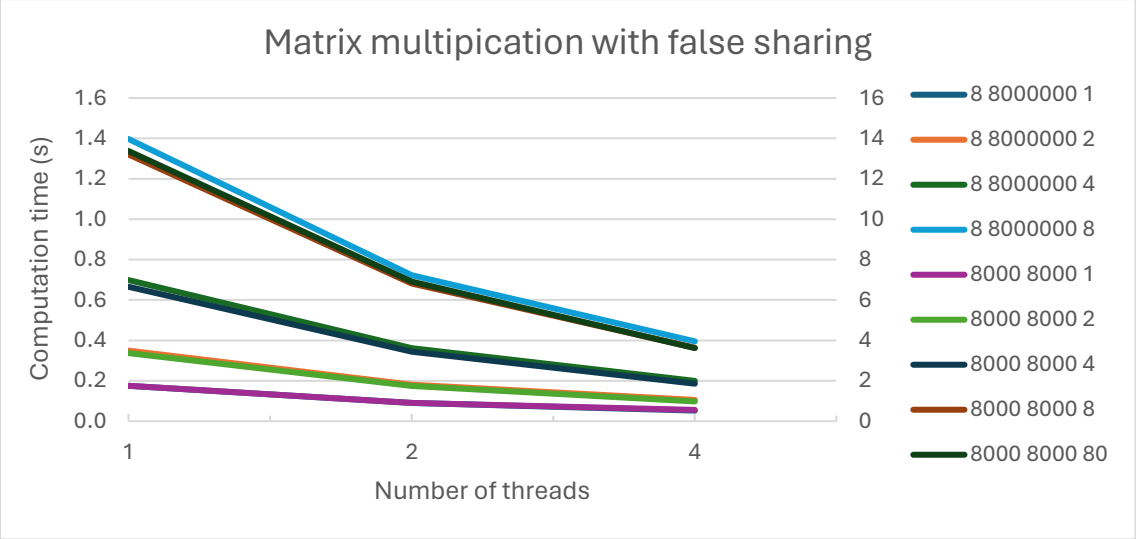
Σε μια προσπάθεια μείωσης του προβλήματος της ψευδούς κοινοχρησίας, η οποία αναμένουμε να είναι παρούσα κυρίως στις περιπτώσεις όπου ο τελικός πίνακας είναι σχετικά μικρός και χωράει σε λίγα cache lines, αποφασίσαμε να δοκιμάσουμε και μια δεύτερη προσέγγιση (*Minimized False Sharing – MFS*) κατά την οποία δημιουργούμε για το κάθε νήμα ξεχωριστά έναν ιδιωτικό χώρο μνήμης, μεγέθους ίσο με την περιοχή του τελικού πίνακα που πρόκειται να υπολογιστεί από αυτό. Με αυτόν τον τρόπο επιδιώκουμε να αυξήσουμε σημαντικά τα cache hits κάθε φορά που κάποιο νήμα εγγράφει μια καινούργια τιμή για τον τελικό πίνακα, ενώ όταν ολοκληρωθούν οι υπολογισμοί κάθε νήματος, τότε οι τελικές τιμές θα αντιγράφονται στον τελικό πίνακα. Ο χρόνος που χρειάζεται για την αντιγραφή των τιμών στον τελικό πίνακα δεν συνυπολογίζεται, επειδή θεωρούμε πως αυτή δεν θα ήταν μια δίκαιη σύγκριση, καθώς και πριν την αντιγραφή έχουμε αποθηκευμένα όλα τα στοιχεία του πίνακα και μπορούμε να τα χρησιμοποίησουμε για τυχόν περαιτέρω υπολογισμούς. Η μόνη διαφορά είναι ότι δεν βρίσκονται όλα σε συνεχόμενες θέσεις μνήμεις, γεγονός που

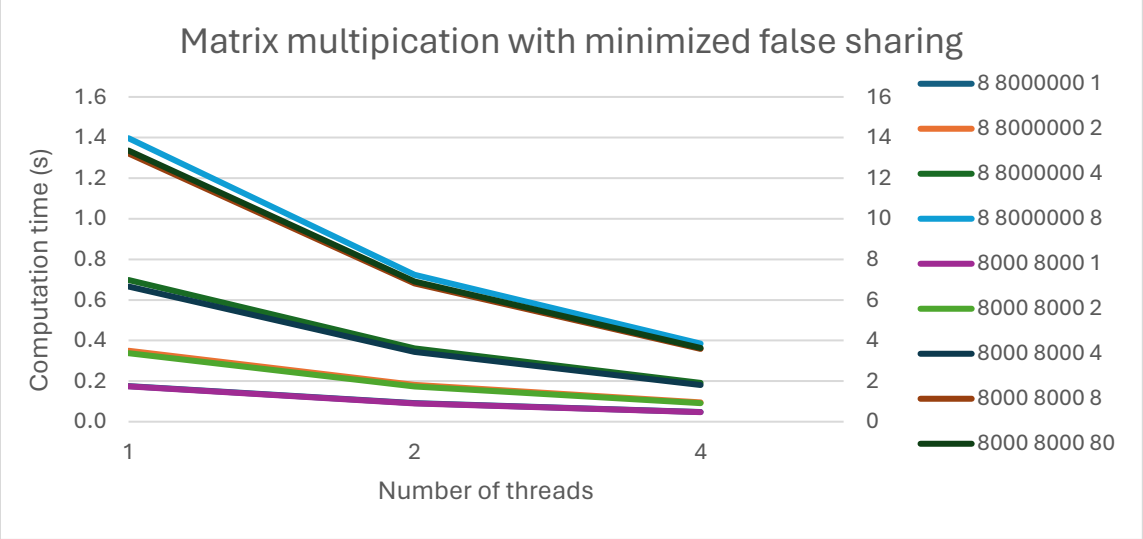
θεωρούμε ότι δεν επηρεάζει οποιαδήποτε μετέπειτα χρήση αυτών. Παρακάτω παρουσιάζουμε τους μέσους όρους των αποτελεσμάτων για τις δύο προσεγγίσεις που ακολουθήσαμε.

m	n	p	Number of Threads	Avg Elapsed Time (s)		Time decrease (%)
				FS	MFS	
8	8000000	1	1	0.17519938	0.17536346	-0.094
			2	0.09044795	0.09089618	-0.496
			4	0.05248437	0.04771471	9.088
8	8000000	2	1	0.34949056	0.3496612	-0.049
			2	0.18075212	0.18138694	-0.351
			4	0.10545188	0.09507418	9.841
8	8000000	4	1	0.69869882	0.69814204	0.080
			2	0.36145998	0.36087804	0.161
			4	0.19959106	0.19120606	4.201
8	8000000	8	1	1.3972674	1.3966726	0.043
			2	0.72262144	0.72315456	-0.074
			4	0.39525846	0.38473456	2.663
8000	8000	1	1	0.17528142	0.17412056	0.662
			2	0.09021721	0.09043083	-0.237
			4	0.05592604	0.04707241	15.831
8000	8000	2	1	0.33717038	0.33754126	-0.110
			2	0.17396758	0.17404426	-0.044
			4	0.09779616	0.09138899	6.552
8000	8000	4	1	0.66556568	0.66594338	-0.057
			2	0.34408824	0.34430702	-0.064
			4	0.18577862	0.1807324	2.716
8000	8000	8	1	1.3206626	1.3214368	-0.059
			2	0.681814	0.68144464	0.054
			4	0.36354786	0.35872878	1.326
8000	8000	80	1	13.377476	13.358456	0.142
			2	6.9138684	6.9113826	0.036
			4	3.6200904	3.6336738	-0.375

Αρχικά παρατηρούμε και στις δύο προσεγγίσεις μια συνεχή βελτίωση στους χρόνους εκτέλεσης καθώς αυξάνουμε τους αριθμούς των νημάτων για τις διάφορες διαστάσεις πινάκων. Ωστόσο, φαίνεται πως η δεύτερη προσέγγιση (*Minimized False Sharing – MFS*) επιτυγχάνει καλύτερα αποτελέσματα ενώ αυξάνουμε τον αριθμό των νημάτων στις περιπτώσεις όπου ο τελικός πίνακας είναι σχετικά μικρός (με άλλα λόγια, στις περιπτώσεις όπου το γινόμενο m*p είναι σχετικά μικρό). Αυτό φαίνεται καλύτερα και στο παρακάτω γράφημα, όπου οι τιμές για τον πίνακα (8000,80) τοποθετήθηκαν στον δεξιό κατακόρυφο άξονα, για καλύτερη οπτικοποίηση. Αντιθέτως, στις περιπτώσεις όπου οι τελικοί πίνακες έχουν μεγάλες διαστάσεις παρατηρείται λιγότερο αυτό το

φαινόμενο, ενώ συγκεκριμένα στον πίνακα με διαστάσεις (8000,80) έχουμε οριακά καλύτερα αποτελέσματα με τη μέθοδο *FS* παρά με τη μέθοδο *MFS*.

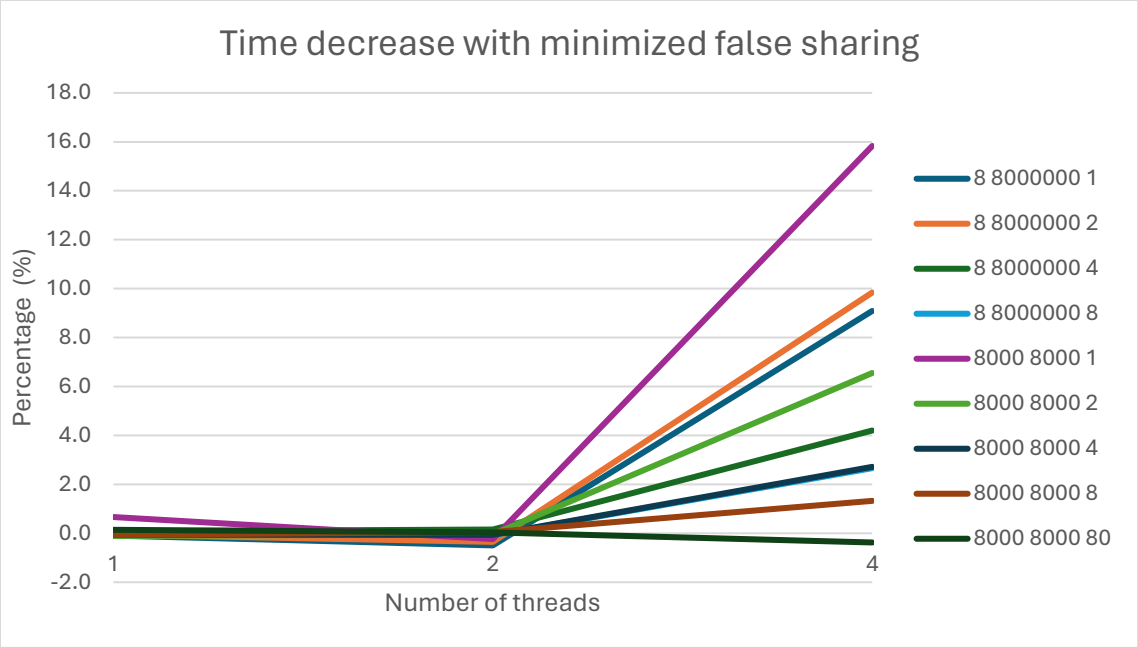




Οι παρατηρούμενες καλυτερες επιδόσεις στους σχετικά μικρούς πίνακες θεωρούμε πως οφείλονται στη ψευδή κοινοχρησία. Υπάρχουν πειράματα όπου ο τελικός πίνακας χωράει σε σχετικά λίγα cache lines, ακόμη και σε ένα στην περίπτωση των διαστάσεων (8, 8000000, 1), καθώς ο τελικός πίνακας που προκύπτει στη συγκεκριμένη περίπτωση αποτελείται από 8 doubles, δηλαδή 64 bytes συνολικά, όσο ακριβώς και το μέγεθος του cache line σε κάθε μία από τις 3 caches του συστήματος μας. Σε τέτοιες καταστάσεις το πρόβλημα δημιουργείται από τη στιγμή που κάποιο thread γράψει σε μια θέση του τελικού πίνακα, καθώς αυτή θα είναι invalidated κάθε φορά που ένα άλλο thread επιχειρήσει να γράψει σε μια άλλη θέση του, δεδομένου ότι αυτές οι δύο βρίσκονται στο ίδιο cache line. Έτσι λοιπόν, στις περιπτώσεις που το μέγεθος του τελικού πίνακα είναι σχετικά μικρό, αυτός χωράει σε λίγα cache lines και αυξάνεται κατ' επέκταση σημαντικά η πιθανότητα να παρατηρείται αυτό το φαινόμενο.

Αντίθετως, στις περιπτώσεις όπου ο τελικός πίνακας είναι πολύ μεγάλος, το παραπάνω φαινόμενο συμβαίνει πολύ σπάνια, καθώς ο πίνακας έχει χωριστεί σε πολλά cache lines, οπότε θα είναι λίγα εκείνα που θα περιέχουν εγγραφές τιμών από δύο διαφορετικά νήματα, συνεπώς το όλο φαινόμενο σε αυτές τις περιπτώσεις θα επηρεάζει ελάχιστα τα τελικά αποτελέσματα των πειραμάτων μας. Γι' αυτόν ακριβώς το λόγο παρατηρούμε πως η προσέγγιση *MFS* παράγει πανομοιότυπα αποτελέσματα με την πρώτη στην περίπτωση με το μεγαλύτερο τελικό πίνακα, διαστάσεων (8000,80). Αυτό συμβαίνει επειδή σε αυτήν την περίπτωση η ψευδής κοινοχρησία παρουσιάζεται αρκετά σπάνια, συνεπώς τα όποια οφέλη από την ελαχιστοποίηση της είναι μηδαμινά.

Αυτές οι παρατηρήσεις αποτυπώνονται καλύτερα και στο παρακάτω διάγραμμα, όπου παρουσιάζουμε το ποσοστό της χρονικής μείωσης των αποτελεσμάτων της δεύτερης μεθόδου σε σχέση με την πρώτη. Παρατηρούμε πως τα καλύτερα αποτελέσματα τα πετυχαίνουμε με μικρούς πίνακες, με εξαίρεση την περίπτωση του πίνακα (8000, 1), όπου ίσως η παρατηρούμενη μεγάλη αύξηση οφείλεται σε άλλους παράγοντες χρονοδρομολόγησης από το σύστημα μας.



● <u>Εργασία 1.3 (OpenMP)</u>

Σε αυτήν την άσκηση βασιστήκαμε και πάλι στο δοσμένο πρόγραμμα πολλαπλασιασμού πίνακα – διανύσματος και το τροποποιήσαμε κατάλληλα, ώστε να υπολογίζει αποδοτικά τον πολλαπλασιασμό ενός άνω τριγωνικού πίνακα με ένα διάνυσμα, με χρήση OpenMP. Το πρόγραμμα μας δέχεται ως είσοδο τον αριθμό των νημάτων, τη διάσταση n του πίνακα και του διανύσματος και εκτυπώνει το αποτέλεσμα του πολλαπλασιασμού καθώς και το συνολικό χρόνο εκτέλεσης των υπολογισμών.

Το πρώτο βήμα βελτιστοποίησης που ακολουθήσαμε ήταν να αποθηκεύσουμε μόνο τα μη μηδενικά στοιχεία του πίνακα, δηλαδή τα στοιχεία της κυρίας διαγωνίου και αυτά που βρίσκονται

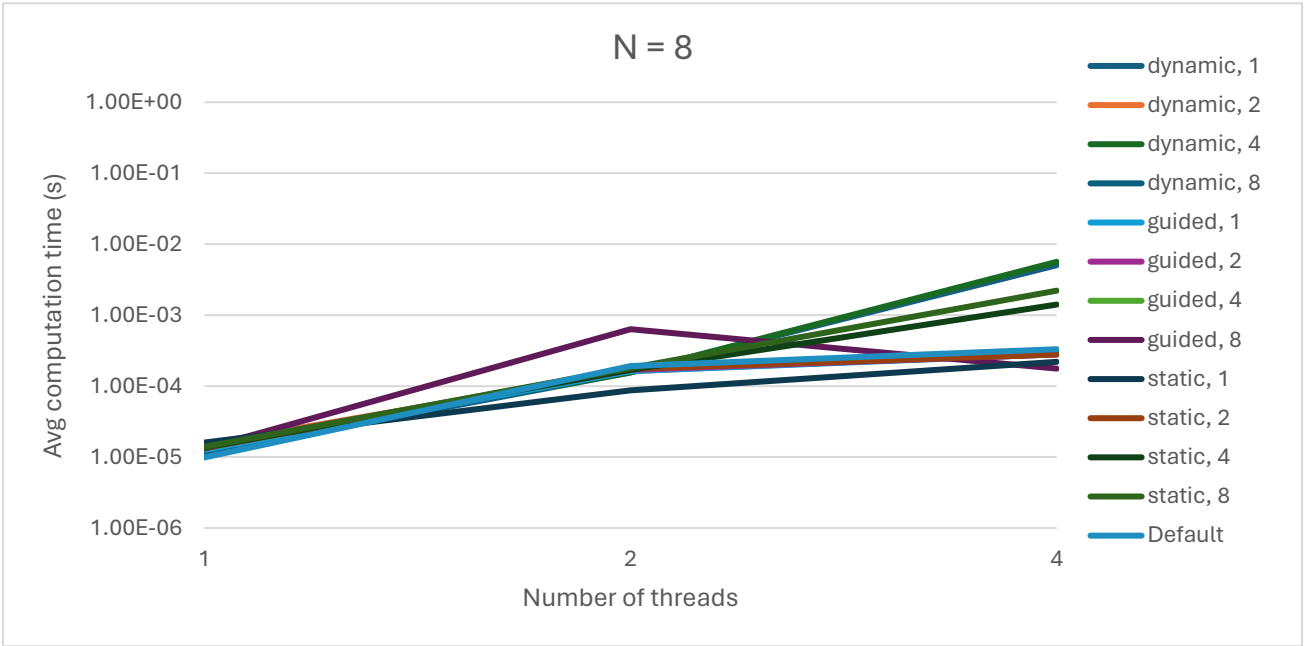
πάνω από αυτήν. Μέσω αυτής της επιλογής μας μπορούμε να εξοικονομήσουμε σημαντικά πόρους, καθώς ειδικά στην περίπτωση ενός σχετικά μεγάλου άνω τριγωνικού πίνακα καταλήγουμε να χρειαζόμαστε σχεδόν τη μισή μνήμη από αυτή που θα χρειαζόμασταν αν δεσμεύαμε χώρο για ολόκληρο τον πίνακα. Ωστόσο για να προχωρήσουμε σε αυτήν τη βελτιστοποίηση χρειαστήκαμε την πληροφορία του πλήθους των μη μηδενικών στοιχείων ανά γραμμή. Στην πρώτη γραμμή του πίνακα μας δεν υπάρχουν μηδενικά στοιχεία, στη δεύτερη υπάρχει ένα, στην τρίτη δύο κ.ο.κ.. Αν θέσουμε λοιπόν έναν δείκτη k, ο οποίος θα αναπαριστά σε ποια γραμμή βρισκόμαστε, ο k θα παίρνει τιμές από 0 έως και n, ενώ στην k γραμμή θα έχουμε n-k μη μηδενικά στοιχεία. Κατ' επέκταση έως την i-1 γραμμή θα έχουμε συνολικά:

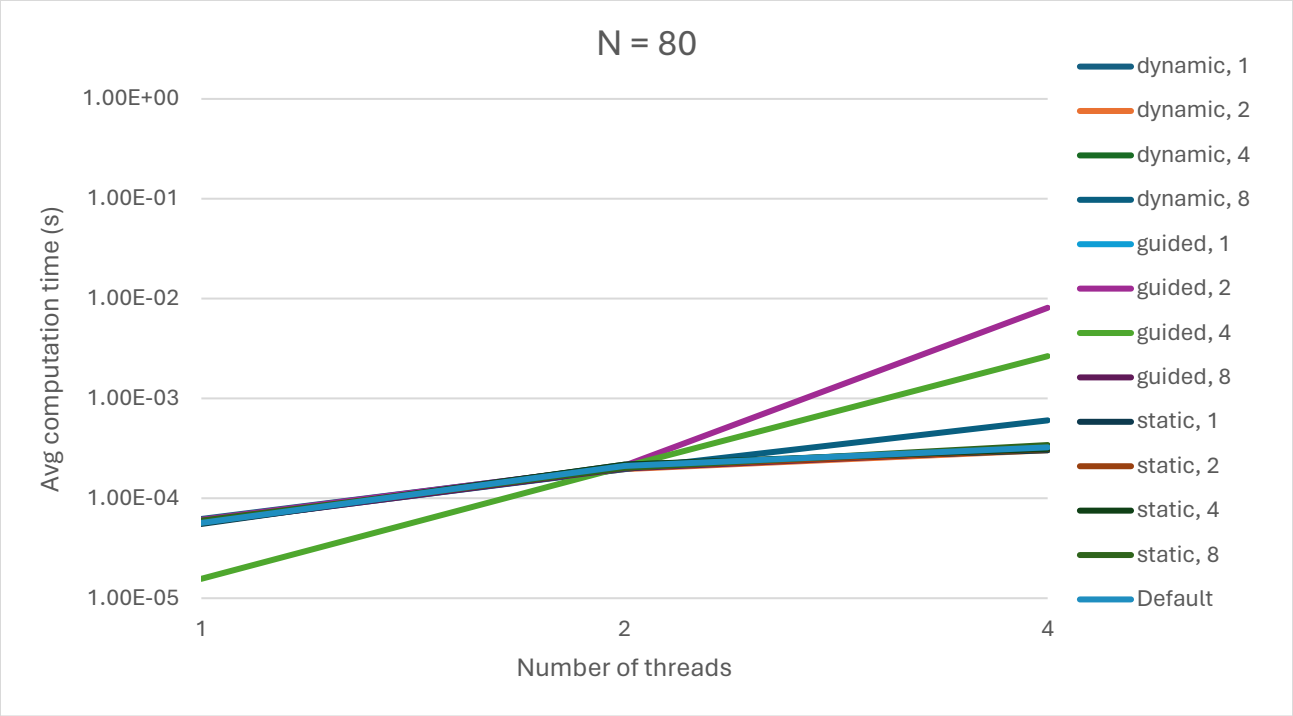
$$\sum_{k=0}^{i-1}(n-k) = \sum_{k=0}^{i-1} n - \sum_{k=0}^{i-1} k = i \cdot n - i \cdot \frac{i-2}{2} = i \cdot \left(n - \frac{i-2}{2}\right).$$

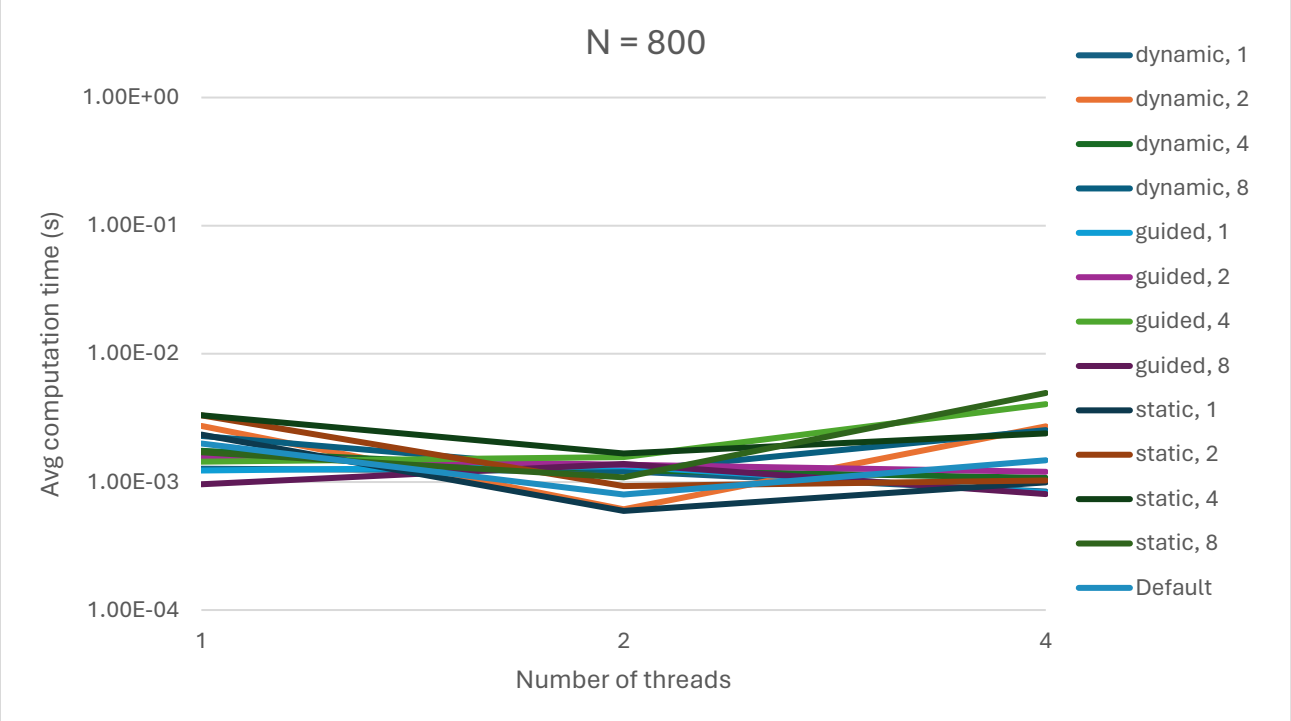
Στον παραπάνω συλλογισμό επιλέξαμε κατάλληλα το εύρος του k ώστε να συμβαδίζει με το εύρος των indices στα for loops του κώδικα μας, ενώ επειδή το άθροισμα μας θέλουμε να έχει n στοιχεία (όσες και οι γραμμές του πίνακα), διαλέξαμε να είναι έως το i-1. Συγκεκριμένα για i=n μπορούμε να βρούμε τα συνολικά μη μηδενικά στοιχεία του αρχικού πίνακα:

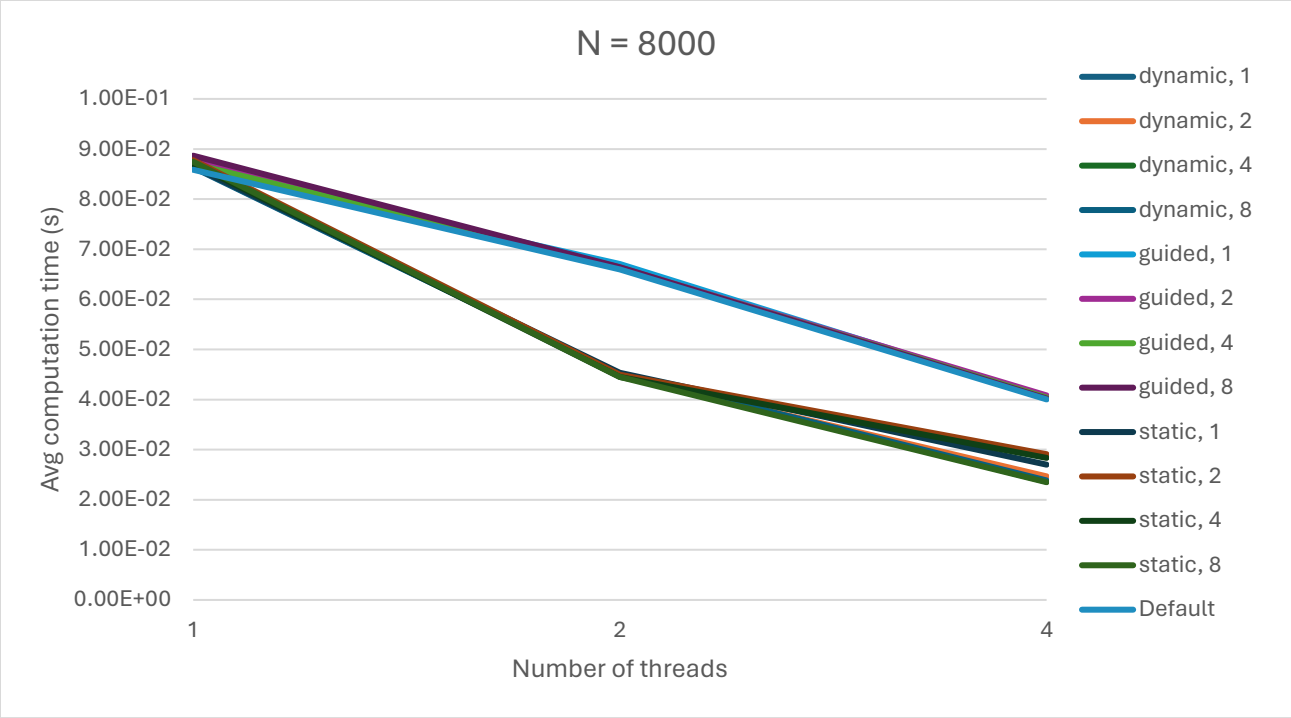
$$n \cdot \left(n - \frac{i-2}{2}\right) = n \cdot \frac{n+1}{2}$$

Τα παραπάνω αξιοποιήθηκαν ώστε να δεσμεύσουμε μόνο όσες θέσεις μνήμης χρειαζόμαστε για τη δημιουργία του πίνακα, καθώς και στις υλοποιήσεις των συναρτήσεων void Gen_matrix(double M[]) και double Omp_mat_vect(double M[], double x[], double y[]), οι οποίες αρχικοποιούν με τυχαίες τιμές μεταξύ μηδέν και ένα τον πίνακα και πραγματοποιούν τον παράλληλο πολλαπλασιασμό με το διάνυσμα. Σε ένα επόμενο στάδιο βελτιστοποίησης πειραματιστήκαμε με τη μεταβλητή περιβάλλοντος OMP_SCHEDULE, ώστε να δοκιμάσουμε διάφορες επιλογές ανάθεσης επαναλήψεων στα νήματα. Πιο συγκεκριμένα εξερευνίσαμε συνδιασμούς των μεταβλητών schedule type και chunk size, ώστε να διαπιστώσουμε ποιος λειτουργεί καλύτερα για διάφορα μεγέθη πινάκων. Η μεταβλητή schedule type καθορίζει τον τρόπο που κατανέμονται οι επαναλήψεις ανάμεσα στα νήματα, ενώ το chunk size αναπαριστά τον αριθμό των συνεχόμενων επαναλήψεων που ανατίθονται σε ένα συγκεκριμένο νήμα. Παραθέτουμε στις επόμενες σελίδες τα αποτελέσματα των πειραμάτων μας και τα γραφήματα που προκύπτουν από αυτά, ενώ οι συγκεντρωτικοί πίνακες υπάρχουν στο αντίστοιχο αρχείο Excel στον φάκελο exer1_3.




N = 80
1.00E+00
1.00E-01
1.00E-02
1.00E-03
1.00E-04
1.00E-05
Avg computation time (s)
1
2
4
Number of threads
dynamic, 1
dynamic, 2
dynamic, 4
dynamic, 8
guided, 1
guided, 2
guided, 4
guided, 8
static, 1
static, 2
static, 4
static, 8
Default


N = 800
1.00E+00
1.00E-01
1.00E-02
1.00E-03
1.00E-04
Avg computation time (s)
1
2
4
Number of threads
dynamic, 1
dynamic, 2
dynamic, 4
dynamic, 8
guided, 1
guided, 2
guided, 4
guided, 8
static, 1
static, 2
static, 4
static, 8
Default



Αρχικά παρατηρούμε πως σε μικρές και μεσαίες διαστάσεις πινάκων η αύξηση του αριθμού των νημάτων δεν είναι ευεργετική. Αυτό ήταν αναμενόμενο έως ένα βαθμό, γιατί σε αυτές τις περιπτώσεις έχουμε πιο συχνά το φαινόμενο της ψευδούς κοινοχρησίας καθώς αυξάνουμε τα νήματα, επειδή το διάνυσμα που προκύπτει από τον πολλαπλασιασμό καταλαμβάνει σχετικά λίγα cache lines. Μπορούμε επίσης να διαπιστώσουμε πως σε αυτές τις κλίμακες η προκαθορισμένη ανάθεση επαναλήψεων στα νήματα, δηλαδή ένας static scheduler με chunk size total_iterations/num_treads, φέρνει καλύτερα αποτελέσματα σε σχέση με πιο σύνθετες δρομολογίσεις, όπως guided ή dynamic. Αυτό θεωρούμε ότι συμβαίνει επειδή στις μικρές διαστάσεις πινάκων η ανομοιομορφία του φορτίου αναμέσα στις επαναλήψεις δεν επηρεάζει τόσο όσο το overhead μιάς πιο σύνθετης επιλογής ανάθεσης.

Για λίγο μεγαλύτερο πίνακα (n=800), παρατηρούμε μια ελαφρώς καλύτερη συμπεριφορά, όσον αφορά τις επιλογές dynamic και guided με chunk size 4, ωστόσο το φαινόμενο της ψευδούς κοινοχρησίας φαίνεται ακόμα να μας επηρεάζει, καθώς δεν παρουσιάζεται κάποια σημαντική βελτίωση με την αύξηση των νημάτων. Αντιθέτως, στην περίπτωση του μεγάλου πίνακα (n=8000), έχουμε αισθητά καλύτερα αποτελέσματα για περισσότερα νήματα, ενώ παράλληλα γίνεται αντιληπτό ότι η default μέθοδος υστερεί σημαντικά σε σχέση με τις guided και dynamic.

Ο λόγος που παρατηρείται αυτό συνδέεται με το γεγονός πως με τις επιλογές guided και dynamic οι επιλογές ανάθεσης επαναλήψεων σε νήματα πραγματοποιούνται κατά το χρόνο εκτέλεσης και όχι κατά το χρόνο μεταγλώττισης. Έτσι λοιπόν, δεδομένου ότι το φόρτο εργασίας ανά επανάληψη δεν είναι σταθερό λόγω του άνω τριγωνικού πίνακα, μια μη στατική μέθοδος είναι αναμενόμενο να φέρνει τα βέλτιστα αποτελέσματα, ειδικά στην περίπτωση ενός σχετικά μεγάλου πίνακα.

● Εργασία 1.4 (Pthreads)

Σε αυτήν την άσκηση υλοποιήσαμε τα δικά μας κλειδώματα ανάγνωσης – εγγραφής, τα οποία περιέχονται στο αρχείο *rwlocks.c* του φακέλου exer1_4 και αξιοποιούν ένα mutex, δύο μεταβλητές συνθήκης και τέσσερεις παραμέτρους καταμέτρησης, σύμφωνα με την δοσμένη περιγραφή. Εφαρμόσαμε αυτά τα κλειδώματα στο δοσμένο πρόγραμμα συνδεδεμένης λίστας του βιβλίου, τροποποιώντας την είσοδο του ώστε να δίνεται από το χρήστη ο αριθμός των νημάτων, τα ποσοστά ανάγνωσης και εγγραφής ως πιθανότητες, καθώς και ένας αριθμός εκ των 0,1 και 2. Οι αριθμοί αυτοί αντιπροσωπεύουν αντίστοιχα σειριακή προσέγγιση, παράλληλη προσέγγιση με προτεραιότητα στα νήματα ανάγνωσης και νήμματα εγγραφής αντίστοιχα.

Η υλοποίηση των κλειδωμάτων περιλαμβάνει δύο συναρτήσεις για τη δημιουργία και τη καταστροφή τους, δύο συναρτήσεις υπεύθυνες για τα κλειδώματα ανάγνωσης και εγγραφής αντίστοιχα και τέλος μία ακόμη υπεύθυνη για το ξεκλείδωμα. Συγκεκριμένα στην τελευταία περνιέται σαν επιπλέον όρισμα η πληροφορία για το αν πρόκειται για ξεκλείδωμα ανάγνωσης ή εγγραφης, καθώς και η προτεραιότητα που πρόκειται να ακολουθηθεί.

Η συνδεδεμένη λίστα αρχικοποιήθηκε με 1000 στοιχεία-κλειδιά μέσω του δοσμένου γεννήτορα ψευδοτυχαίων αριθμών my_rand, ενώ στη συνέχεια πραγματοποιήθηκαν συνολικά 500.000 ρουτίνες ανάγνωσης (*Member()*) και εγγραφής (*Insert() & Delete()*). Κάθε μία από τις πιθανότητες των ρουτινών Insert() και Delete() τέθηκε ίση με το μισό της δοθείσας πιθανότητας εγγραφής. Παρακάτω παρατίθενται τα αποτελέσματα των πειραμάτων μας.

Approach	Threads	Search percentage (%)	Insert percentage (%)	Delete percentage (%)	Avg Elapsed Time (s)	Speedup	Efficiency
Serial	1	90	5	5	39.095	-	-
		95	2.5	2.5	11.835		
		99.9	0.05	0.05	0.877		
Read_first	2	90	5	5	37.808	1.03	0.52
		95	2.5	2.5	14.167	0.84	0.42
		99.9	0.05	0.05	0.500	1.75	0.88
	4	90	5	5	40.721	0.96	0.24
		95	2.5	2.5	14.219	0.83	0.21
		99.9	0.05	0.05	0.354	2.48	0.62
Write_first	2	90	5	5	37.034	1.06	0.53
		95	2.5	2.5	14.130	0.84	0.42
		99.9	0.05	0.05	0.501	1.75	0.87
	4	90	5	5	39.595	0.99	0.25
		95	2.5	2.5	13.702	0.86	0.22
		99.9	0.05	0.05	0.353	2.49	0.62

Αυτό που παρατηρούμε αρχικά είναι ότι και με τις δύο προσεγγίσεις οι επιταχύνσεις που πετυχαίνουμε δεν είναι ιδιαίτερα ικανοποιητικές, με εξαίρεση την περίπτωση που έχουμε πολύ μικρό ποσοστό για εγγραφή (0,1%). Η παρατήρηση αυτή ήταν αναμενόμενη σε κάποιο βαθμό, λαμβάνοντας υπόψιν το overhead από τα συνεχόμενα κλειδώματα-ξεκλειδώματα, καθώς και το Νομο του Amdahl, αφού όσο αυξάνεται το ποσοστό των εγγραφών στη λίστα έχουμε όλο και

περισσότερα αντίστοιχα κλειδώματα, τα οποία έτσι όπως έχουν κατασκευαστεί επιτρέπουν την προσπέλαση της λίστας μόνο σε ένα νήμα. Το γεγονός αυτό έχει ως αποτέλεσμα να μην παρατηρούμε καλές επιταχύνσεις στις περιπτώσεις όπου οι εγγραφές αυξάνονται.

Όσον αφορά τη σύγκριση των δύο παράλληλων προσεγγίσεων, παρατηρούμε ότι οι χρόνοι είναι πανομοιότυποι, ανεξαρτήτως του αριθμού των νημάτων. Το γεγονός αυτό υποδηλώνει ότι οι δύο προσεγγίσεις είναι ισοδύναμες, κάτι που ίσως ισχύει τελικά. Εξάλλου για κάθε εισαγωγή ενός νέου στοιχείου στη λίστα καλείται η malloc, συνεπώς δεν υφίσταται κάποιου είδους τοπικότητα στα δεδομένα μας. Λαμβάνοντας υπόψιν το τελευταίο, η περίπτωση που δίνεται προτεραιότητα στα νήματα ανάγνωσης δεν εγγυάται παραπάνω cache hits και αυτός είναι ένας λόγος που δεν παρατηρούμε καλύτερους χρόνους με αυτή την προσέγγιση.

● Εργασία 1.5 (Pthreads)

Σε αυτήν την άσκηση καλούμαστε να συγκρίνουμε την ταχύτητα των pthread κλειδωμάτων με τη χρήση ατομικών εντολών. Πιο συγκεκριμένα, στο πρόγραμμα το οποίο υλοποιούμε αρχικοποιούμε με μηδέν μια κοινόχρηστη μεταβλητή στη main και στη συνέχεια το κάθε νήμα υλοποιεί ένα for loop με σταθερό αριθμό επαναλήψεων, και σε κάθε επανάληψη αυξάνει κατά ένα την κοινόχρηστη μεταβλητή. Ο χρήστης πρέπει να δώσει ως είσοδο τον αριθμό τον νημάτων, τον συνολικό αριθμό επαναλήψεων που θα τρέξουν τα νήματα μαζί και μια τιμή εκ των 0 ή 1, οι οποίες αντιστοιχούν στη χρήση mutexes και atomic instructions αντιστοιχα. Ο παρακάτω πίνακας περιέχει τα συγκεντρωτικά αποτελέσματα των πειραμάτων μας.

Total iterations	Threads	Approach	Avg Elapsed Time (s)	atomic_time/mutex_time
1.E+04	2	atomic	0.0007	0.290
		mutex	0.0023	
	4	atomic	0.0006	0.247
		mutex	0.0025	
1.E+05	2	atomic	0.0052	0.326
		mutex	0.0159	
	4	atomic	0.0074	0.561
		mutex	0.0132	
1.E+06	2	atomic	0.0312	0.500
		mutex	0.0623	
	4	atomic	0.0346	0.519
		mutex	0.0667	
1.E+07	2	atomic	0.2182	0.413
		mutex	0.5278	
	4	atomic	0.2064	0.337
		mutex	0.6115	
1.E+08	2	atomic	1.9910	0.368
		mutex	5.4063	
	4	atomic	2.1403	0.346
		mutex	6.1886	

Αυτό που γίνεται εύκολα αντιληπτό είναι το γεγονός πως ανεξαρτήτως του πλήθους των συνολικών επαναλήψεων και του αριθμού των νημάτων, οι ατομικές εντολές φαίνονται να είναι σημαντικά πιο γρήγορες από τα mutexes. Ο λόγος που συμβαίνει αυτό θεωρούμε ότι είναι το σχετικά μεγάλο overhead που παρουσιάζουν τα mutexes σε σχέση με τις ατομικές εντολές. Επιπλέον, οι ατομικές εντολές συνήθως υλοποιούνται σε επίπεδο υλικού, χρησιμοποιώντας συγκεκριμένες οδηγίες του επεξεργαστή που μπορούν να εκτελέσουν λειτουργίες ανάγνωσης-τροποποίησης-εγγραφής ατομικά. Αυτό σημαίνει ότι μπορούν να ολοκληρωθούν πολύ πιο γρήγορα από την απόκτηση και την απελευθέρωση ενός mutex κλειδώματος.

● **Εργασία 1.6 (OpenMP)**

Σε αυτήν την άσκηση υλοποιούμε ένα πρόγραμμα επίλυσης μεγάλων γραμμικών συστημάτων μέσω της απαλοιφής Gauss και μιας αντικατάστασης προς τα πίσω. Στο πρόγραμμα μας εμπεριέχεται ένας σειριακός και ένας παράλληλος αλγόριθμος με χρήση OpenMP, ενώ αναμένονται από το χρήστη ως ορίσματα το μέγεθος n του γραμμικού συστήματος, η προσέγγιση που θα ακολουθηθεί, και ο αριθμός των νημάτων (1 στην περίπτωση του σειριακού αλγορίθμου).

Ως πρώτο βήμα επιλέξαμε να τροποποιήσουμε τη συνάρτηση void Gen_matrix(double A[]), η οποία κατασκευάζει τους συντελεστές των μεταβλητών του συστήματος μας, ώστε τα στοιχεία της κυρίας διαγωνίου του πίνακα να μην είναι μηδέν, καθώς αυτά τα στοιχεία πρόκειται να είναι οι διαιρέτες στη μέθοδο απαλοιφής Gauss. Στη συνέχεια υπολογίζουμε, ανάλογα με τη προσέγγιση που ζητηθεί από το χρήστη, τους αντίστοιχους χρόνους εκτίμησης των αποτελεσμάτων των συναρτήσεων απαλοιφής Gauss και αντικατάστασης προς τα πίσω.

Συγκεκριμένα όσον αφορά την παράλληλη υλοποίηση με OpenMP, διαπιστώσαμε ότι δεν είναι δυνατό να παραλληλοποιηθεί το εξωτερικό for loop στην απαλοιφή Gauss. Αυτό οφείλεται στο γεγονός ότι παρουσιάζονται σειριακές εξαρτήσεις για τον υπολογισμό του ratio, καθώς υπάρχει περίπτωση κάποιο νήμα να διαβάσει ένα στοιχείο της διαγώνιου με λανθασμένη τιμή, αν αυτή αλλάξει από το προηγούμενο loop του i στην εξωτερική for. Αντιθέτως, η μεσαία for είναι παραλληλοποιήσιμη και αποδείχθηκε μάλιστα πιο αποδοτική επιλογή σε σχέση με την εσωτερική. Αξιοποιήσαμε ένα parallel for directive, ενώ επιλέξαμε να μην ορίσουμε κάποιο συγκεκριμένο scheduling, καθώς θεωρούμε ότι υφίσταται μια ομοιόρμοφη κατανομή φορτίου ανάμεσα στις επαναλήψεις του μεσαίου for loop.

Όσον αφορά την αντικατάσταση προς τα πίσω, διαπιστώσαμε ότι στην εξωτερική for υφίστανται κάποιες εξαρτήσεις οι οποίες δεν διευκολύνουν την παραλληλοποίηση, καθώς υπάρχει περίπτωση στην εντολή x[row] -= A[row][col]*x[col], κάποιο νήμα να μην έχει ολοκληρώσει τον υπολογισμό του x[col] προτού ζητηθεί από ένα άλλο νήμα για τον υπολογισμό του x[row]. Αντιθέτως, διαπιστώθηκε πως είναι εφικτή η παραλληλοποίηση της εσωτερικής for μέσω ενός parallel directive και ενός reduction clause με reduction operator την αφαίρεση και reduction variable την temp. Όπως και στην απαλοιφή Gauss, έτσι και εδώ επιλέξαμε να μην ορίσουμε κάποιο συγκεκριμένο scheduling, καθώς θεωρούμε ότι υφίσταται μια ομοιόρμοφη κατανομή φορτίου ανάμεσα στις επαναλήψεις του μεσαίου for loop.

Τα συγκεντρωτικά αποτελέσματα της παράλληλης υλοποίησης για διάφορα μεγέθη πινάκων συνοψίζονται στον παρακάτω πίνακα, όπου περιέχονται και τα αντίστοιχα της σειριακής, με μόνη εξαίρεση την περίπτωση n=10.000, η οποία παραλήφθηκε λόγω του εξαιρετικά μεγάλου χρόνου εκτέλεσης.

N	Approach	Threads	Functionality	Avg Elapsed Time (s)	Speedup	Efficiency
500	Serial	1	Back substitution	3.623E-04	-	-
			Gauss elimination	1.207E-01		
	Parallel	2	Back substitution	1.292E-03	0.28	0.14
			Gauss elimination	6.366E-02	1.90	0.95
		4	Back substitution	1.261E-03	0.29	0.07
			Gauss elimination	3.394E-02	3.56	0.89
1000	Serial	1	Back substitution	1.470E-03	-	-
			Gauss elimination	9.551E-01		
	Parallel	2	Back substitution	2.899E-03	0.51	0.25
			Gauss elimination	5.012E-01	1.91	0.95
		4	Back substitution	2.791E-03	0.53	0.13
			Gauss elimination	2.711E-01	3.52	0.88
2000	Serial	1	Back substitution	5.911E-03	-	-
			Gauss elimination	7.831E+00		
	Parallel	2	Back substitution	7.156E-03	0.83	0.41
			Gauss elimination	4.101E+00	1.91	0.95
		4	Back substitution	6.413E-03	0.92	0.23
			Gauss elimination	2.202E+00	3.56	0.89
5000	Serial	1	Back substitution	3.659E-02	-	-
			Gauss elimination	1.228E+02		
	Parallel	2	Back substitution	2.861E-02	1.28	0.64
			Gauss elimination	6.403E+01	1.92	0.96
		4	Back substitution	2.115E-02	1.73	0.43
			Gauss elimination	3.448E+01	3.56	0.89
8000	Serial	1	Back substitution	9.317E-02	-	-
			Gauss elimination	5.020E+02		
	Parallel	2	Back substitution	6.208E-02	1.50	0.75
			Gauss elimination	2.624E+02	1.91	0.96
		4	Back substitution	4.291E-02	2.17	0.54
			Gauss elimination	1.407E+02	3.57	0.89
10000	Parallel	2	Back substitution	9.155E-02	-	-
			Gauss elimination	5.111E+02		
		4	Back substitution	6.068E-02		
			Gauss elimination	2.747E+02		

Αρχικά παρατηρούμε ότι για την μέθοδο απαλοιφής Gauss έχουμε την ίδια αποδοτικότητα για δεδομένο αριθμό νημάτων, ανεξαρτήτως του μεγέθους του αρχικού συστήματος, γεγονός που υποδηλώνει πως είναι ισχυρά κλιμακώσιμη με την OpenMP. Αξιοσημείωτο είναι επίσης ότι η απόδοση είναι σταθερά καλύτερη για δύο παρά για τέσσερα νήματα για τις διάφορες τιμές του n, κάτι που υποδεικνύει ότι η χρήση δύο επιπλέον πυρήνων δεν προσφέρει το ανάλογο όφελος. Αν λάβουμε μάλιστα υπόψιν ότι η αποδοτικότητα με δύο νήματα είναι σταθερά 95-96%, γίνεται εύκολα

αντιληπτό πως ο ιδανικός αριθμός νημάτων για την υλοποίηση μας είναι τα δύο. Τα επίπεδα της απόδοσης θεωρούμε πως είναι άκρως ικανοποιητικά, αν λάβουμε υπόψιν τις προαναφερθείσες εξαρτήσεις στο εξωτερικό for loop και ό,τι αυτές επιφέρουν σύμφωνα με το Νόμο του Amdahl.

Όσον αφορά την αντικατάσταση προς τα πίσω, παρατηρούμε πως για σχετικά μικρά συστήματα έχουμε πολύ χαμηλές αποδόσεις, είτε για δύο είτε για τέσσερα νήματα. Αυτό υποδηλώνει ότι στις περιπτώσεις με μικρό n, το overhead της επαναλαμβανόμενης δημιουργίας και σύζευξης των νημάτων στην εσωτερική for loop παίζει καθοριστικό ρόλο στα τελικά αποτελέσματα. Το σημείο όπου η παραλληλοποίηση της αντικατάστασης προς τα πίσω ξεκινά να έχει επιτάχυνση μεγαλύτερη του ένα είναι όταν πάμε σε μεγέθη συστημάτων με n=5.000, ωστόσο και πάλι η αποδοτικότητα της παράλληλης υλοποίησης υστερεί κατά πολύ σε σχέση με την αντίστοιχη της απαλοιφής Gauss. Η αιτία είναι και πάλι οι αλληλοεξαρτήσεις στην εξωτερική for και ότιδήποτε συνεπάγεται από αυτές λόγω του Νόμου του Amdahl.

Σε κάθε περίπτωση ωστόσο, ανεξαρτήτως του μεγέθους του αρχικού συστήματος, η συνεισφορά της αντικατάστασης προς τα πίσω στον συνολικό χρόνο είναι πολύ μικρή, συνεπώς μέσω της παράλληλης υλοποίησης επιτυγχάνουμε τελικά μια σημαντική επιτάχυνση στην επίλυση του συστήματος.