HELLENIC REPUBLIC

National and Kapodistrian University of Athens

EST. 1837

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Εργασία 1 – Προγραμματισμός με Pthreads & OpenMP

Παράλληλα Υπολογιστικά Συστήματα

(M127)

Διδάσκων: Βασίλειος Καρακώστας

Παναγιώτα Γύφτου, 7115112400025

Περιεχόμενα

Άσκηση 1 (Pthreads & OpenMP)

1.1. Περιγραφή προβλήματος

Η υλοποίηση υλοποιεί την εκτίμηση της τιμής του π με τη μέθοδο Monte Carlo: ρίχνει «βελάκια» (τυχαία σημεία) στο τετράγωνο $[-1,1]\times[-1,1]$ και μετρά πόσα πέφτουν μέσα στον κύκλο ακτίνας 1. Ο λόγος των «χτυπημάτων» προς τον συνολικό αριθμό βολών, επί 4, δίνει την προσέγγιση του π . Οι παραλλαγές με Pthreads και OpenMP παραλληλοποιούν τον βρόχο δειγματοληψίας, κατανέμοντας τις ρίψεις σε πολλαπλούς threads για ταχύτερη εκτέλεση, ενώ διατηρούν την ίδια βασική Monte Carlo λογική.

1.2. Περιγραφή Λύσης

Η σειριακή υλοποίηση ξεκινά με την αρχικοποίηση ενός μετρητή «χτυπημάτων» (arrows = 0). Στη συνέχεια εκτελεί έναν απλό βρόχο for με τόσες επαναλήψεις όσες οι ζητούμενες βολές (num_darts), όπου σε κάθε επανάληψη υπολογίζονται τυχαίες συντεταγμένες (x, y) στο διάστημα [-1, 1] και ελέγχεται αν το σημείο βρίσκεται εντός του κύκλου ακτίνας 1. Αν ναι, ο τοπικός μετρητής αυξάνεται. Στο τέλος, το π υπολογίζεται και εκτυπώνονται τα αποτελέσματα μαζί με τον συνολικό χρόνο εκτέλεσης.

Η υλοποίηση με Pthreads οργανώνει την ίδια λογική σε πολλαπλά νήματα. Αρχικά υπολογίζεται πόσες βολές αναλογούν σε κάθε νήμα (λαμβάνοντας υπόψη πιθανό υπόλοιπο. Κάθε νήμα ξεκινάει με το δικό του βρόχο for παράγει τυχαία σημεία, υπολογίζει πρώτα τοπικά τα "χτυπήματά" του και, αφού ολοκληρώσει τον βρόχο, κλειδώνει μία φορά το mutex για να τα προσθέσει στον κοινό μετρητή arrows, εξασφαλίζοντας έτσι ακεραιότητα των δεδομένων (χωρίς race conditions) κατά την ταυτόχρονη πρόσβαση των νημάτων και ταυτόχρονα ελαχιστοποιώντας το κόστος των κλειδωμάτων. Κάνοντας την πρόσθεση των τοπικών "χτυπημάτων" **μετά το πέρας** ολόκληρου του βρόχου for πετυχαίνουμε δύο πράγματα:

1. **Ελαχιστοποίηση της συμφόρησης**: το mutex κλειδώνει μόνο μία φορά ανά νήμα, αντί για μια φορά ανά βολή.
2. **Γρήγορη και ασφαλή συγχώνευση**: η κρίσιμη περιοχή είναι μόνο μια απλή, σύντομη πρόσθεση, χωρίς κίνδυνο race conditions.

Όταν όλα τα νήματα ολοκληρώσουν (με pthread_join()), υπολογίζεται η τελική τιμή του π.

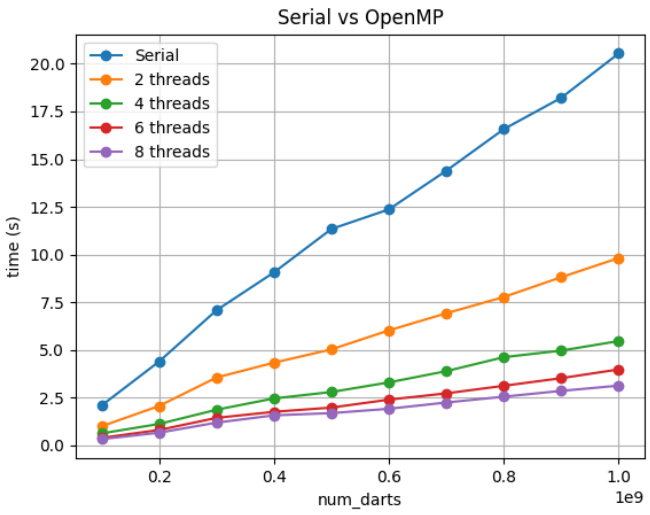
Στην OpenMP υλοποίηση, το πρόγραμμα ρυθμίζει πρώτα τον αριθμό των threads με omp_set_num_threads(thread_count). Έπειτα εισέρχεται σε μια παράλληλη περιοχή (#pragma omp parallel), όπου κάθε thread εκτελεί τμήμα του βρόχου δειγματοληψίας. Η κατανομή των επαναλήψεων γίνεται αυτόματα μέσω #pragma omp for reduction(+ : arrows), το οποίο διαχειρίζεται ξεχωριστούς μετρητές ανά thread και τους συγχωνεύει στο τέλος χωρίς χειροκίνητο κλείδωμα. Με την έξοδο από την παράλληλη περιοχή, υπολογίζεται η προσεγγιστική τιμή του π. Να σημειωθεί ότι για κάθε thread το seed είναι διαφορετικό , καθώς αν όλα τα threads χρησιμοποιούσαν το ίδιο seed και την ίδια ακολουθία, θα παρήγαγαν τις ίδιες «τυχαίες» τιμές, με αποτέλεσμα όλες οι ρίψεις να συμπίπτουν μεταξύ των threads και να χάνεται ολόκληρο το νόημα της παραλληλίας.

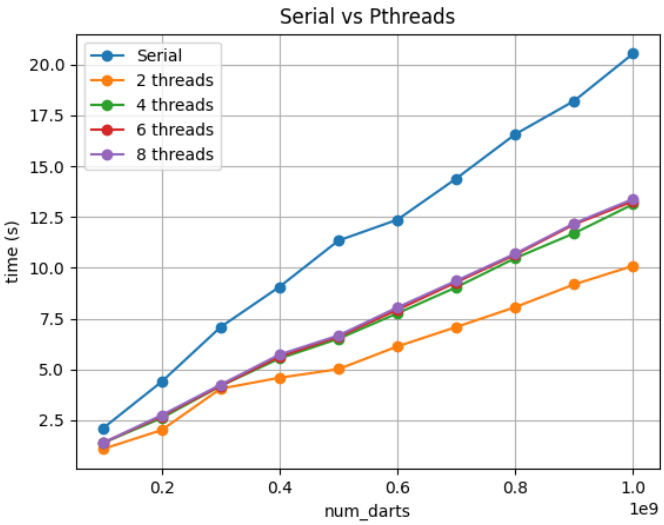
1.3. Πειραματικά Αποτελέσματα

Είσοδος:

1. πλήθος ρίψεων: από 10^8 έως 10^9 με βήμα 10^8,
2. νήματα: 1, 2, 4, 6 και 8.

Στο διάγραμμα της OpenMP βλέπουμε φανερή βελτίωση της απόδοσης καθώς αυξάνεται ο αριθμός των νημάτων και το πλήθος των ρίψεων. Για μικρές εργασίες (100 M darts), ο σειριακός κώδικας ολοκληρώνει σε ~2 s, ενώ με 2 threads η OpenMP έκδοση πέφτει σε ~1 s, με 4 threads σε ~0.6 s και με 8 threads σε ~0.4 s δηλαδή στο μέγιστο πλήθος των πυρήνων (8). Καθώς ανεβαίνουμε σε 1×10^9 βολές, ο σειριακός χρόνος εκτοξεύεται στα ~20.5 s, ενώ με 8 threads παραμένει μόλις ~3.1 s, επιβεβαιώνοντας ότι το overhead παραλληλοποίησης (δημιουργία, συγχρονισμός) παραμένει πολύ χαμηλό και ότι η κατανομή της εργασίας είναι σχεδόν ιδανική.





Αντίθετα, στο διάγραμμα της Pthreads υλοποίησης διακρίνουμε δύο περιοχές: αρχικά, με 2 νήματα, το συνολικό κόστος πέφτει από ~20 s σε ~10 s, αλλά μόλις προσθέσουμε 4 νήματα, η πτώση επιβραδύνεται και σταθεροποιείται γύρω στα 12–13 s. Με 6 ή 8 νήματα ο χρόνος μειώνεται οριακά περαιτέρω. Αυτό οφείλεται στη συμφόρηση στο mutex: κάθε νήμα, αφού ολοκληρώσει τον κύκλο των δικών του ρίψεων, συγκεντρώνεται στο ίδιο κλειδωμένο τμήμα για να αθροίσει τα αποτελέσματά του, δημιουργώντας ουρά αναμονής και καταστρέφοντας το όφελος της παραλληλίας.

Επομένως, ενώ και οι δύο εκδοχές παραλληλοποιούν το πρόβλημα των Monte Carlo ρίψεων, μόνο η OpenMP καταφέρνει να εκμεταλλευτεί πλήρως τους διαθέσιμους πυρήνες, αποφεύγοντας τα κόστη lock contention που "φρενάρουν" την Pthreads υλοποίηση.

Implementation	Threads	Darts	Arrows	Time	pi
Serial	-	100000000	78539790	2091534	3141592
OpenMp	2	100000000	78537319	1001770	3141493
Pthreads	2	100000000	78537319	1081786	3141493
OpenMp	4	100000000	78533591	627631	3141344
Pthreads	4	100000000	78533591	1374306	3141344
OpenMp	6	100000000	78539358	394725	3141574
Pthreads	6	100000000	78539357	1377745	3141574
OpenMp	8	100000000	78538811	329464	3141552
Pthreads	8	100000000	78538811	1380675	3141552
Serial	-	200000000	157079415	4393280	3141588
OpenMp	2	200000000	157080085	2064896	3141602
Pthreads	2	200000000	157077316	2016277	3141546
OpenMp	4	200000000	157077580	1123322	3141552
Pthreads	4	200000000	157077580	2607182	3141552
OpenMp	6	200000000	157073621	808332	3141472
Pthreads	6	200000000	157079499	2696032	3141590
OpenMp	8	200000000	157078684	666423	3141574
	.....	.....	.....		
Pthreads	8	200000000	157078684	2748736	3141574
Serial	1	900000000	706858282	18215669	3141592
OpenMp	2	900000000	706858399	8815806	3141593
Pthreads	2	900000000	706858322	9185828	3141593
OpenMp	4	900000000	706858663	4967868	3141594
Pthreads	4	900000000	706855901	11702410	3141582
OpenMp	6	900000000	706858930	3522994	3141595
Pthreads	6	900000000	706856859	12140411	3141586
OpenMp	8	900000000	706851531	2856787	3141562
Pthreads	8	900000000	706854752	12192753	3141577
Serial	1	1000000000	785398045	20542048	3141592

OpenMp	2	1000000000	785401568	9817469	3141606
Pthreads	2	1000000000	785399289	10098902	3141597
OpenMp	4	1000000000	785399388	5467509	3141598
Pthreads	4	1000000000	785401176	13131824	3141605
OpenMp	6	1000000000	785397549	3981579	3141590
Pthreads	6	1000000000	785399921	13275920	3141600
OpenMp	8	1000000000	785398203	3131485	3141593
Pthreads	8	1000000000	785398203	13386479	3141593

Άσκηση 2 (Pthreads)

2.1. Περιγραφή προβλήματος

Η εν λόγω άσκηση υλοποιεί και συγκρίνει δύο τεχνικές ασφαλούς αύξησης μιας κοινόχρηστης μεταβλητής σε πολυνηματικό περιβάλλον, με βάση τον χρόνο εκτέλεσης.

1. **Mutex-based mutual exclusion** (κλειδώνοντας/ξεκλειδώνοντας έναν mutex)
2. **Ατομικές λειτουργίες** (__atomic_fetch_add)

Και στις δύο περιπτώσεις χρησιμοποιεί τον ίδιο αριθμό νημάτων και επαναλήψεων για να αξιολογήσει την απόδοσή τους.

2.2. Περιγραφή Λύσης

Κάθε τεχνική ακολουθεί τον ίδιο βασικό «σκελετό»: αρχικά δημιουργούνται *n* νημάτα, μοιράζεται ο αριθμός των επαναλήψεων σε κάθε νήμα, εκτελείται σε κάθε επανάληψη η αύξηση του κοινού μετρητή. Το μόνο που διαφοροποιείται είναι ο τρόπος με τον οποίο προστατεύεται η κρίσιμη περιοχή όπου γίνεται η αύξηση του μετρητή.

Στην προσέγγιση με **mutex-based mutual exclusion**, δηλώνεται ένας κοινόχρηστος mutex (pthread_mutex_t Mutex) που προστατεύει την κοινή μεταβλητή shared_var. Κάθε νήμα, μέσα στη συνάρτηση culc_sum, εκτελεί σε κάθε επανάληψη την κλήση

```
pthread_mutex_lock(&Mutex);
 shared_var++;
pthread_mutex_unlock(&Mutex);
```

Το pthread_mutex_lock φροντίζει να εισέρχεται μόνο ένα νήμα κάθε φορά στην κρίσιμη περιοχή, και το pthread_mutex_unlock να την απελευθερώνει, αποφεύγοντας συνθήκες ταυτόχρονης εγγραφής αλλά επιβαρύνοντας με το κόστος lock/unlock και πιθανών context switches .

Στην τεχνική με **ατομικές λειτουργίες**, αντί mutex χρησιμοποιείται η gcc ατομική πράξη

```
__atomic_fetch_add(&shared_var, 1, __ATOMIC_SEQ_CST);
```

Μέσα στη συνάρτηση culc_sum_atomic, κάθε νήμα κάνει απευθείας ατομική αύξηση της shared_var, εξασφαλίζοντας τη διατήρηση της σωστής σειράς εκτέλεσης χωρίς ρητό συγχρονισμό μέσω κλειδώματος/ξεκλειδώματος. Η CPU αναλαμβάνει να εκτελέσει την εντολή ατομικά, μειώνοντας την επιβάρυνση σε σύγκριση με το mutex.

2.3. Πειραματικά Αποτελέσματα

Είσοδος:

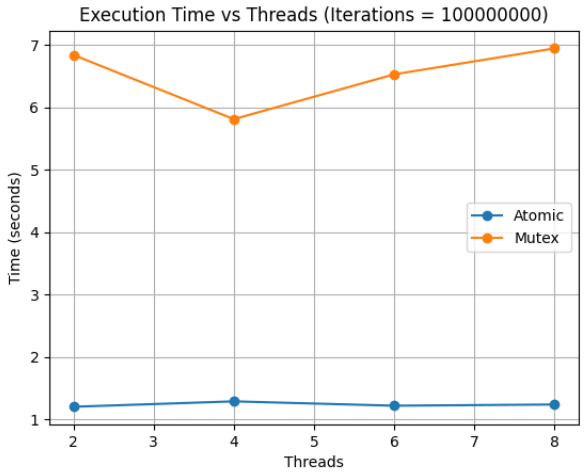
1. Αριθμός επαναλήψεων : από 10^8 έως 5*10^8 με βήμα 10^8
2. Νήματα: 2,4,6 και 8

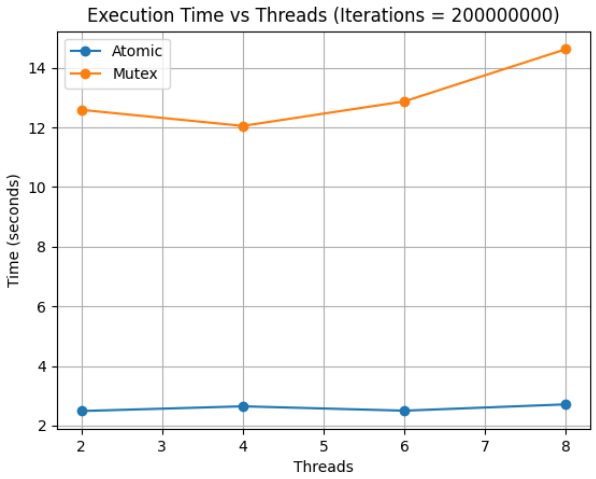
Παρατηρούμε ότι, σε κάθε γράφημα, η ατομική μέθοδος (μπλε) εκτελείται πολύ πιο γρήγορα από την προσέγγιση με mutex (πορτοκαλί).

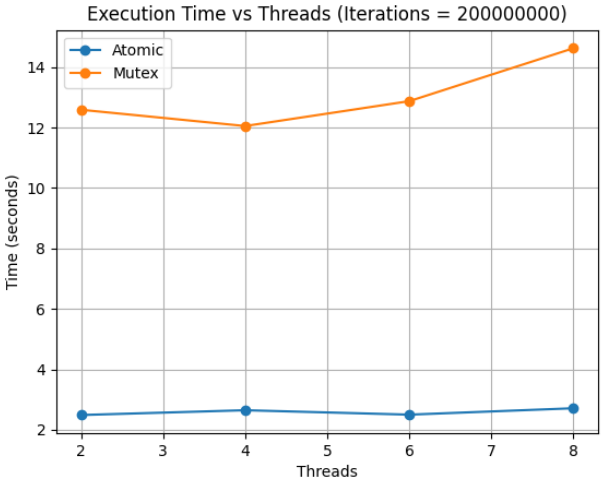
Σε κάθε διάγραμμα η καμπύλη της ατομικής μεθόδου (μπλε) εμφανίζει σχεδόν σταθερό χρόνο εκτέλεσης καθώς αυξάνουμε τα νήματα από 2 σε 8. Υπάρχει μία μικρή βελτίωση γύρω στα 4-6 νήματα, όπου ο χρόνος φτάνει σε ένα τοπικό ελάχιστο, αλλά η διαφορά με τις τιμές στους 2 ή στους 8 πυρήνες είναι μικρή. Αυτό δείχνει ότι η χρήση της εντολής __atomic_fetch_add κλιμακώνεται πολύ ήπια, με ελάχιστη hardware επιβάρυνση και χωρίς καθυστέρηση από blocking.

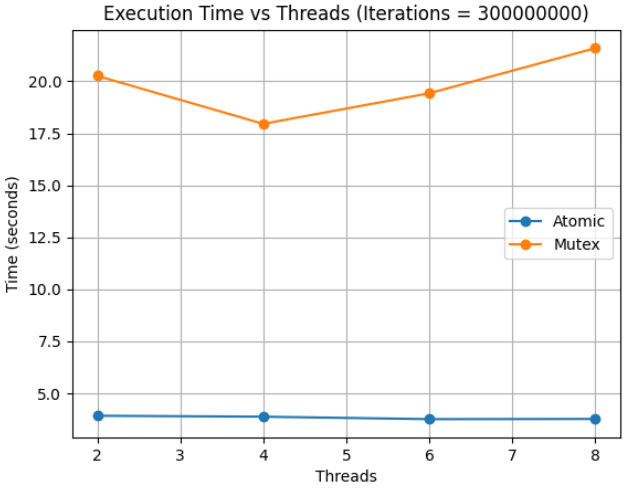
Η προσέγγιση με mutex (πορτοκαλί) αρχικά ωφελείται όταν περνάμε από 2 σε 4 νήματα, ο χρόνος μειώνεται επειδή κάθε νήμα κάνει λιγότερα lock/unlock και συνεπώς λιγότερη συμφόρηση. Ωστόσο, μόλις ξεπεράσουμε τα 4 νήματα, ο χρόνος αυξάνεται σταθερά, λόγω του μεγάλου κόστους των κλήσεων pthread_mutex_lock/pthread_mutex_unlock.

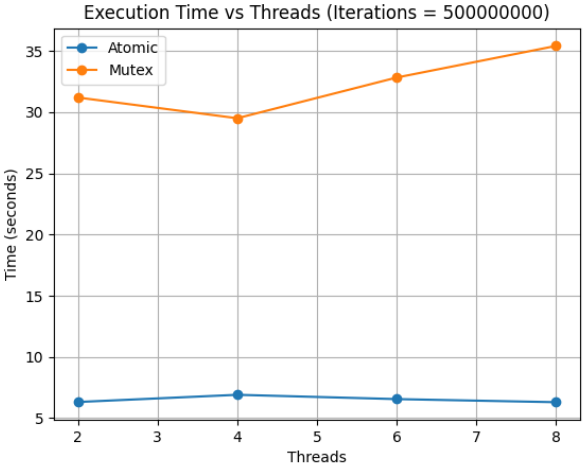
Οι atomic operations υπερτερούν σε ταχύτητα, κλιμάκωση και σταθερότητα σε σχέση με τους mutex για την απλή περίπτωση προσαύξησης ενός κοινόχρηστου μετρητή. Οι mutex είναι γενικότερα απαραίτητοι όταν απαιτείται προστασία πιο σύνθετων περιοχών κώδικα, αλλά για στοιχειώδεις λειτουργίες όπως ++/--, οι atomic πράξεις είναι σαφώς πιο αποτελεσματικές.











Lock	Threads	Iterations	Time
Mutex	2	100000000	6840602
Atomic	2	100000000	1199012
Mutex	4	100000000	5811207
Atomic	4	100000000	1285395
Mutex	6	100000000	6529889
Atomic	6	100000000	1215960
Mutex	8	100000000	6946317
Atomic	8	100000000	1235112
Mutex	2	200000000	12589596
Atomic	2	200000000	2491045
Mutex	4	200000000	12051355
Atomic	4	200000000	2649253
Mutex	6	200000000	12874399
Atomic	6	200000000	2503226
Mutex	8	200000000	14625334
Atomic	8	200000000	2715095
Mutex	2	300000000	20264242
Atomic	2	300000000	3921321
Mutex	4	300000000	17957042
Atomic	4	300000000	3876310
Mutex	6	300000000	19427079
Atomic	6	300000000	3758121
Mutex	8	300000000	21596561
Atomic	8	300000000	3767432
Mutex	2	400000000	25911491
Atomic	2	400000000	4687163
Mutex	4	400000000	23314971
Atomic	4	400000000	5521513
Mutex	6	400000000	25799418
Atomic	6	400000000	5074029
Mutex	8	400000000	28611857
Atomic	8	400000000	4949739
Mutex	2	500000000	31198788
Atomic	2	500000000	6323243
Mutex	4	500000000	29506067
Atomic	4	500000000	6911749
Mutex	6	500000000	32831775
Atomic	6	500000000	6560770
Mutex	8	500000000	35415572
Atomic	8	500000000	6307727

Άσκηση 3 (Pthreads)

3.1. Περιγραφή προβλήματος

Η υλοποίηση αυτή βασίζεται στην άσκηση 2, με τη διαφορά ότι κάθε νήμα διαθέτει τη δική του θέση σε έναν κοινό πίνακα και σε κάθε επανάληψη αυξάνει απευθείας το στοιχείο που του αντιστοιχεί.

3.2. Περιγραφή Λύσης

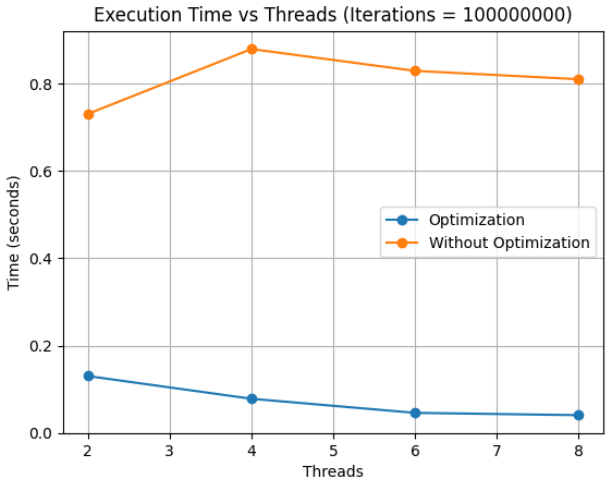
Αρχικά, οι επαναλήψεις κατανέμονται στα νήματα, τα οποία στη συνέχεια δημιουργούνται και εκτελούν αυτή τη βασική λειτουργία, δηλαδή την προσαύξηση του αντίστοιχου κελιού στον πίνακα. Δεν χρησιμοποιήθηκε κανένα μέσο συγχρονισμού (π.χ. mutex ή atomic) για την προστασία των μετρήσεων, διότι κάθε νήμα γράφει αποκλειστικά σε ξεχωριστό στοιχείο του πίνακα, έτσι αποφεύγονται τα data races χωρίς πρόσθετο κόστος συγχρονισμού.

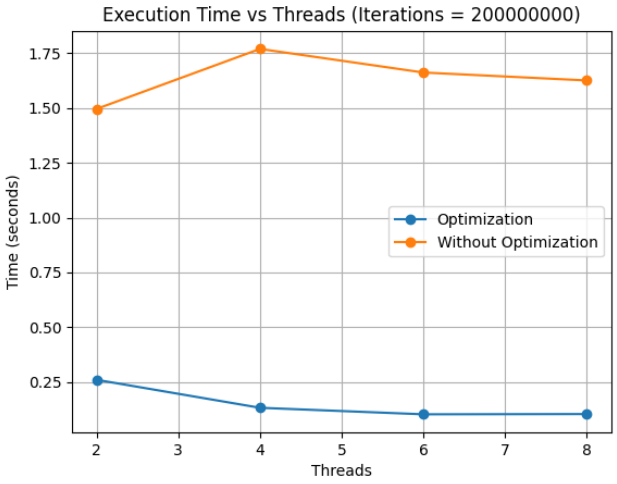
3.3. Πειραματικά Αποτελέσματα

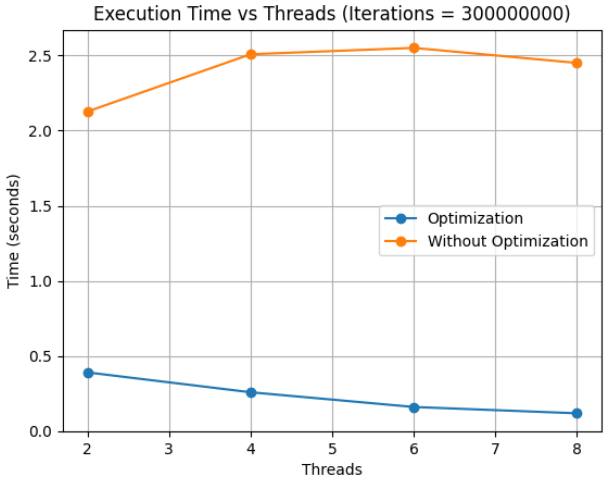
Είσοδος:

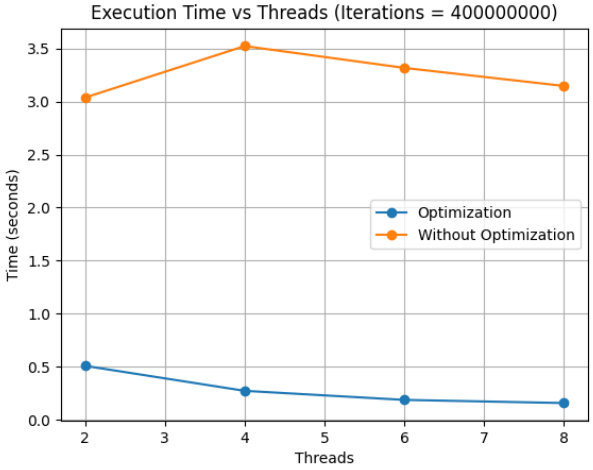
1. Αριθμός επαναλήψεων : από 10^8 έως 5*10^8 με βήμα 10^8
2. Νήματα: 2,4,6 και 8

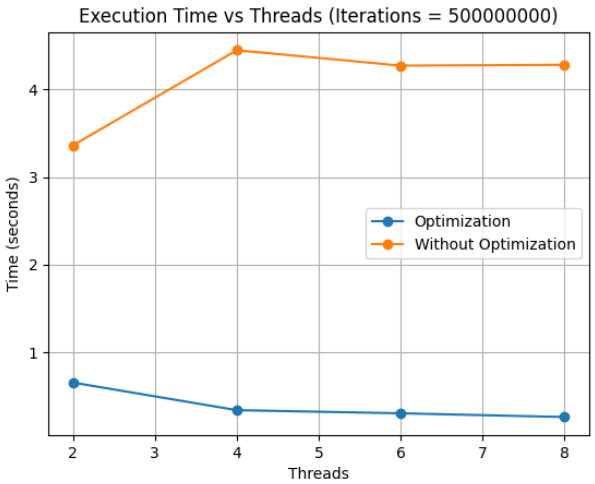
Παρατηρούμε ότι η απόδοση όχι μόνο δεν κλιμακώνεται γραμμικά με τα νήματα, αλλά χειροτερεύει έως το πλήθος των φυσικών πυρήνων (4 νήματα), όπου ο χρόνος εκτέλεσης φτάνει στο μέγιστό του. Παρά την προσθήκη ακόμη περισσότερων νημάτων (6, 8), η μείωση του χρόνου είναι ελάχιστη και δεν αντισταθμίζει την υπερφόρτωση του δίαυλο επικοινωνίας με τη μνήμη από τις συνεχείς, ταυτόχρονες εγγραφές. Επιπλέον, όσο αυξάνεται ο αριθμός επαναλήψεων, επιβαρύνεται η συμφόρηση, με αποτέλεσμα ακόμα και σε 8 νήματα το πρόγραμμα να μην ξεπερνά σε απόδοση τα 2 νήματα. Αυτό καταδεικνύει ότι η συνεχής, επαναλαμβανόμενη εγγραφή στα ίδια κελιά μνήμης επιβαρύνει περισσότερο το σύστημα απ’ ό,τι βοηθάει στη μείωση του χρόνου.











Στη βελτιστοποιημένη εκδοχή, όπου παρουσιάζεται με μπλε χρώμα, κάθε νήμα, αντί να εκτελεί μια εγγραφή στη μνήμη σε κάθε επανάληψη, συσσωρεύει το άθροισμα σε μια τοπική μεταβλητή και μόνο όταν ολοκληρωθούν όλες οι επαναλήψεις γράφει μία φορά το τελικό αποτέλεσμα στο αντίστοιχο στοιχείο του πίνακα. Έτσι ελαχιστοποιούνται οι συνεχείς, ταυτόχρονες εγγραφές μνήμης, μειώνεται ο φόρτος κυκλοφορίας στον δίαυλο

επικοινωνίας με τη μνήμη και η επιβάρυνση από ψευδο-κοινοχρησία δεδομένων, και επιτυγχάνεται σημαντικά καλύτερη κλιμάκωση.

lock	Threads	Iterations	Time
Without Optimization	2	100000000	730920
Optimization	2	100000000	130241
Without Optimization	4	100000000	879756
Optimization	4	100000000	78131
Without Optimization	6	100000000	829786
Optimization	6	100000000	45813
Without Optimization	8	100000000	810584
Optimization	8	100000000	40513
Without Optimization	2	200000000	1494882
Optimization	2	200000000	261197
Without Optimization	4	200000000	1768266
Optimization	4	200000000	133694
Without Optimization	6	200000000	1660713
Optimization	6	200000000	103943
Without Optimization	8	200000000	1624579
Optimization	8	200000000	105088
Without Optimization	2	300000000	2126891
Optimization	2	300000000	390136
Without Optimization	4	300000000	2508507
Optimization	4	300000000	257897
Without Optimization	6	300000000	2550641
Optimization	6	300000000	159945
Without Optimization	8	300000000	2450654
Optimization	8	300000000	117736
Without Optimization	2	400000000	3036809
Optimization	2	400000000	508912
Without Optimization	4	400000000	3522784
Optimization	4	400000000	273463
Without Optimization	6	400000000	3316865
Optimization	6	400000000	188133
Without Optimization	8	400000000	3146454
Optimization	8	400000000	158357
Without Optimization	2	500000000	3360746
Optimization	2	500000000	658467
Without Optimization	4	500000000	4444842
Optimization	4	500000000	343910
Without Optimization	6	500000000	4269447
Optimization	6	500000000	308165
Without Optimization	8	500000000	4278586
Optimization	8	500000000	265718

Άσκηση 4 (Pthreads)

4.1. Περιγραφή προβλήματος

Πρόκειται για μια υλοποίηση όπου συγκρίνει τρεις διαφορετικές εκδοχές κατασκευής φράγματος (barrier), όπου επιτυγχάνεται ο συγχρονισμός των νημάτων σε ένα σημείο. Η πρώτη είναι μία έτοιμη υλοποίηση από την βιβλιοθήκη Pthreads, η δεύτερη με χρήση κλειδώματος και μεταβλητή συνθήκης (condition variable) και τέλος η τρίτη όπου υλοποιείται με αναμονή σε εγρήγορση και είναι εφικτή η επαναχρησιμοποίηση του φράγματος. Εδώ συγκρίνουμε ποια από αυτές τις τρείς τεχνικές είναι η καλύτερη ως προς των χρόνο σε υπεράριθμο πλήθος επαναλήψεων.

4.2. Περιγραφή Λύσης

Έτοιμη υλοποίηση Pthreads (pthreadBarrier.c): Χρησιμοποιήθηκε η έτοιμη λειτουργία φράγματος που παρέχει η βιβλιοθήκη Pthreads. Το pthread_barrier_wait() χρησιμοποιείται απευθείας ώστε κάθε νήμα να περιμένει μέχρι να φτάσουν όλα τα υπόλοιπα νήματα στο ίδιο σημείο πριν συνεχίσουν. Ο συγχρονισμός είναι απλός και αποτελεσματικός, καθώς χρησιμοποιεί την ενσωματωμένη λειτουργικότητα της βιβλιοθήκης. Στην αρχή ο barrier αρχικοποιείται με την pthread_barrier_init() για να περιμένει μέχρι να φτάσουν στο σημείο αναμονής όλα τα νήματα, πριν προχωρήσουν παρακάτω. Μετά την ολοκλήρωση των νημάτων, ο barrier καταστρέφεται με την pthread_barrier_destroy().

Χρήση κλειδώματος και μεταβλητής συνθήκης (condBarrier.c): Χρησιμοποιήθηκε ο δοσμένος κώδικας του βιβλίου. Εδώ με τη χρήση ενός mutex (barrier_mutex) και μιας μεταβλητής συνθήκης (ok_to_proceed), κάθε νήμα αυξάνει έναν μετρητή (barrier_thread_cnt) όταν φτάσει στο φράγμα. Με την condition variable αν το νήμα είναι το τελευταίο δηλαδή ο μετρητής την συγκεκριμένη στιγμή είναι ίσος με το πλήθος των νημάτων, επαναφέρει τον μετρητή στο μηδέν και ξυπνά όλα τα άλλα νήματα μέσω της συνάρτησης pthread_cond_broadcast(), τα οποία ήταν σε κατάσταση αναστολής, όπου με το συμβάν ξεμπλοκαρίστηκαν.

Φράγμα με spin-wait (spinBarrier.c): Υλοποιήθηκε ο ψευδοκώδικας *Centralized Barrier with Sense Reversal* του βιβλίου «Parallel Computer Architecture: A Hardware/Software Approach» σελ.332[1]. Η δομή αυτού του φράγματος έχει έναν κεντρικό μετρητή (barrier_count) και ένα flag. Χρησιμοποιήθηκε η τεχνική "sense reversal" για να επιτευχθεί η επαναχρησιμοποιήσιμη λειτουργικότητα. Κάθε νήμα αυξάνει τον κοινό μετρητή, και αν είναι το τελευταίο νήμα που φτάνει στο φράγμα, ανανεώνει το flag ώστε να συνεχίσουν τα υπόλοιπα νήματα που περιμένουν σε έναν ενεργό βρόχο (busy wait).

- Γιατί το sense-reversal centralized barrier, αν και βασίζεται σε αναμονή σε εγρήγορση) είναι επαναχρησιμοποιούμενο?

Το βασικό πρόβλημα προκύπτει όταν ένα νήμα κολλήσει (π.χ., λόγω διακοπής από το λειτουργικό σύστημα) στην αναμονή της μεταβλητής flag (spin loop). Κατά τη διάρκεια αυτή, τα υπόλοιπα νήματα μπορεί να συνεχίσουν, να εκτελέσουν τον υπολογισμό τους και να φτάσουν σε ένα επόμενο φράγμα, όπου ένα άλλο νήμα επανεκκινεί τον μετρητή και το flag (αλλάζοντας την τιμή του). Το μπλοκαρισμένο νήμα, όταν επιστρέψει, περιμένει ακόμη την παλιά τιμή της flag για να συνεχίσει, αλλά η flag έχει ήδη αλλάξει, και πλέον δεν θα πάρει ποτέ την αναμενόμενη τιμή (καθώς απαιτεί να φτάσουν ξανά όλα τα νήματα στο φράγμα για να ξαναλλάξει η τιμή της). Έτσι δημιουργείται ένα deadlock.

Η λύση στο πρόβλημα γίνεται με τη χρήση της τεχνικής sense reversal. Σε αυτή τη μέθοδο, κάθε νήμα διατηρεί μια τοπική μεταβλητή (local_sense) που καθορίζει ποια τιμή της κοινής μεταβλητής flag θα περιμένει (είτε 0 είτε 1). Όταν το νήμα φτάνει στο φράγμα, αντιστρέφει την τιμή του local_sense και περιμένει η flag να αποκτήσει αυτήν την τιμή. Το τελευταίο νήμα που φτάνει στο φράγμα επαναφέρει τον μετρητή και αλλάζει την τιμή της flag ώστε να ταιριάζει με αυτή που περιμένουν όλα τα νήματα. Έτσι, τα νήματα που περιμένουν σε αναμονή ενεργής κατάστασης (spin loop) θα δουν την επιθυμητή τιμή και θα προχωρήσουν. Με αυτόν τον τρόπο αποφεύγεται το πρόβλημα της πρόωρης επαναφοράς της flag, που θα μπορούσε να οδηγήσει σε αδιέξοδο αν κάποιο νήμα είχε καθυστερήσει. Κάθε νήμα περιμένει τη σωστή, εναλλασσόμενη τιμή και η αλλαγή της flag γίνεται μόνο όταν όλα τα νήματα έχουν φτάσει στο φράγμα, εξασφαλίζοντας την ασφάλεια και επαναχρησιμοποίηση της υλοποίησης.

[1] https://www.iqytechnicalcollege.com/Parallel%20Computer%20Architecture.pdf

4.3. Πειραματικά Αποτελέσματα

Είσοδος:

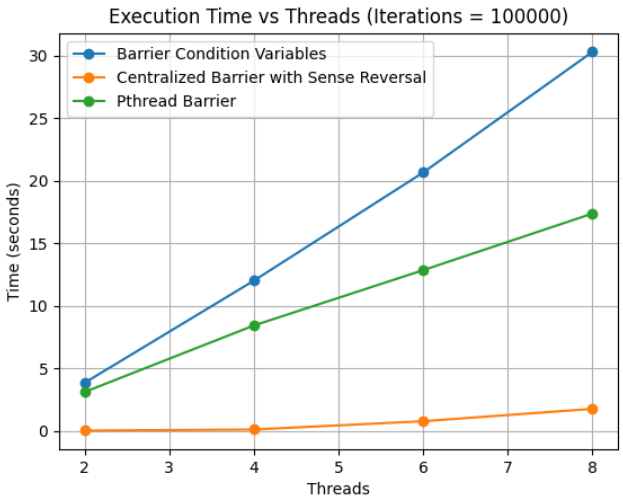
1. Αριθμός επαναλήψεων : από 10^8 έως 5*10^8 με βήμα 10^8
2. Νήματα: 2,4,6 και 8

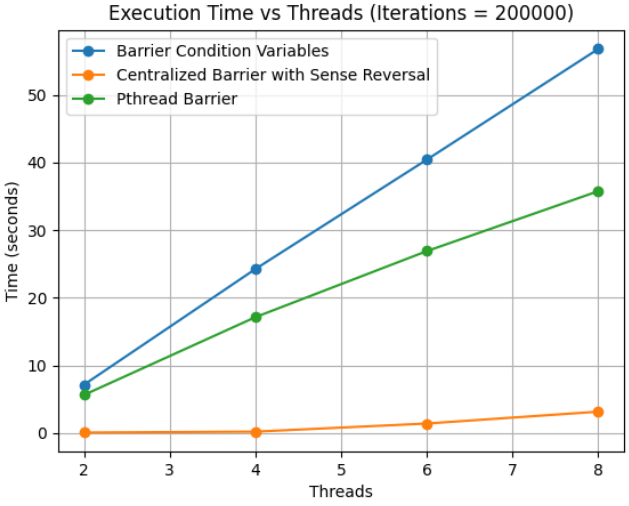
Στις γραφικές παραστάσεις παρατηρείται ξεκάθαρα ότι ο χρόνος εκτέλεσης αυξάνεται τόσο με τον αριθμό των νημάτων όσο και με τον αριθμό των επαναλήψεων. Ωστόσο, οι τρεις διαφορετικές υλοποιήσεις barrier παρουσιάζουν πολύ διαφορετική συμπεριφορά όσον αφορά την απόδοσή τους.

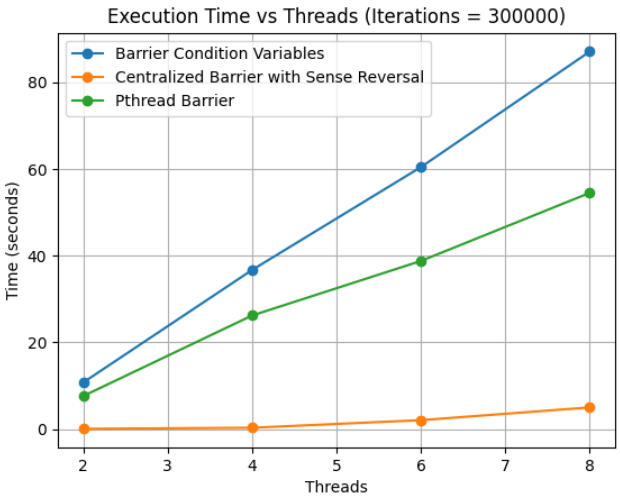
Η υλοποίηση με condition variables (μπλε γραμμή) είναι σταθερά η πιο αργή. Καθώς αυξάνονται τα threads, ο χρόνος εκτέλεσης αυξάνεται απότομα, ιδιαίτερα όταν οι επαναλήψεις ξεπερνούν τις 300.000. Αυτό υποδηλώνει υψηλό κόστος συγχρονισμού: κάθε νήμα πρέπει να περιμένει στη μεταβλητή κατάστασης και να αφυπνιστεί με broadcast από το τελευταίο νήμα. Αν κάποιο νήμα δεν έχει ακόμα φτάσει στη `pthread_cond_wait` όταν έρθει το broadcast, ενδέχεται να μείνει μπλοκαρισμένο. Γι' αυτό και αυτή η υλοποίηση είναι ευάλωτη σε προβλήματα χρονισμού.

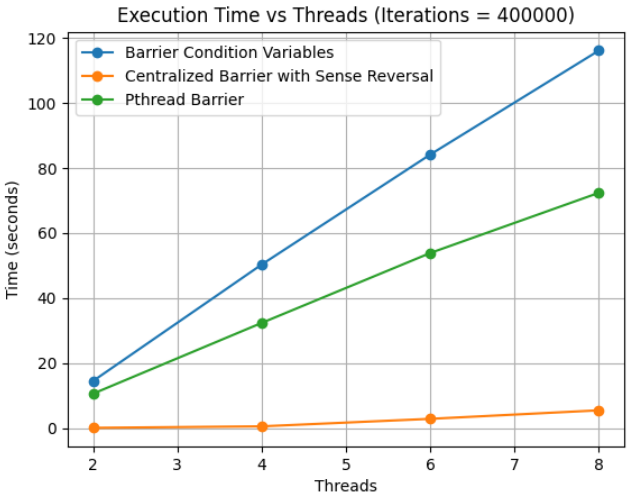
Αντίθετα, το centralized barrier με sense reversal (πορτοκαλί γραμμή) είναι εντυπωσιακά αποδοτικό. Ο χρόνος εκτέλεσης παραμένει χαμηλός, ακόμα και με πολλά threads και μεγάλο αριθμό επαναλήψεων. Αυτή η υλοποίηση χρησιμοποιεί spinlock και busy-waiting με μία κοινή σημαία flag, κάτι που φαίνεται να λειτουργεί αποδοτικά στο πείραμα.

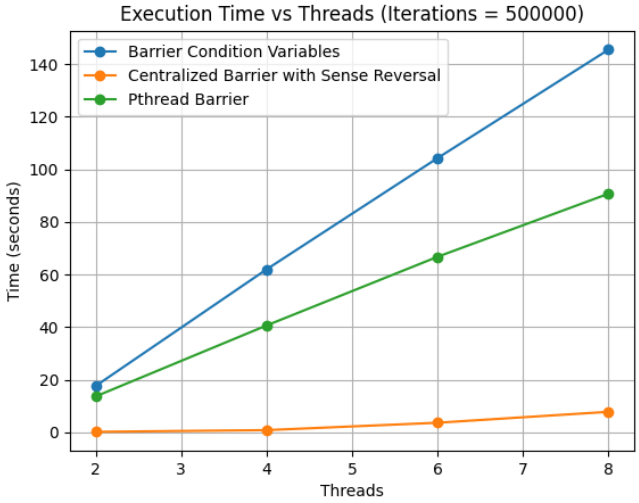
Η τρίτη υλοποίηση, με το pthread barrier (πράσινη γραμμή), αποτελεί μία ισορροπημένη προσέγγιση. Δεν είναι τόσο γρήγορη όσο το spinlock, αλλά έχει πολύ σταθερή και προβλέψιμη συμπεριφορά. Ο χρόνος εκτέλεσης αυξάνεται γραμμικά και ομαλά.

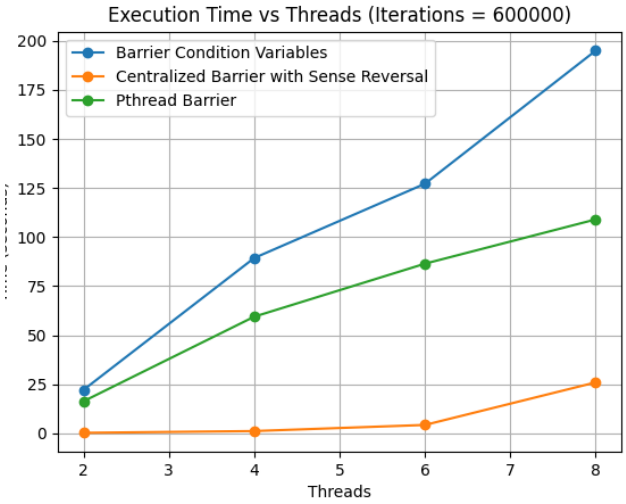


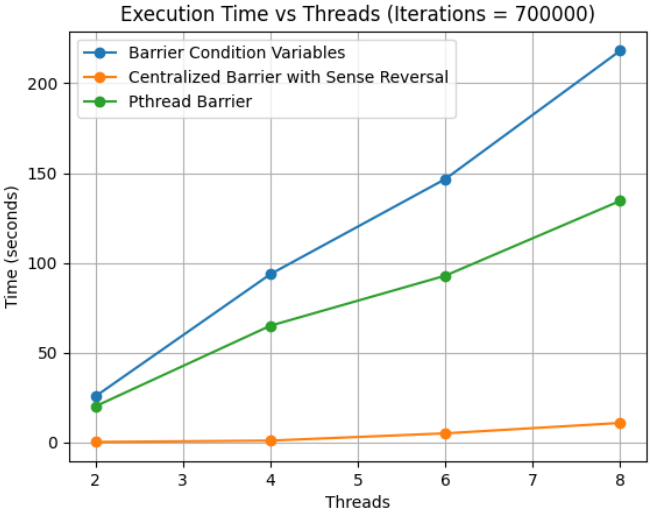


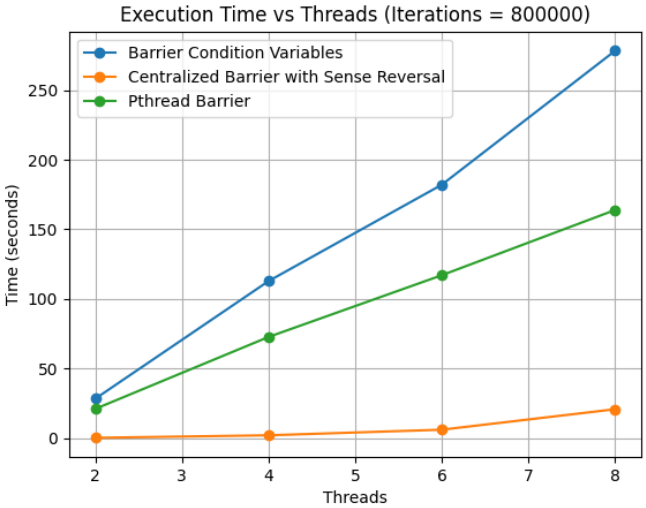

Execution Time vs Threads (Iterations = 300000)
Barrier Condition Variables
Centralized Barrier with Sense Reversal
Pthread Barrier
Time (seconds)
Threads

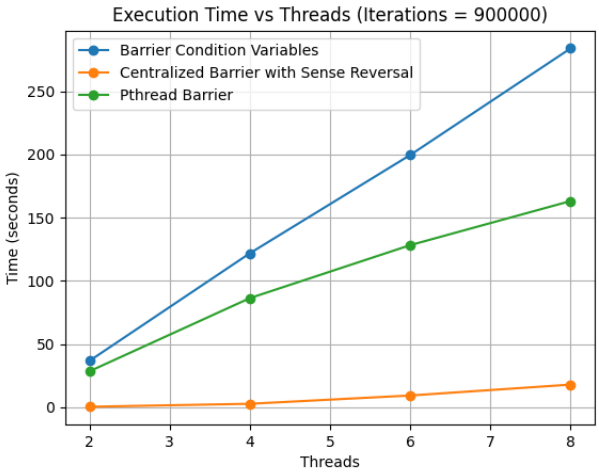

Execution Time vs Threads (Iterations = 400000)
Barrier Condition Variables
Centralized Barrier with Sense Reversal
Pthread Barrier
Time (seconds)
Threads

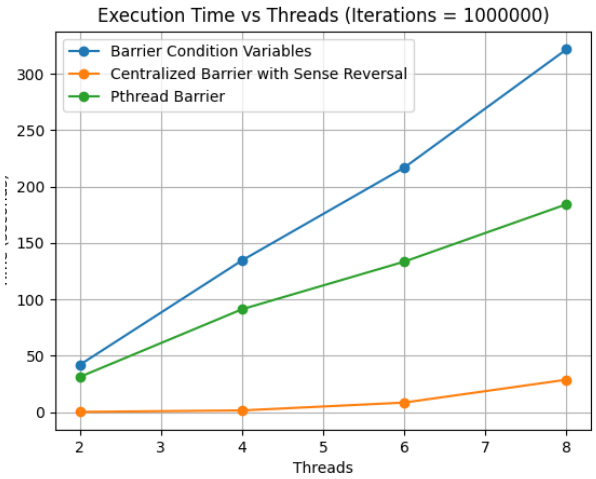

Execution Time vs Threads (Iterations = 500000)
Barrier Condition Variables
Centralized Barrier with Sense Reversal
Pthread Barrier
Time (seconds)
Threads


Execution Time vs Threads (Iterations = 600000)
Barrier Condition Variables
Centralized Barrier with Sense Reversal
Pthread Barrier
Threads

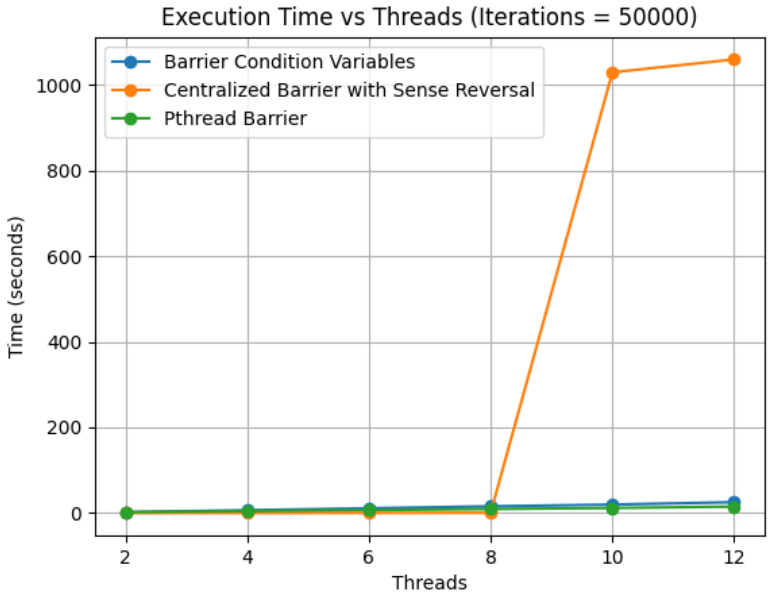

Execution Time vs Threads (Iterations = 700000)
Barrier Condition Variables
Centralized Barrier with Sense Reversal
Pthread Barrier
Time (seconds)
Threads


Execution Time vs Threads (Iterations = 800000)
Barrier Condition Variables
Centralized Barrier with Sense Reversal
Pthread Barrier
Time (seconds)
Threads





- Τι συμβαίνει όταν ο αριθμός των νημάτων είναι μεγαλύτερος από τον αριθμό των πυρήνων;



Το γράφημα που απεικονίζει τον χρόνο εκτέλεσης για 50.000 επαναλήψεις με διαφορετικό αριθμό νημάτων (2,4,6,8,10,12) δείχνει μια χαρακτηριστική συμπεριφορά υπερφόρτωσης του συστήματος. Συγκεκριμένα, παρατηρείται αρχικά μείωση του χρόνου εκτέλεσης όσο αυξάνεται ο αριθμός των νημάτων, κάτι που οφείλεται στην καλύτερη εκμετάλλευση των διαθέσιμων πυρήνων του επεξεργαστή (μέχρι και 8, που είναι το όριο των φυσικών πυρήνων στο σύστημα). Ωστόσο, μόλις ο αριθμός των νημάτων ξεπεράσει τους 8, εμφανίζεται αύξηση του χρόνου εκτέλεσης. Αυτό οφείλεται στο γεγονός ότι τα επιπλέον νήματα δεν μπορούν να εκτελούνται ταυτόχρονα, αλλά εναλλάσσονται στους ίδιους πυρήνες μέσω μηχανισμών χρονοπρογραμματισμού του λειτουργικού συστήματος

(context switching). Η συχνή εναλλαγή μεταξύ νημάτων αυξάνει το υπολογιστικό κόστος και προκαλεί καθυστερήσεις, ειδικά όταν τα νήματα πρέπει να συγχρονίζονται μέσω μηχανισμών όπως τα barriers. Συνεπώς, η απόδοση βελτιώνεται μόνο μέχρι το σημείο που ο αριθμός των νημάτων ισούται με τον αριθμό των πυρήνων, ενώ πέρα από αυτό το σημείο η προσθήκη περισσότερων νημάτων οδηγεί σε μείωση της συνολικής αποδοτικότητας του προγράμματος.

Άσκηση 5 (OpenMP)

5.1. Περιγραφή προβλήματος

Στην παρούσα άσκηση ζητείται η τροποποίηση του προγράμματος **πολλαπλασιασμού μήτρας-διανύσματος** omp_mat_vect_rand_split.c, ώστε να εκτελεί **αποδοτικά** τον πολλαπλασιασμό **άνω τριγωνικού πίνακα** με διάνυσμα, με χρήση της βιβλιοθήκης **OpenMP**. Ο στόχος είναι να **αποφευχθούν οι περιττοί υπολογισμοί** που προκύπτουν από την επεξεργασία μηδενικών στοιχείων κάτω από την κύρια διαγώνιο, καθώς σε έναν άνω τριγωνικό πίνακα όλα τα στοιχεία κάτω από την κύρια διαγώνιο είναι μηδενικά και δεν συμβάλλουν στο αποτέλεσμα.

5.2. Περιγραφή Λύσης

Καταρχάς, τροποποιήθηκε η λογική του πολλαπλασιασμού της μήτρας με το διάνυσμα, ώστε να αποφεύγονται οι περιττοί υπολογισμοί στα στοιχεία κάτω από την κύρια διαγώνιο. Συγκεκριμένα, αντί να υπολογίζονται όλα τα στοιχεία της γραμμής j=0 έως n, ο εσωτερικός βρόχος ξεκινά πλέον από j=i, δηλαδή από τη διαγώνιο και πάνω. Με αυτόν τον τρόπο εκμεταλλευόμαστε τη δομή του άνω τριγωνικού πίνακα, παραλείποντας πράξεις που αφορούν διπλότυπα στοιχεία και συνεπώς δεν συμβάλλουν στο αποτέλεσμα.

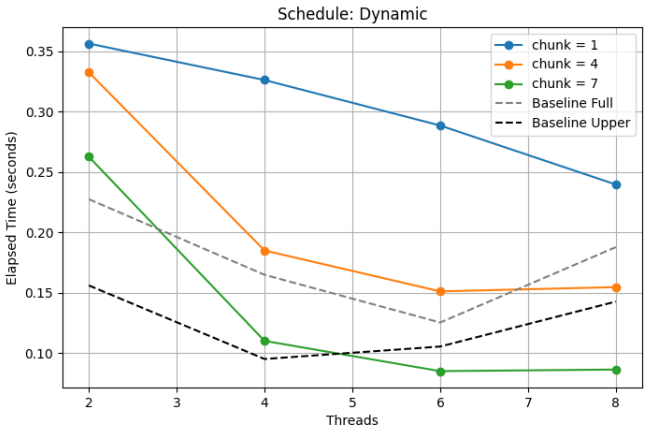
Για περαιτέρω βελτίωση της απόδοσης, προστέθηκε η δυνατότητα χρήσης της πολιτικής schedule(runtime) της OpenMP. Αυτή επιτρέπει την επιλογή της στρατηγικής κατανομής των επαναλήψεων κατά το χρόνο εκτέλεσης, μέσω της μεταβλητής περιβάλλοντος OMP_SCHEDULE. Με τον τρόπο αυτό, πειραματιζόμαστε με διαφορετικά schedule (static, dynamic, guided) και διαφορετικά μεγέθη chunk. Η χρήση schedule(runtime) και της μεταβλητής περιβάλλοντος OMP_SCHEDULE επιτρέπει τη δοκιμή διαφορετικών επιλογών.

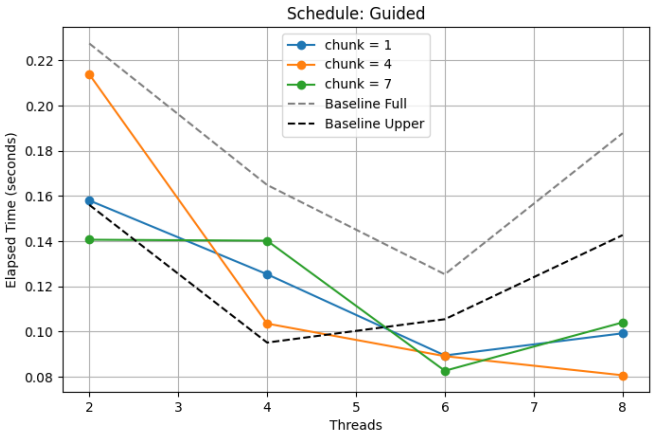
5.3. Πειραματικά Αποτελέσματα

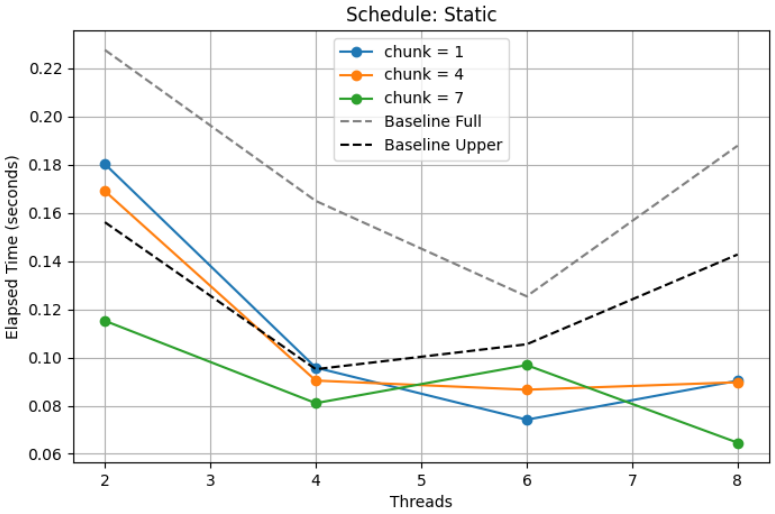
Είσοδος:

1. Νήματα: 2, 4, 6, και 8
2. schedules: Static, Dynamic, Guided
3. chunks:1, 4, 7
4. Matrix dimensions: 10000 × 10000
5. Matrix types:
   - full: Full matrix multiplication (no optimization)
   - upper: Multiplication of only the upper triangular matrix (with upper optimization)

Με την τροποποίηση της υλοποίησης ώστε να λαμβάνει υπόψη ότι ο πίνακας είναι **άνω τριγωνικός**, περιορίζεται το εσωτερικό loop από j=0 έως n, σε j=i έως n. Αυτή η βελτιστοποίηση, ακόμη και χωρίς περαιτέρω παράλληλη βελτίωση, οδηγεί σε μικρότερο υπολογιστικό φόρτο και επομένως ταχύτερη εκτέλεση. Ωστόσο, όταν εφαρμόζεται και παραλληλοποίηση μέσω OpenMP, η απόδοση επηρεάζεται σημαντικά από την επιλογή πολιτικής κατανομής επαναλήψεων. Τα πειραματικά αποτελέσματα δείχνουν ότι η καλύτερη απόδοση επιτυγχάνεται με χρήση του dynamic ή static scheduling και κατάλληλα επιλεγμένο chunk size (συγκεκριμένα το chunk = 7), ενώ η στρατηγική guided προσφέρει μια αξιόπιστη και σταθερή εναλλακτική.







Threads	Type	Schedule	Chunk	Time
2	Schedule	Static	1	180220,6
2	Schedule	Static	4	168999,8
2	Schedule	Static	7	115198,3
2	Schedule	Dynamic	1	356553,6
2	Schedule	Dynamic	4	332701,8
2	Schedule	Dynamic	7	262716,1
2	Schedule	Guided	1	158043,8
2	Schedule	Guided	4	213892,6
2	Schedule	Guided	7	140664,7
2	Baseline	Full	-	227552,3
2	Baseline	Upper	-	156089,7
4	Schedule	Static	1	95768,07
4	Schedule	Static	4	90402,95
4	Schedule	Static	7	81039,63
4	Schedule	Dynamic	1	326313,7
4	Schedule	Dynamic	4	185041,9
4	Schedule	Dynamic	7	110133,7
4	Schedule	Guided	1	125373,1
4	Schedule	Guided	4	103518,9
4	Schedule	Guided	7	140229,9
4	Baseline	Full	-	164918,7
4	Baseline	Upper	-	95116,22
6	Schedule	Static	1	74174,88
6	Schedule	Static	4	86622,01
6	Schedule	Static	7	96826,21
6	Schedule	Dynamic	1	288615,1
6	Schedule	Dynamic	4	151180,1
6	Schedule	Dynamic	7	85054,61
6	Schedule	Guided	1	89402,2

6	Schedule	Guided	4	89104,02
6	Schedule	Guided	7	82624,18
6	Baseline	Full	-	125354,8
6	Baseline	Upper	-	105453,5
8	Schedule	Static	1	90342,87
8	Schedule	Static	4	89693,26
8	Schedule	Static	7	64649,53
8	Schedule	Dynamic	1	239729,2
8	Schedule	Dynamic	4	154680,2
8	Schedule	Dynamic	7	86367,49
8	Schedule	Guided	1	99211,61
8	Schedule	Guided	4	80634,85
8	Schedule	Guided	7	104037,4
8	Baseline	Full	-	187825,4
8	Baseline	Upper	-	142696,8

Άσκηση 6 (OpenMP)

6.1. Περιγραφή προβλήματος

Πρόκειται για την προσομοίωση του Παιχνιδιού της Ζωής του John Conway, με δύο εκδοχές **1)** Σειριακή υλοποίηση (serial_game_of_life) και **2)** Παράλληλη υλοποίηση με OpenMP (parallel_game_of_life). Η κύρια λειτουργία του προγράμματος είναι να προσομοιώσει για έναν αριθμό γενεών την εξέλιξη ενός τετραγωνικού πίνακα (πλέγματος) διαστάσεων $n \times n$, στον οποίο κάθε κελί είναι είτε ζωντανό (1) είτε νεκρό (0), με βάση κάποιους απλούς κανόνες γειτνίασης.

Οι βασικοί κανόνες του παιχνιδιού:

Για κάθε κελί:

- Ζωντανό με <2 ή >3 γείτονες → πεθαίνει.
- Νεκρό με ακριβώς 3 γείτονες → γίνεται ζωντανό.
- Σε κάθε άλλη περίπτωση → διατηρεί την κατάστασή του.

6.2. Περιγραφή Λύσης

Η serial_game_of_life είναι η **σειριακή** εκδοχή της προσομοίωσης του Παιχνιδιού της Ζωής. Δέχεται ως ορίσματα τον αριθμό των γενεών, το μέγεθος του πλέγματος n, και δύο πίνακες: grid (τρέχουσα γενιά) και next_grid (επόμενη γενιά). Η συνάρτηση εκτελεί έναν εξωτερικό βρόχο για κάθε γενιά. Στο εσωτερικό, δύο φωλιασμένοι βρόχοι (for) σαρώνουν όλα τα κελιά του πλέγματος. Για κάθε κελί (i, j) καλείται η συνάρτηση count_neighbors, η οποία επιστρέφει τον αριθμό ζωντανών γειτόνων.

Ανάλογα με αυτόν τον αριθμό, εφαρμόζονται οι κανόνες του Conway:

- Αν το κελί είναι **ζωντανό** (grid[i][j] == 1) και έχει **λιγότερους από 2 ή περισσότερους από 3** γείτονες, τότε πεθαίνει στο next_grid.
- Αν είναι **νεκρό** και έχει **ακριβώς 3** ζωντανούς γείτονες, τότε ζωντανεύει στο next_grid.
- Σε κάθε άλλη περίπτωση, διατηρεί την ίδια τιμή στο next_grid.

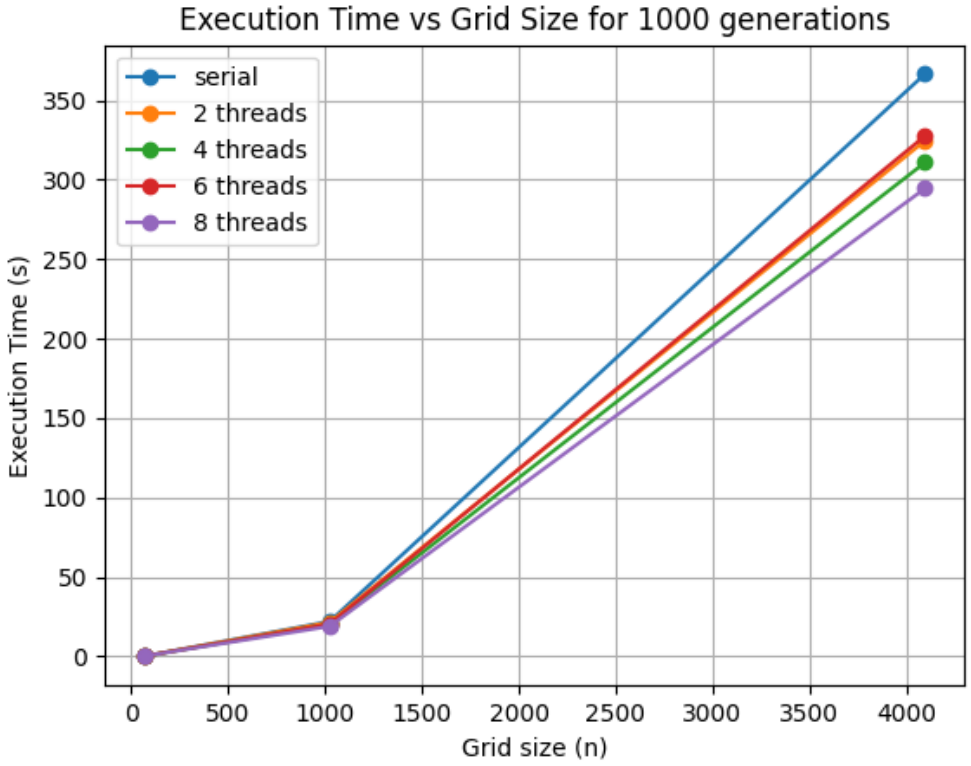
Μετά την επεξεργασία όλων των κελιών για την παρούσα γενιά, γίνεται αντιγραφή του next_grid πίσω στο grid, χρησιμοποιώντας memcpy, ώστε να προετοιμαστεί η επόμενη γενιά.

Η **παράλληλη** εκδοχή με χρήση OpenMP. Η λογική είναι παρόμοια με τη σειριακή, αλλά η επεξεργασία των κελιών γίνεται παράλληλα με χρήση #pragma omp parallel for collapse(2). Αυτό επιτρέπει την ταυτόχρονη εκτέλεση πολλών γραμμών και στηλών από διαφορετικά threads. Η πρώτη φάση υπολογίζει και ενημερώνει τον πίνακα next_grid, ενώ η δεύτερη φάση αντιγράφει το next_grid στο grid. Έτσι κάθε γενιά προχωρά γρήγορα με παράλληλη υπολογιστική ισχύ.

6.3. Πειραματικά Αποτελέσματα

Είσοδος:

1. αριθμός γενιών: 1000
2. μέγεθος του πλέγματος: 64, 1024, 4096
3. σειριακή ή ο παράλληλη εκτέλεση
4. νήματα: 2, 4, 6 και 8



Το διάγραμμα παρουσιάζει τον χρόνο εκτέλεσης σε συνάρτηση με το μέγεθος του πλέγματος για 1000 γενεές, συγκρίνοντας τη σειριακή και την παράλληλη υλοποίηση του Game of Life με διαφορετικό αριθμό νημάτων (2, 4, 6 και 8). Παρατηρούμε ότι ο χρόνος αυξάνεται πολύ όσο μεγαλώνει το πλέγμα, γεγονός που είναι αναμενόμενο, καθώς το πρόβλημα είναι υπολογιστικά εντατικό με πολυπλοκότητα O(n^2). Η σειριακή εκδοχή έχει σταθερά τον μεγαλύτερο χρόνο εκτέλεσης, ειδικά για μεγάλες διαστάσεις, κάτι που αναδεικνύει την ανάγκη για παραλληλοποίηση. Οι παράλληλες εκδοχές εμφανίζουν σημαντικά μικρότερους χρόνους, με την απόδοση να βελτιώνεται όσο αυξάνεται ο αριθμός των νημάτων. Ωστόσο, η επιτάχυνση δεν είναι γραμμική, δηλαδή δεν διπλασιάζεται ο ρυθμός όταν διπλασιάζεται ο αριθμός των νημάτων. Αυτό πιθανότατα οφείλεται στο κόστος συγχρονισμού των tasks, στην περιορισμένη μεταφορά μνήμης (memory bandwidth) και σε πιθανή ανισοκατανομή του φορτίου.

n	threads	generations	time
64	serial	1000	100687
64	2	1000	125878
64	4	1000	111878

64	6	1000	176613
64	8	1000	175769
1024	serial	1000	22095231
1024	2	1000	21523464
1024	4	1000	20195892
1024	6	1000	20431730
1024	8	1000	19023498
4096	serial	1000	366623445
4096	2	1000	324597934
4096	4	1000	310891668
4096	6	1000	327608454
4096	8	1000	294569740

Άσκηση 7 (OpenMP)

7.1. Περιγραφή προβλήματος

Η υλοποίηση αυτή αφορά το γνωστό πρόβλημα της επίλυσης ενός συστήματος γραμμικών εξισώσεων της μορφής Ax=b. Συγκεκριμένα, επιλέγεται η μέθοδος απαλοιφής Gauss, που αποτελεί μια από τις πιο γνωστές μεθόδους για την επίλυση τέτοιων συστημάτων. Σε αυτό το πρόβλημα καλούμαστε να το υλοποιήσουμε για τις εκδοχές σειριακή και παράλληλη δομή.

Η σειριακή υλοποίηση που παρέχεται είναι άμεση και απλή. Το πρόβλημα επιλύεται διαδοχικά είτε γραμμή-γραμμή είτε στήλη-στήλη, ξεκινώντας από την τελευταία γραμμή (ή στήλη) και υπολογίζοντας κάθε μεταβλητή του αγνώστου διανύσματος x. Η σειριακή έκδοση είναι ιδιαίτερα χρήσιμη για μικρού και μεσαίου μεγέθους συστήματα ή για σύγκριση με την παράλληλη εκδοχή, προκειμένου να εκτιμηθεί η επιτάχυνση που προσφέρει η παραλληλοποίηση.

Η επιλογή παραλληλοποίησης προκύπτει από το γεγονός ότι η διαδικασία απαλοιφής Gauss μπορεί να αξιοποιήσει την ανεξαρτησία πολλών πράξεων, ειδικά κατά τη διάρκεια της οπισθοδρομικής αντικατάστασης.

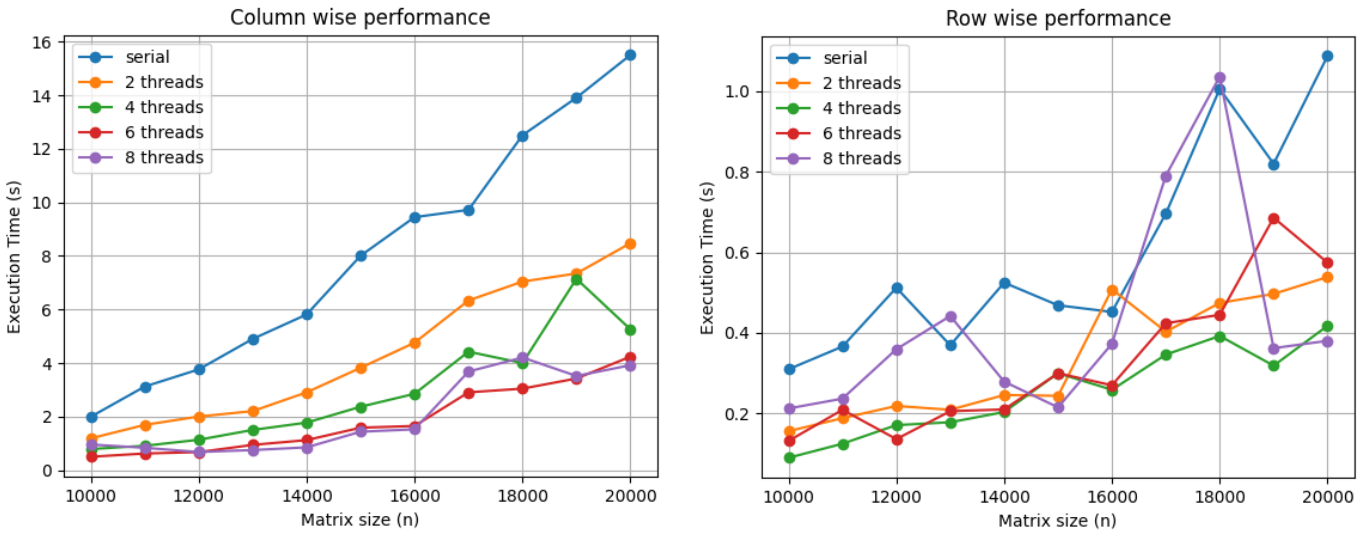
7.2. Περιγραφή Λύσης

Η παραλληλοποίηση έχει εφαρμοστεί τόσο στη γραμμή-γραμμή (row-wise) όσο και στη στήλη-στήλη (column-wise) μέθοδο. Στην περίπτωση της row-wise παραλληλοποίησης, εκτελούνται παράλληλα οι πράξεις υπολογισμού του αθροίσματος των γινομένων που αφορούν την κάθε γραμμή, χρησιμοποιώντας αναγωγή (reduction). Στην περίπτωση της column-wise παραλληλοποίησης, οι πράξεις ενημέρωσης του διανύσματος x σε κάθε στήλη εκτελούνται ανεξάρτητα και παράλληλα. Με αυτόν τον τρόπο αξιοποιούνται καλύτερα οι σύγχρονοι πολυπύρηνοι επεξεργαστές, παρέχοντας μεγαλύτερη απόδοση συγκριτικά με την αντίστοιχη σειριακή υλοποίηση.

7.3. Πειραματικά Αποτελέσματα

Είσοδος:

1. μέγεθος διάστασης: από 10^4 έως 2*10^4 με νήμα 10^3
2. σειριακή ή ο παράλληλη εκτέλεση
3. σειρά-σειρά ή στήλη-στήλη
4. νήματα: 2, 4, 6 και 8



Τα αποτελέσματα των δύο γραφημάτων απεικονίζουν την απόδοση των σειριακών και παράλληλων υλοποιήσεων της μεθόδου οπισθοδρομικής αντικατάστασης Gauss, τόσο με στήλη-στήλη (column-wise) όσο και με γραμμή-γραμμή (row-wise) προσέγγιση. Η αξιολόγηση γίνεται για διαφορετικά μεγέθη πίνακα και για διάφορους αριθμούς νημάτων

(2, 4, 6, 8), προκειμένου να διαπιστωθεί η αποδοτικότητα της παραλληλοποίησης σε σχέση με τη σειριακή εκτέλεση.

Στο πρώτο γράφημα, που αφορά την column-wise υλοποίηση, παρατηρείται ότι η σειριακή εκδοχή έχει σαφώς μεγαλύτερους χρόνους εκτέλεσης, οι οποίοι αυξάνονται σχεδόν γραμμικά με το μέγεθος του πίνακα. Η παράλληλη εκτέλεση επιτυγχάνει σταθερή και σημαντική επιτάχυνση, ιδιαίτερα όταν χρησιμοποιούνται 6 ή 8 νήματα, υποδεικνύοντας ότι η συγκεκριμένη μέθοδος επωφελείται ιδιαίτερα από την παραλληλοποίηση. Αυτό οφείλεται στην ανεξαρτησία των πράξεων κατά τη διαδικασία ενημέρωσης των στοιχείων του διανύσματος λύσης, γεγονός που επιτρέπει υψηλό βαθμό ταυτόχρονης εκτέλεσης.

Αντίθετα, στο δεύτερο γράφημα που αφορά την row-wise υλοποίηση, παρατηρείται ότι ο συνολικός χρόνος εκτέλεσης είναι πολύ μικρότερος, ακόμα και για τη σειριακή εκδοχή. Αυτό δείχνει ότι η row-wise μέθοδος παρουσιάζει καλύτερη χωρική τοπικότητα στη μνήμη, με αποτέλεσμα ταχύτερη εκτέλεση. Η παράλληλη υλοποίηση βελτιώνει αρχικά την απόδοση, όμως η αύξηση των νημάτων πέραν των 4 οδηγεί σε αστάθεια και διακυμάνσεις στους χρόνους εκτέλεσης. Αυτό οφείλεται πιθανότατα στο κόστος συγχρονισμού και στη μικρότερη δυνατότητα παραλληλισμού της συγκεκριμένης προσέγγισης.

n	threads	row_or_col	time
10000	serial	row wise	309495
10000	serial	column wise	1990759
10000	2	row wise	155722
10000	2	column wise	1184525
10000	4	row wise	89688
10000	4	column wise	785291
10000	6	row wise	131551
10000	6	column wise	497950
10000	8	row wise	212096
10000	8	column wise	955193
.....	.....	.....	.....
19000	serial	row wise	819635
19000	serial	column wise	13912403
19000	2	row wise	496564
19000	2	column wise	7343933

19000	4	row wise	318349
19000	4	column wise	7146194
19000	6	row wise	685713
19000	6	column wise	3420728
19000	8	row wise	361782
19000	8	column wise	3516740
20000	serial	row wise	1087326
20000	serial	column wise	15506968
20000	2	row wise	538483
20000	2	column wise	8477437
20000	4	row wise	417422
20000	4	column wise	5257859
20000	6	row wise	574027
20000	6	column wise	4237337
20000	8	row wise	380151
20000	8	column wise	3917893

Άσκηση 8 (OpenMP)

8.1. Περιγραφή προβλήματος

Η άσκηση έχει ως σκοπό την παραλληλοποίηση του αλγόριθμου ταξινόμησης με συγχώνευση (mergesort), και συγκεκριμένα της εκδοχής από πάνω προς τα κάτω (top-down recursive). Για την παραλληλοποίηση αξιοποιείται το OpenMP και η οδηγία #pragma omp task, που επιτρέπει την εκτέλεση κομματιών του κώδικα ως ανεξάρτητες εργασίες (tasks) σε διαφορετικά νήματα (threads).

8.2. Περιγραφή Λύσης

Ο αλγόριθμος ταξινόμησης καλείται ανάλογα με την επιλογή του χρήστη. Η **σειριακή έκδοση (serial_mergeSort)** καλεί αναδρομικά τη συνάρτηση και συγχωνεύει τα αποτελέσματα με την merge. Η **παράλληλη έκδοση (parallel_mergeSort)** κάνει το ίδιο,

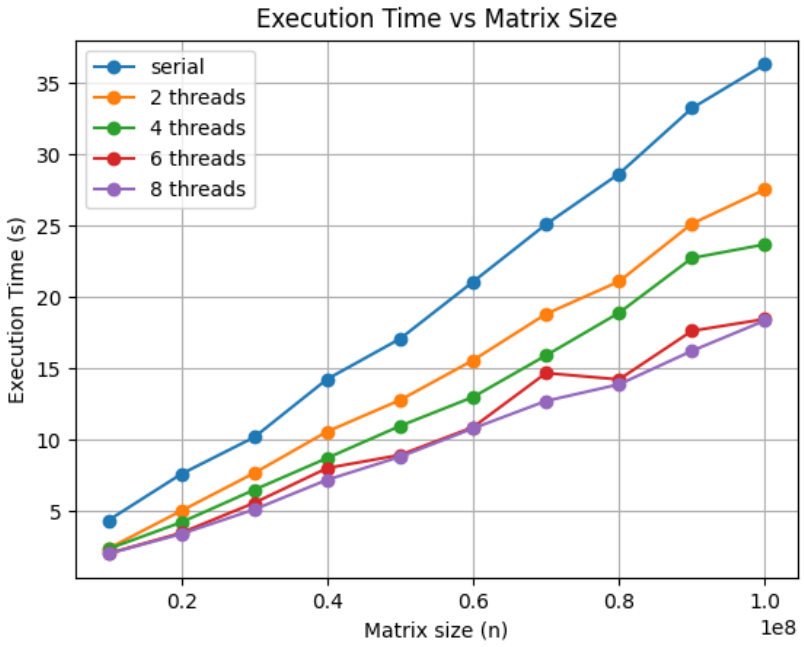
αλλά χρησιμοποιεί #pragma omp task για να αναθέσει κάθε υποεργασία ταξινόμησης σε νέο task. Για να αποφευχθεί υπερβολική δημιουργία tasks σε μικρούς πίνακες (που θα μείωνε την απόδοση), χρησιμοποιείται η συνθήκη if (right - left > 1000000) έτσι ώστε να δημιουργούνται tasks μόνο για αρκετά μεγάλους υποπίνακες.

8.3. Πειραματικά Αποτελέσματα

Είσοδος:

1. μέγεθος πίνακα: από 10^7 έως 10^8 με νήμα 10^7
2. σειριακή ή ο παράλληλη εκτέλεση
3. νήματα: 2,4,6,8

Το γράφημα δείχνει τον **χρόνο εκτέλεσης** του αλγορίθμου mergesort σε σχέση με το **μέγεθος του πίνακα** για διαφορετικό αριθμό νημάτων (threads). Παρατηρούμε ότι όσο μεγαλώνει το μέγεθος του πίνακα, ο χρόνος εκτέλεσης αυξάνεται για όλες τις περιπτώσεις, κάτι αναμενόμενο λόγω της υπολογιστικής πολυπλοκότητας. Η **σειριακή εκδοχή** (μπλε γραμμή) είναι σταθερά η πιο αργή, γεγονός που επιβεβαιώνει την αποτελεσματικότητα της παραλληλοποίησης.Όσο αυξάνεται ο αριθμός των νημάτων, παρατηρείται **μείωση στον χρόνο εκτέλεσης**, ιδίως για μεγάλα μεγέθη πίνακα. Τα 8 νήματα εμφανίζουν τον **χαμηλότερο χρόνο** για μεγάλους πίνακες, γεγονός που αναδεικνύει τη **θετική επίδραση του παραλληλισμού σε υπολογιστικά απαιτητικά προβλήματα**. Ωστόσο, πέρα από ένα σημείο (π.χ. από 6 σε 8 νήματα), η **βελτίωση μειώνεται**, κάτι που υποδεικνύει την ύπαρξη ορίων στην επιτάχυνση λόγω του **κόστους δημιουργίας και συγχρονισμού των tasks**.



n	threads	time
10000000	serial	4365701
10000000	2	2390866
10000000	4	2367178
10000000	6	2036317
10000000	8	2035644
20000000	serial	7611538
20000000	2	5036337
20000000	4	4230497
20000000	6	3493445
20000000	8	3412684
30000000	serial	10187732
30000000	2	7677263
30000000	4	6503612
30000000	6	5589498
30000000	8	5131269
40000000	serial	14258331
40000000	2	10584529
40000000	4	8728762
40000000	6	8026326
40000000	8	7206092
50000000	serial	17085472
50000000	2	12793472
50000000	4	10978886
50000000	6	8946389
50000000	8	8820505
60000000	serial	21074563
60000000	2	15579987
60000000	4	13002448
60000000	6	10891920
60000000	8	10806206
70000000	serial	25100909
70000000	2	18821286
70000000	4	15918889
70000000	6	14682999
70000000	8	12727526
80000000	serial	28633506
80000000	2	21094879
80000000	4	18897460
80000000	6	14233626
80000000	8	13887866
90000000	serial	33228642
90000000	2	25141295
90000000	4	22734369

90000000	6	17635590
90000000	8	16236592
100000000	serial	36296963
100000000	2	27537931
100000000	4	23690520
100000000	6	18460127
100000000	8	18358383

Άσκηση 9 (OpenMP)

9.1. Περιγραφή προβλήματος

Σε αυτή την άσκηση εξετάζεται η υλοποίηση **της παράλληλης εκτέλεσης** με χρήση **OpenMP tasks** αντί για parallel for. Η συνάρτηση serial_game_of_life είναι ακριβώς ίδια με την προηγούμενη: εφαρμόζει τους κανόνες του Conway σειριακά. Η συνάρτηση count_neighbors παραμένει ίδια και υπολογίζει τους ζωντανούς γείτονες ανά κελί. Η gameInitialization είναι και πάλι υπεύθυνη για την τυχαία αρχικοποίηση.

9.2. Περιγραφή Λύσης

Η νέα προσέγγιση στην parallel_game_of_life βασίζεται στον **καταμερισμό του πλέγματος σε υποπεριοχές (chunks)** και στη **χρήση της OpenMP task** για την παράλληλη εκτέλεση. Αντί να εκτελείται κάθε στοιχείο (i, j) του πίνακα ανεξάρτητα και παράλληλα, όπως στην collapse(2) εκδοχή, εδώ ολόκληρο το πλέγμα χωρίζεται σε μικρότερα ορθογώνια υποπλέγματα. Το μέγεθος κάθε υποπλέγματος εξαρτάται από τη μεταβλητή chunk_size, η οποία καθορίζεται δυναμικά με βάση την επιθυμητή πυκνότητα κατανομής (chunks_per_dim = 16). Για κάθε τέτοιο υποπλέγμα, δημιουργείται ένα **task** με τη διεύθυνση #pragma omp task, που καθοδηγεί το runtime του OpenMP να αναθέσει την επεξεργασία αυτής της περιοχής σε οποιοδήποτε διαθέσιμο thread. Αν η περιοχή είναι πολύ μικρή (κάτω από το όριο THRESHOLD), το task παραλείπεται για να αποφευχθεί το κόστος δημιουργίας πολλών μικρών tasks (λόγω υπερβολικής ανάγκης για συγχρονισμό και συχνών αλλαγών ενεργού νήματος). Αυτός ο έλεγχος εξασφαλίζει καλύτερη **ισορροπία υπολογιστικού φορτίου**.

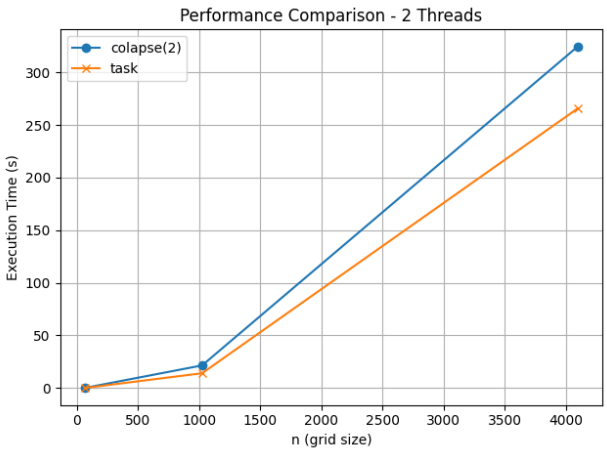
Στο εσωτερικό κάθε task, εκτελείται η ίδια λογική του Game of Life: υπολογισμός γειτόνων και ενημέρωση κατάστασης κελιών για το next_grid. Μόλις δημιουργηθούν όλα τα tasks για τη συγκεκριμένη γενιά, η εντολή #pragma omp taskwait διασφαλίζει ότι όλα

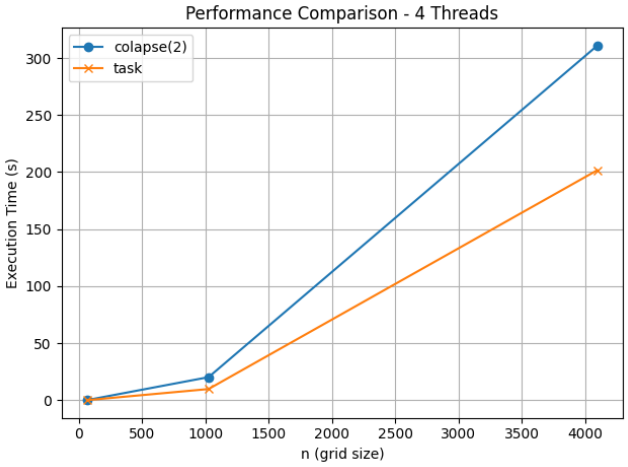
τα tasks έχουν ολοκληρωθεί πριν προχωρήσει στην επόμενη φάση. Τέλος, η αντιγραφή του next_grid στο grid γίνεται παράλληλα με #pragma omp parallel for, ώστε το πλέγμα να είναι έτοιμο για την επόμενη γενιά. Αυτή η προσέγγιση προσφέρει **δυναμική κατανομή εργασιών**, μειώνει τις ανάγκες για συντονισμό μεταξύ των νημάτων και προσαρμόζεται καλύτερα σε μεταβλητά μεγέθη πίνακα και διαθεσιμότητα threads, επιτυγχάνοντας έτσι πιο **αποδοτική χρήση των πόρων** του συστήματος.

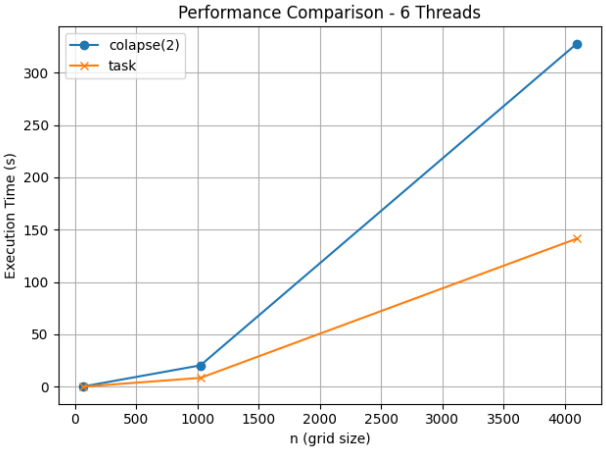
9.3. Πειραματικά Αποτελέσματα

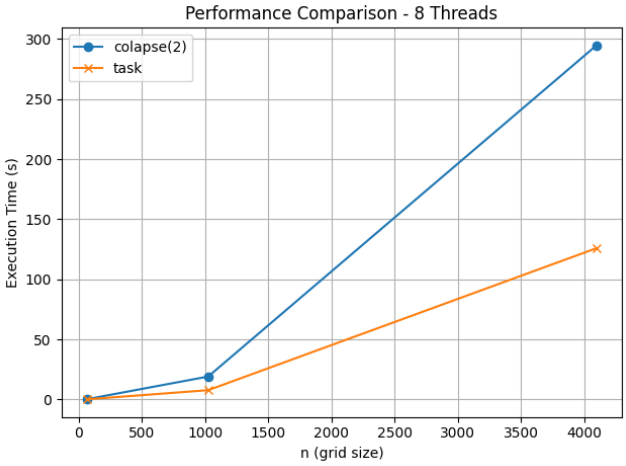
Είσοδος:

1. αριθμός γενιών: 1000
2. μέγεθος του πλέγματος: 64, 1024, 4096
3. σειριακή ή ο παράλληλη εκτέλεση
4. νήματα: 2, 4, 6 και 8









Τα γραφήματα συγκρίνουν τις δύο μεθόδους παραλληλοποίησης (collapse(2) και tasks) στο Παιχνίδι της ζωής, με διαφορετικό αριθμό νημάτων (2, 4, 6, 8). Παρατηρούμε ότι η μέθοδος των tasks είναι σταθερά πιο γρήγορη από τη μέθοδο collapse(2) σε όλα τα μεγέθη πλέγματος. Η διαφορά μεγαλώνει σημαντικά όσο αυξάνεται το μέγεθος του πλέγματος, λόγω του καλύτερου δυναμικού καταμερισμού φόρτου που προσφέρουν τα tasks. Αν και η αύξηση των νημάτων βελτιώνει τη συνολική απόδοση και των δύο μεθόδων, η προσέγγιση με tasks παραμένει πάντα αποδοτικότερη. Έτσι, η χρήση OpenMP tasks αποτελεί καλύτερη επιλογή για μεγάλα υπολογιστικά προβλήματα.

n	threads	generations	time
64	serial	1000	77433
64	2	1000	65538
64	4	1000	50851
64	6	1000	48601
64	8	1000	80155
1024	serial	1000	21022870
1024	2	1000	14094167
1024	4	1000	9789553
1024	6	1000	8456660
1024	8	1000	7673300
4096	serial	1000	375928736
4096	2	1000	265747052
4096	4	1000	201646769
4096	6	1000	141398395
4096	8	1000	125803472

Operating System: Windows/10

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 8

Thread(s) per core: 2

Core(s) per socket: 4

Socket(s): 1

Model name: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx

gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0