Raport Laboratori – Programim Paralel me OpenMP (C++)

Emër, Mbiemër: Adisa Hoxhaj

Tema: Testime të ekzekutimeve të kodeve në Programim Paralel OpenMP C++

2. Karakteristikat e kompjuterit të përdorur për ekzekutim:

- **Operating System:** Windows 11 Pro, Version 24H2, 64-bit operating system, x64-based processor
- **Processor (CPU):** AMD Ryzen 5 PRO 7540U w/ Radeon 740M Graphics
- **Memory (RAM):** 16 GB
- **Logical Processors :** 12
- **Cores :** 6

3. Metoda 1 - Llogaritja paralele e mesatares së numrave të rastësishëm me OpenMP

1. Përshkrimi Metodës

Metoda llogarit mesataren aritmetike të një vargu numrash të rastësishëm . Metoda teston se si ndryshon koha e ekzekutimit kur rrisim numrin e Threads (nga 1 deri në 12 dhe kur rrisim ngarkesën e punës (N nga 100 mijë në 100 milionë).

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

N është numri total i elementeve (përmasa e problemit).
x_i është numri i rastësishëm i gjeneruar në iteracionin i.

Për të vlerësuar fitimin nga paralelizimi, do të llogarisim **Speedup-in (S)** sipas Ligjit të Amdahl dhe **Efikasitetin** :

$$S = \frac{T_{serial}}{T_{paralel}} \qquad E = \frac{Speedup}{Nr_{Threads}}$$

Ku T_{serial} është koha e ekzekutimit me 1 Thread dhe $T_{paralel}$ është koha me P Threads.

Si funksionon kodi ?

1. **Faza Fillestare Programi nis ekzekutimin me një thread të vet#m.** Në këtë fazë, kodi rezervon memorien për tabelat, inicializon gjeneratorin e numrave të rastit (srand), dhe llogarit kohën e fillimit.Kompjuteri po punon vetëm me 1 bërthamë
2. **Faza e Degëzimit** Sapo ekzekutimi arrin te rreshti #pragma omp parallel, ndodh degëzimi.**Threadi kryesor** krijon një ekip threadsh të tjera .Numri i tyre varet nga sa kemi kërkuar ne .

3. **Ndarja e Punës** Kjo ndodh te cikli #pragma omp for. OpenMP e ndan automatikisht numrin total të iteracioneve (N) në blloqe të barabarta.Të gjithë threads punojnë në të njëjtën kohë, secili duke llogaritur shumën e pjesës së vet.
4. **Mbledhja e Rezultateve** Për shkak të komandës reduction(+:Sum), secili thread ka mbajtur një shumë private (lokale).Kur threads mbarojnë punën e tyre, ata presin te një vijë fundore. Sistemi merr të gjitha shumat private, i mbledh bashkë, dhe rezultatin ia shton variablit kryesor Sum.

Shpjegimi i Funksioneve

omp_set_num_threads(x): Cakton sa threads do punojnë.

omp_get_wtime(): Mat kohën reale (kronometër).

#pragma omp parallel for: Paralelizon ciklin for (ndan iteracionet).

reduction(+:Sum): Krijon kopje lokale te shumave te ndara dhe i mbledh në fund.

srand(time(0)): Siguron që numrat e rastit të jenë ndryshe çdo herë që hapim programin.

long long: Tip variabli për numra shumë të mëdhenj (mbi 2 miliardë), i sigurt për N=100 Milionë.

Madhesia (N)	Threads	Koha (sek)	Speedup (X)	Mesatarja
100000	1	0.00300	1.00	0.49973
100000	2	0.00200	1.50	0.49963
100000	4	0.00000	—-	0.50181
100000	6	0.00000	—-	0.50104
100000	12	0.00000	—-	0.50280
10000000	1	0.09400	1.00	0.50002
10000000	2	0.04700	2.00	0.50010
10000000	4	0.03200	2.94	0.50006
10000000	6	0.02000	4.70	0.50007
10000000	12	0.01400	6.71	0.50007
50000000	1	0.42700	1.00	0.50001
50000000	2	0.23800	1.79	0.50006
50000000	4	0.14400	2.97	0.50000
50000000	6	0.10300	4.15	0.50004
50000000	12	0.07600	5.62	0.50010
100000000	1	0.86500	1.00	0.49998
100000000	2	0.45300	1.91	0.50004
100000000	4	0.28200	3.07	0.50000
100000000	6	0.21900	3.95	0.50005
100000000	12	0.15700	5.51	0.50005

Tabela 1. Koha e ekzekutimit të metodës 1 për përmasa nga 100000-100 M dhe numri Thread 1,2,4,6,12

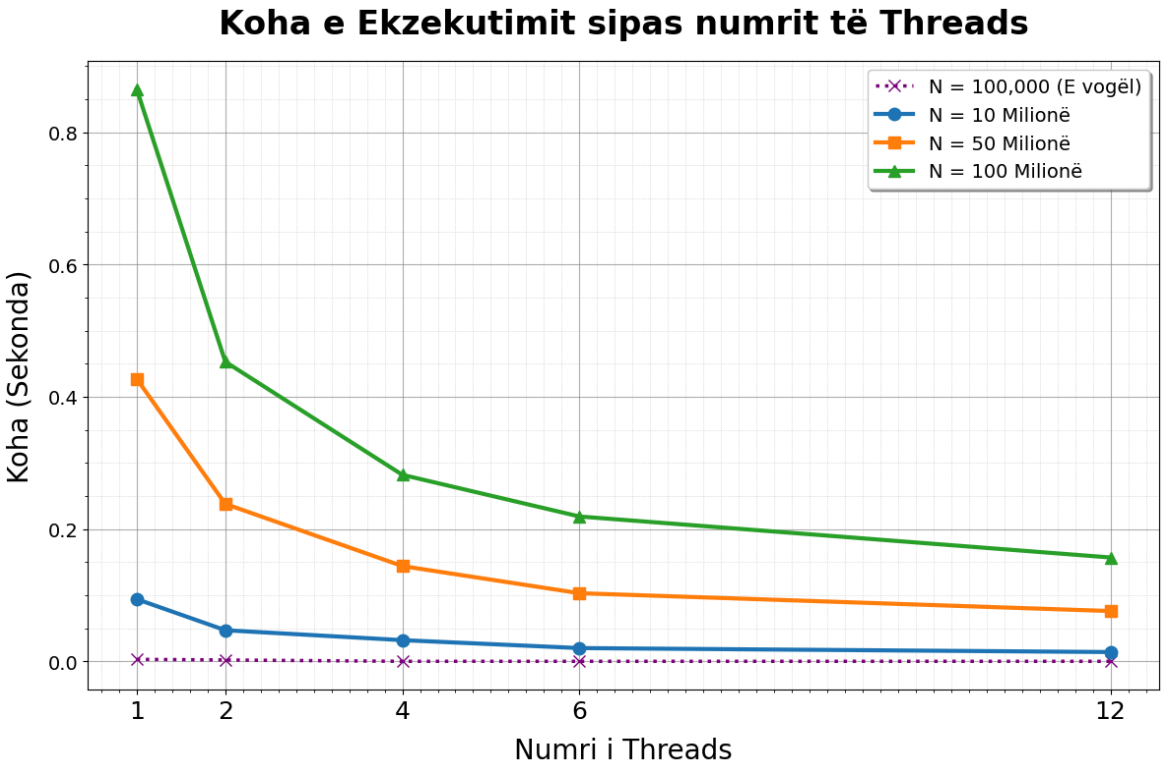


Figura 1 Grafiku i kohës së ekzekutimit sipas Threads per Metoden 1

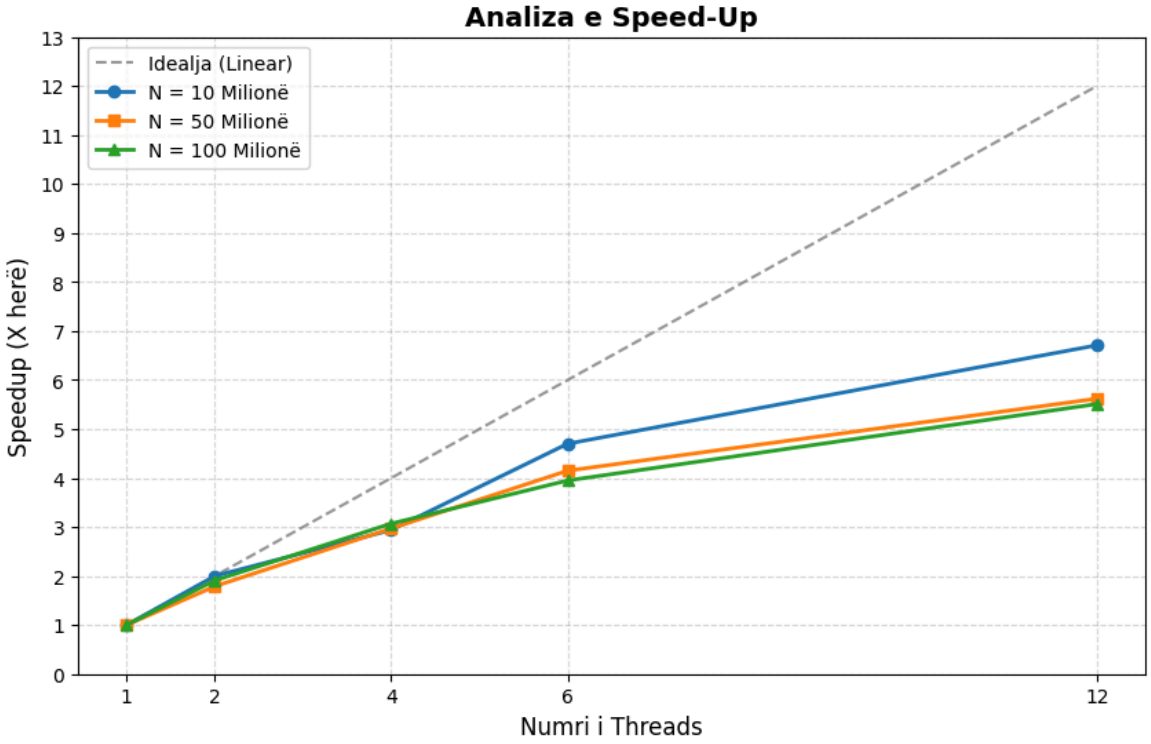


Figura 2 Grafiku i Analizës së Speed-Up për Metodën 1

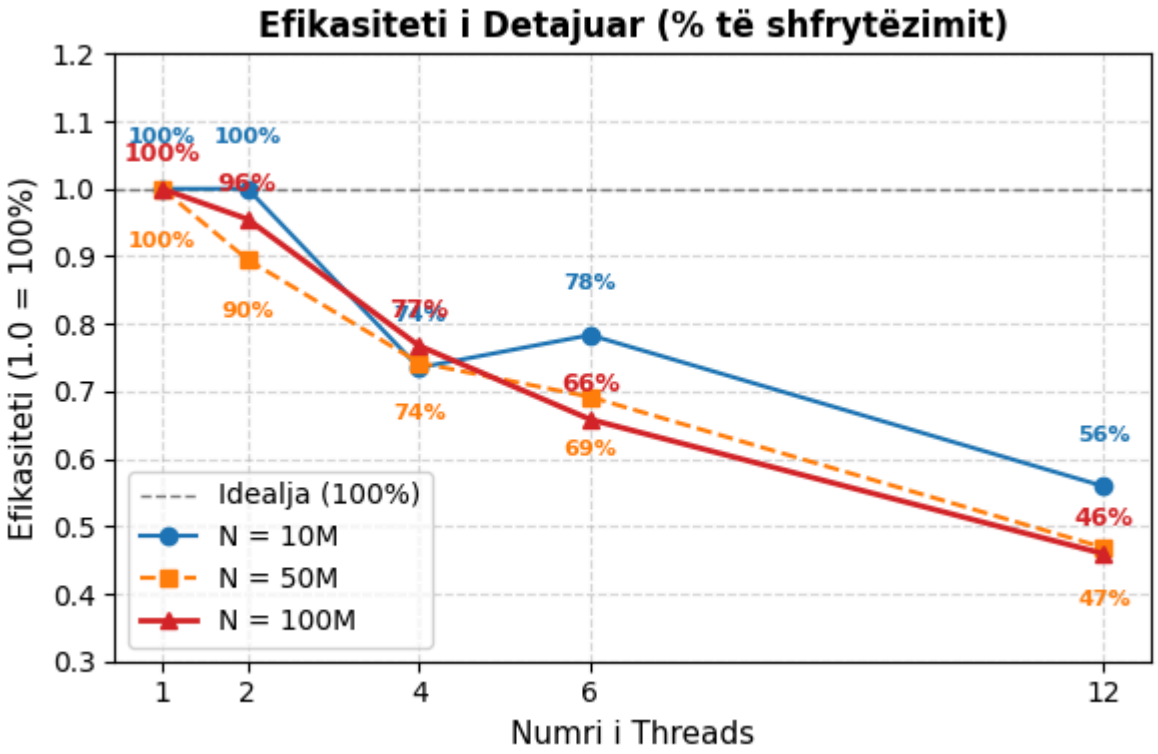


Figura 3 Grafiku i Efikasitetit për metodën 1

2. Përfundime

Në testime me ngarkesë të vogël, vumë re se paralelizimi **nuk është efikas**.Për probleme të vogla, koha që i duhet procesorit për të krijuar dhe menaxhuar threads (**Overhead**) është më e madhe se koha që fitohet nga ndarja e punës. Prandaj, për N të vogël, metoda seriale (1 Thread) është më e mira. Kur rritëm ngarkesën në 100 Milionë numra, paralelizimi tregoi potencialin e tij të plotë.Koha zbriti nga **0.865s** (1 Thread) në **0.157s** (12 Threads).Arritëm një **Speedup Maksimal prej 5.51x**. Kjo tregon se kodi po shfrytëzon me sukses fuqinë llogaritëse të procesorit për të përshpejtuar ndjeshëm kohën e zgjidhjes.Edhe pse përdorëm 12 Threads, shpejtësia u rrit me rreth 5.5 herë, jo 12 herë. Kjo ndodh për dy arsye kryesore teknike:**Bërthamat Fizike vs Logjike:** Laptopi ka **6 bërthama fizike**. 6 threads e parë japin rritjen më të madhe të performancës, Threads shtesë (nga 7 në 12) janë thjesht logjikë dhe ndajnë resurset e bërthamave fizike, ndaj shtojnë performancë, por jo dyfishim.

Analiza e Efikasitetit

Ndryshe nga algoritmet llogaritëse, këtu efikasiteti bie në mënyrë drastike me shtimin e procesorëve.Për rastin N=100M (vija e kuqe), efikasiteti fillon në **100%** me 1 Thread, bie në **77%** me 4 Threads, dhe përfundon në vetëm **46%** me 12 Threads. Kjo do të thotë se kur përdorim 12 bërthama, më shumë se gjysma e kapacitetit llogaritës (54%) shkon dëm . Procesorët nuk janë duke punuar, por janë duke "pritur".Grafiku tregon se pas **4 ose 6 Threads**, kurba e efikasitetit bie ndjeshëm nën 70%.Kjo sugjeron se për operacione kaq të thjeshta matematikisht, përdorimi i një numri të lartë threads (si 12) është i

panevojshëm dhe joefikas.

3. Shtojca 1

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <iomanip>

using namespace std;

int main() {

    long long N_values[] = { 100000, 10000000, 50000000, 100000000 };
    int total_N = 4;
       int threads_to_test[] = { 1, 2, 4, 6, 12 };    int total_T = 5;

    srand(time(0));
       cout << "STARTIMI I TESTEVE ME SPEEDUP (Analiza e Plote)" << endl;
       cout << setw(15) << "Madhesia (N)"
         << setw(10) << "Threads"
         << setw(15) << "Koha (sek)"
         << setw(15) << "Speedup (X)"
         << setw(15) << "Mesatarja" << endl;

    for (int i = 0; i < total_N; i++) {
        long long n = N_values[i];
               double t_serial = 0.0;

        for (int j = 0; j < total_T; j++) {
            int num_threads = threads_to_test[j];
            omp_set_num_threads(num_threads);
            double Sum = 0.0;
            double Aver = 0.0;
            double start_time = omp_get_wtime();
            #pragma omp parallel for reduction(+:Sum)
            for (long long k = 0; k < n; k++) {
                Sum += (double)rand() / RAND_MAX;
            }

            double end_time = omp_get_wtime();
            double time_taken = end_time - start_time;
            if (num_threads == 1) {
                t_serial = time_taken;
            }
                        double speedup = t_serial / time_taken;

            Aver = Sum / (double)n;
            cout << setw(15) << n
                 << setw(10) << num_threads
                 << setw(15) << fixed << setprecision(5) << time_taken
                 << setw(15) << setprecision(2) << speedup
                 << setw(15) << setprecision(5) << Aver << endl;
        }
            }
        int x; cin >> x;
    return 0;
}
```

4. Metoda 2 – Shuma e vektorëve

1. Përshkrimi Metodës

Kjo metodë ka për qëllim të krijojë dy vektorë me përmasa shumë te mëdha dhe t'i mbledhi element për element duke përdorur paralelizmin me OpenMP për të përshpejshtuar llogaritjet. Ky program llogarit shumën e dy vektorëve A dhe B dhe e ruan rezultatin në vektorin C

$$C[i] = A[i] + B[i].$$

Si funksionon kodi ?

Ndryshe nga ndarja e thjeshtë (statike) ku puna ndahet barabartë që në fillim, ky kod përdor schedule(dynamic, chunk).

Iteracionet e ciklit (nga 0 tek N) ndahen në pako të vogla të quajtura **"chunks"**. Threads-at marrin një chunk, e mbarojnë, dhe kthehen për të kërkuar chunkun tjetër.Nëse një thread është më i ngadaltë ose nëse disa iteracione kërkojnë më shumë kohë, threads-at e tjerë të shpejtë marrin më shumë chunks. Kjo siguron që askush të mos rrijë pushim.

Shpjegimi i Funksioneve

#include <omp.h>: Është biblioteka kryesore e OpenMP. Pa këtë, nuk funksionojnë komandat si omp_get_wtime() apo omp_set_num_threads().

#define CHUNKSIZE 1000: Përcakton madhësinë e chunk të punës. Kur themi schedule(dynamic, chunk), procesori nuk i merr iteracionet 1 nga 1, por merr 1000 iteracione njëherësh. Kjo rrit efikasitetin sepse zvogëlon kohën e komunikimit (overhead).

long long N_values[]: Përdorim long long (integers 64-bit) në vend të int sepse po punojmë me numra shumë të mëdhenj

delete[] a;: Në fund të çdo cikli, ne e lirojmë memorien

omp_set_num_threads(num_threads):Kjo komandë e "urdhëron" sistemin se sa *threads* duhet të përgatisë për zonën e ardhshme paralele. Ne e ndryshojmë këtë në çdo hap të ciklit (1, 2, 4, 6...).

omp_get_wtime(): Është "kronometri" i OpenMP.

#pragma omp parallel shared(a,b,c,chunk):Krijon grupin e threads.

shared: Tregon se vektorët a, b, c janë të përbashkët. Të gjithë threads punojnë mbi të njëjtën memorie RAM. Kjo kursen memorie.

#pragma omp for schedule(dynamic, chunk) nowait:

for: I tregon OpenMP që punën e ciklit for që vjen më poshtë, ta ndajë midis threads.

schedule(dynamic, chunk):Puna ndahet në blloqe me madhësi chunk (1000).Ndarja bëhet dinamike: Kush mbaron i pari, merr bllokun tjetër

Madhesia (N)	Threads	Koha (sek)	Speedup
10000000	1	0.02100	1.00x
10000000	2	0.01600	1.31x
10000000	4	0.01800	1.17x
10000000	6	0.01100	1.91x
10000000	8	0.00800	2.63x
10000000	12	0.01200	1.75x
50000000	1	0.14300	1.00x
50000000	2	0.06300	2.27x
50000000	4	0.06100	2.34x
50000000	6	0.04900	2.92x
50000000	8	0.04500	3.18x
50000000	12	0.04600	3.11x
100000000	1	0.28100	1.00x
100000000	2	0.14300	1.97x
100000000	4	0.11900	2.36x
100000000	6	0.09600	2.93x
100000000	8	0.08500	3.31x
100000000	12	0.08800	3.19x

Tabela 2. Koha e ekzekutimit dhe Speedup i metodës 2

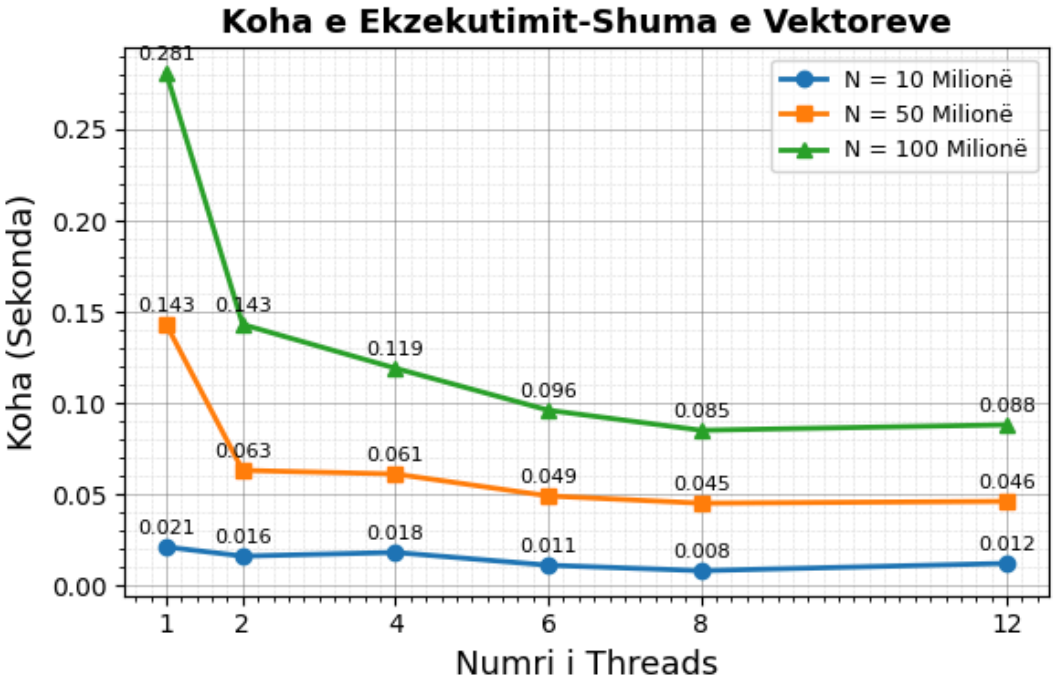


Figura 4 Grafiku i kohës se ekzekutimit për Metodën 2

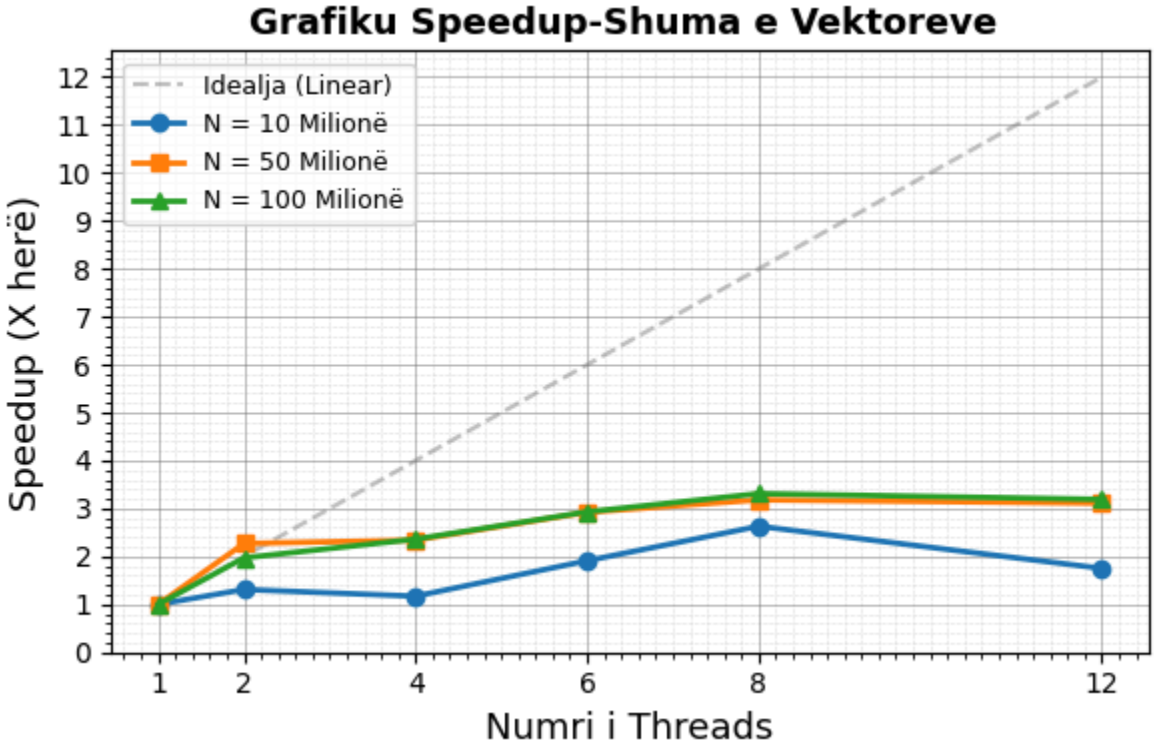


Figura 5 Grafiku i analizës së Speed-up për Metodë 2

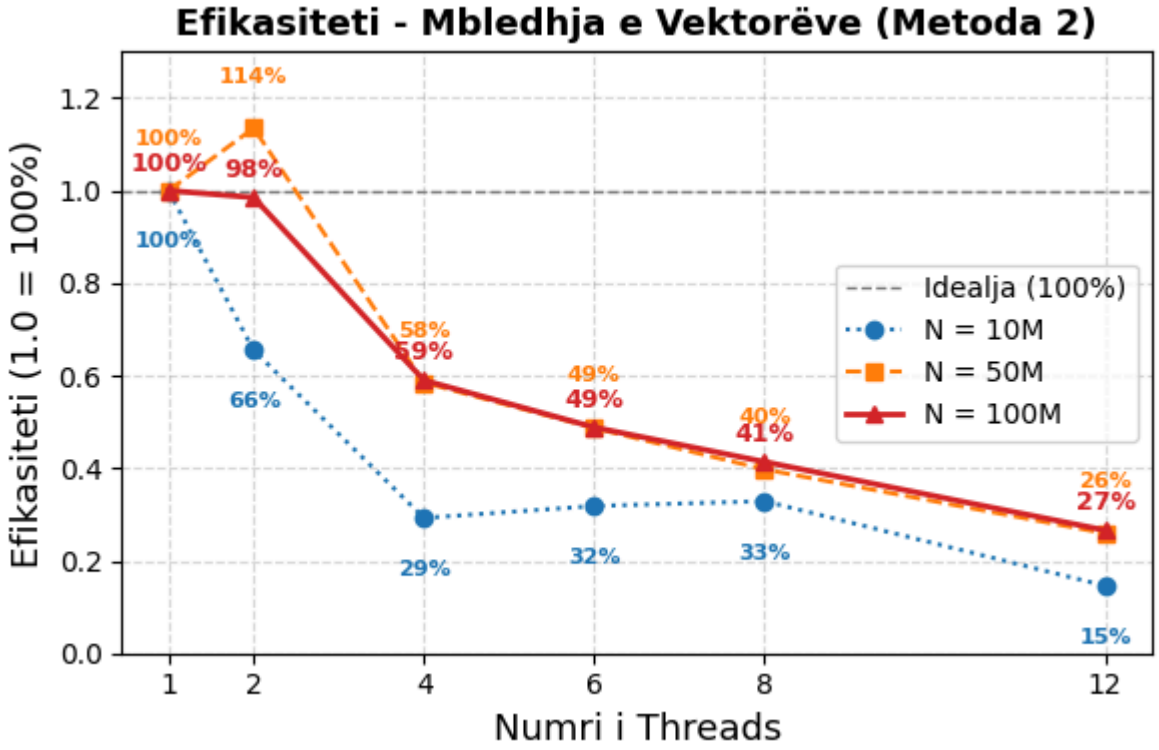


Figura 6 Grafiku i Analizës së Efikasitetit për Metodën 2

2. Përfundime

Për të gjitha madhësitë e testuara (N), vumë re se kalimi nga ekzekutimi serial (1 Thread) në paralel solli reduktim të kohës. Për rastin N=100 Milionë, koha zbriti nga **0.281s** në rreth **0.085s**, duke realizuar detyrën rreth 3 herë më shpejt.Për madhësi të vogla N=10 Milionë, koha e ekzekutimit me 12 threads ishte më e lartë se me 8 threads. Kjo tregon se për probleme të vogla, koha që i duhet sistemit për të menaxhuar dhe sinkronizuar 12 threads është më e madhe se përfitimi që marrim nga ndarja e punës.

Speedup-i maksimal i arritur ishte rreth **3.3x** (te N=100 Milionë), pavarësisht se përdorëm deri në 12 threads.Procesorët ishin të aftë të llogarisnin më shpejt, por duhej të prisnin që të dhënat të vinin nga memoria.Rezultatet treguan se efikasiteti maksimal arrihet me **8 Threads**, ku Speedup-i shënoi vlerën më të lartë (**3.31x**).Vumë re se Speedup-i ishte më i lartë dhe më i qëndrueshëm për N të mëdha. Për N=10M, Speedup-i maksimal ishte vetëm 2.6x, ndërsa për N=100M arriti në 3.3x, duke vërtetuar ligjin e Amdahl-it se paralelizmi shkëlqen në probleme me vëllim të madh pune.

Analiza e Efikasitetit

Metoda 2 tregon performancë të shkëlqyer me 2 Threads (mbi 100% efikasitet), por vuan nga ngopja e memories kur numri i threads rritet. Pika optimale e punës për këtë algoritëm është **2 deri në 4 Threads**; përtej këtij numri, shtimi i procesorëve është shpërdorim resursesh (efikasiteti bie në 26%).

3. Shtojca 2

```
#include <omp.h>
#include <iostream>
#include <iomanip>
#define CHUNKSIZE 1000
using namespace std;
int main() {
    long long N_values[] = { 10000000, 50000000, 100000000 };
    int total_N_tests = 3;
    int threads_to_test[] = { 1, 2, 4, 6, 8, 12 };
    int total_thread_tests = 6;
    int chunk = CHUNKSIZE;
    cout << "  TESTIMI I VEKTOREVE (Schedule Dynamic) - AUTOMATIK" << endl;
    cout << setw(15) << "Madhesia (N)"
         << setw(10) << "Threads"
         << setw(15) << "Koha (sek)"
         << setw(15) << "Speedup" << endl;
    for (int i = 0; i < total_N_tests; i++) {
        long long N = N_values[i];
        float *a = new float[N];
        float *b = new float[N];
        float *c = new float[N];
        #pragma omp parallel for schedule(static)
        for (long long k = 0; k < N; k++) {
            a[k] = k * 1.0f;
            b[k] = k * 2.0f;
        }
        double serial_time = 0.0;
        for (int j = 0; j < total_thread_tests; j++) {
```

```
            int num_threads = threads_to_test[j];
            omp_set_num_threads(num_threads);
            double start_time = omp_get_wtime();
            #pragma omp parallel shared(a,b,c,chunk)
            {
                #pragma omp for schedule(dynamic, chunk) nowait
                for (long long k = 0; k < N; k++) {
                    c[k] = a[k] + b[k];
                }
            }

            double end_time = omp_get_wtime();
            double time_taken = end_time - start_time;
            if (num_threads == 1) {
                serial_time = time_taken;
            }
            double speedup = 0.0;
            if(time_taken > 0.0000001) speedup = serial_time / time_taken;
            cout << setw(15) << N
                 << setw(10) << num_threads
                 << setw(15) << fixed << setprecision(5) << time_taken
                 << setw(14) << setprecision(2) << speedup << "x" << endl;
        }
        delete[] a;
        delete[] b;
        delete[] c;
    }
    cout << "Testimi perfundoi." << endl;
    int x; cin >> x;
    return 0;
}
```

5. Metoda 3 – Shumëzimi i dy matricave

1. Përshkrimi metodës

Shumëzimi i matricave është një operacion algoritmik me kompleksitet të lartë (O(N^3)), çka e bën atë kandidatin ideal për të demonstruar fuqinë e plotë të Paralelizmit.Kur jepen dy matrica katrore A dhe B me përmasa N x N matrica rezultante C llogaritet sipas formulës:

$$\mathbf{C[i][j]} = \sum_{k=0}^{N-1} (A[i][k] \; x \; B[k][j])$$

Si funksionon kodi?

OpenMP ndan **ciklin e jashtëm** (rreshtat i) midis threads-ave të disponueshëm.Secili thread llogarit i pavarur rreshtat e tij të matricës C.Në vend që të përdorim matrica standarde 2D (double A[N][N]), ne përdorim alokim dinamik 1D (new double[N*N]). Kjo krijon një bllok të vazhdueshëm memorieje, gjë që e bën leximin e të dhënave shumë më të shpejtë për procesorin dhe shmang gabimet "Stack Overflow" për N të mëdha.

Shpjegimi i Funksioneve

new double[N * N] dhe delete[]; Alokon memorie në Heap (memoria e madhe e sistemit) dhe jo në Stack.

#pragma omp parallel for Kjo është direktiva kryesore që krijon grupin e threads dhe u ndan atyre punën e ciklit for që vjen menjëherë pas saj.

schedule(static);Ndan punën në blloqe të barabarta që në fillim të ekzekutimit.

shared(A, B, C); Përcakton se të gjithë threads duhet të lexojnë nga të njëjtat matrica A dhe B dhe të shkruajnë te e njëjta matricë C. Kjo kursen memorie duke mos krijuar kopje për çdo thread.

Madhesia	Threads	Koha (sek)	Speedup
500	1	0.3980	1.00x
500	2	0.2080	1.91x
500	4	0.1110	3.59x
500	6	0.0910	4.37x
500	8	0.0690	5.77x
500	12	0.0530	7.51x
1000	1	3.1510	1.00x
1000	2	1.6000	1.97x
1000	4	0.9170	3.44x
1000	6	0.6350	4.96x
1000	8	0.5180	6.08x
1000	12	0.3620	8.70x
1500	1	13.7690	1.00x
1500	2	6.7420	2.04x
1500	4	3.3820	4.07x
1500	6	2.3470	5.87x
1500	8	1.9400	7.10x
1500	12	1.8530	7.43x

Tabela 3. Koha e ekzekutimit dhe Speed-up të metodës 3

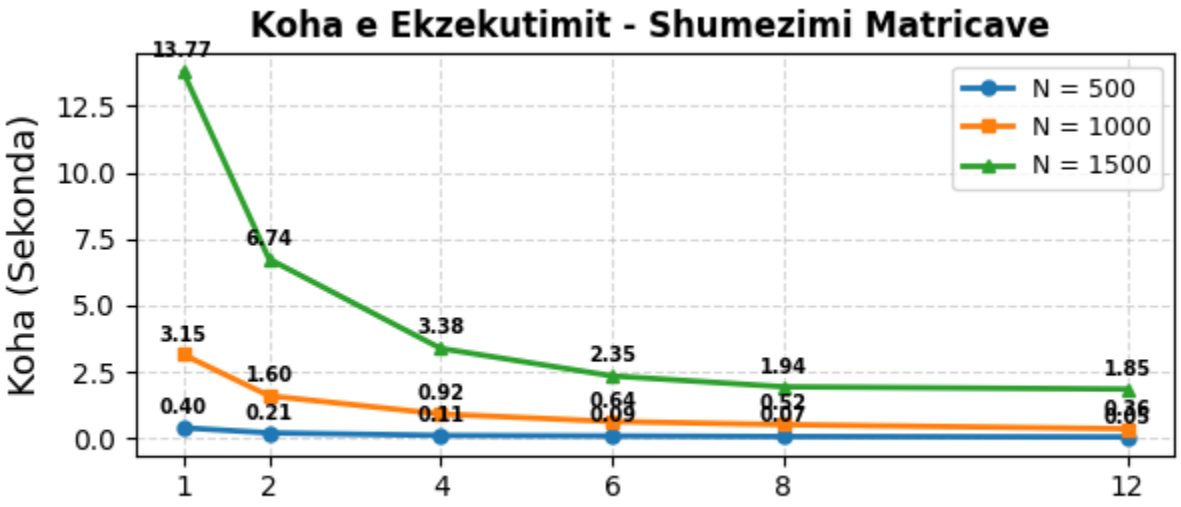


Figura 7 Grafiku i kohës së ekzekutimit në varësi të threads për Metodën 3

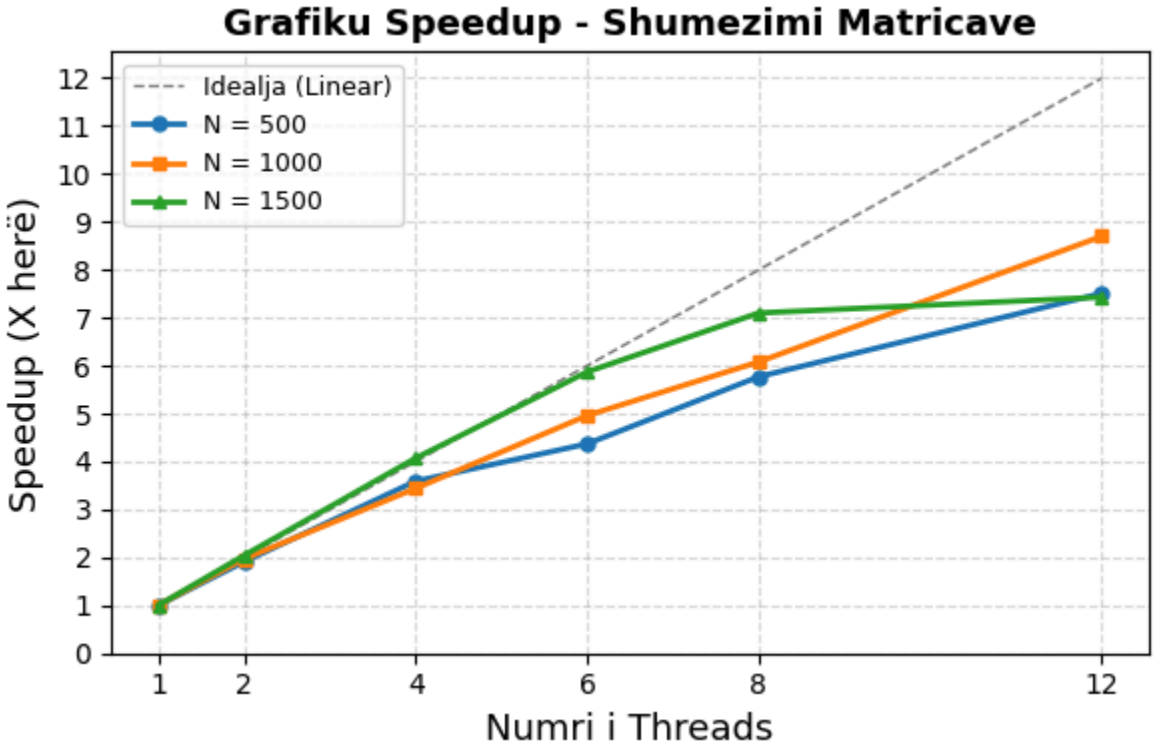


Figura 8 Grafiku i analizës së Speed-up për Metodën 3

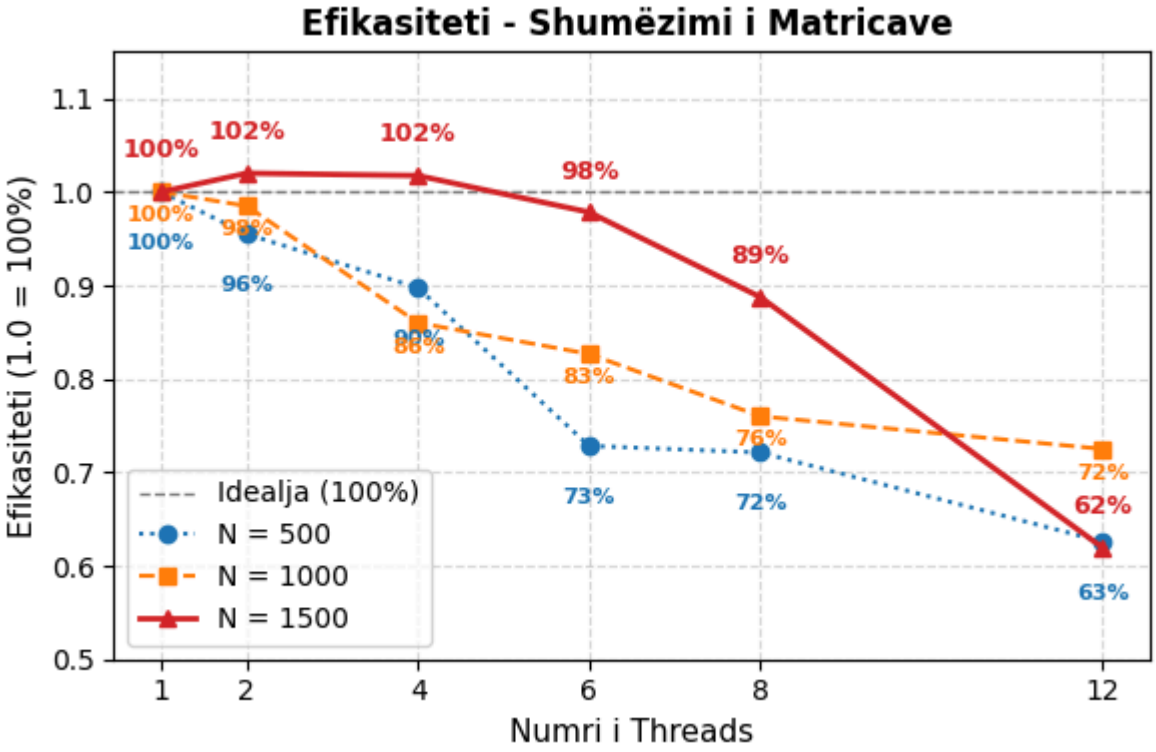


Figura 9 Grafiku i Analizës së Efikasitetit për Metodën 3

2. Përfundime

Grafiku i kohës tregon qartë ndikimin e paralelizmit në detyra me ngarkesë të lartë llogaritëse. Rezultati tregoi se shumëzimi i matricave është kandidati ideal për paralelizëm. Duke përdorur OpenMP, arritëm të reduktojmë kohën e ekzekutimit me rreth **87%** (Speedup 8.7x). Rezultatet treguan se efikasiteti maksimal arrihet kur të dhënat janë mjaftueshëm të mëdha për të justifikuar paralelizmin ($N \geq 1000$), por jo aq të mëdha sa të shkaktojnë mbingarkesë termike në procesor.

Analiza e Efikasitetit

Këtu shohim efikasitet shumë të lartë. Te N=1500, efikasiteti qëndron mbi 89% deri në 8 Threads.Shumëzimi i matricave është CPU Bound. Për çdo numër që merr nga memoria, procesori bën shumë shumëzime dhe mbledhje. Kjo e mban procesorin të zënë me punë dhe nuk e lë të presë RAM-in. Efekti "Super-Linear" : Për N=1500, me 2 dhe 4 Threads, efikasiteti është 102%. Ndarja e matricës së madhe (1500 x 1500) në blloqe më të vogla bëri që procesorët t'i gjenin të dhënat direkt në Cache Memory, duke tejkaluar shpejtësinë teorike.Te N=1500ë, kur kalojmë në 12 Threads, efikasiteti bie papritur në 62%.Kjo mund të ndodhë sepse kostoja e menaxhimit të 12 pjesëve bëhet më e madhe se përfitimi.

3. Shtojca 3

```
#include <omp.h>
#include <iostream>
#include <iomanip>
#include <stdlib.h>
using namespace std;
int main() {
    int N_values[] = { 500, 1000, 1500 };
    int total_N = 3;

    // 2. Numrat e Threads
    int threads_to_test[] = { 1, 2, 4, 6, 8, 12 };
    int total_T = 6;
    cout << setw(10) << "Madhesia"
         << setw(10) << "Threads"
         << setw(15) << "Koha (sek)"
         << setw(15) << "Speedup" << endl;
    for (int t = 0; t < total_N; t++) {
        int N = N_values[t];
        double *A = new double[N * N];
        double *B = new double[N * N];
        double *C = new double[N * N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                A[i * N + j] = i + j + 1;
                B[i * N + j] = i - j;
                C[i * N + j] = 0;
            }
        }
        double serial_time = 0.0;
        for (int j_test = 0; j_test < total_T; j_test++) {
            int num_threads = threads_to_test[j_test];
```

```
            omp_set_num_threads(num_threads);

                            #pragma omp parallel for
            for (int k = 0; k < N * N; k++) C[k] = 0;

            double start_time = omp_get_wtime();
            #pragma omp parallel for schedule(static) shared(A, B, C)
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    double sum = 0.0;
                    for (int k = 0; k < N; k++) {
                        sum += A[i * N + k] * B[k * N + j];
                    }
                    C[i * N + j] = sum;
                }
            }

            double end_time = omp_get_wtime();
            double time_taken = end_time - start_time;
            if (num_threads == 1) serial_time = time_taken;
            double speedup = 0.0;
            if (time_taken > 0) speedup = serial_time / time_taken;
            cout << setw(10) << N
                 << setw(10) << num_threads
                 << setw(15) << fixed << setprecision(4) << time_taken
                 << setw(14) << setprecision(2) << speedup << "x" << endl;
        }
        delete[] A;
        delete[] B;
        delete[] C;
    }
    cout << "Testimi perfundoi." << endl;
    int x; cin >> x;
    return 0;
}
```

6. Metoda 4 – Përafrimi i konstantes π

1. Përshkrimi Metodës

Kjo metodë ka për qëllim vlerësimin numerik të konstantes π (Pi) duke shfrytëzuar formulën e Eulerit, si dhe matjen e saktë të kohës së ekzekutimit për këtë llogaritje. Algoritmi bazohet në formulën e Euler-it për shumën e inversëve të katro

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots = \frac{\pi^2}{6}$$

Nga kjo formulë, ne veçojmë π:

$$\pi = \sqrt{6 \sum_{n=1}^{N} \frac{1}{n^2}}$$

Si funksionon kodi ?

Kemi një variabël të përbashkët s (shuma).Nëse 12 threads tentojnë të shtojnë vlerën e tyre te s në të njëjtën kohë (s = s + ...), krijohet Race Condition (Konflikt për resurse) dhe rezultati del i gabuar.
Për ta zgjidhur këtë, përdoret : #pragma omp parallel for reduction(+:s)
OpenMP i jep secilit thread një kopje private të variablit s (të inicializuar me 0).Çdo thread llogarit pjesën e tij të serisë dhe e shton te s-ja e tij lokale. Kur të gjithë threads mbarojnë punën, OpenMP merr të gjitha shumat lokale dhe i mbledh ato te variabli kryesor global s.

Iteracione (N)	Threads	Koha (sek)	Speedup	Vlera Pi
100000000	1	0.3150	1.00x	3.1415926450
100000000	2	0.1530	2.06x	3.1415926440
100000000	4	0.0950	3.32x	3.1415926440
100000000	6	0.0630	5.00x	3.1415926440
100000000	8	0.0510	6.18x	3.1415926440
100000000	12	0.0550	5.73x	3.1415926440
500000000	1	1.5390	1.00x	3.1415926450
500000000	2	0.7890	1.95x	3.1415926469
500000000	4	0.4420	3.48x	3.1415926507
500000000	6	0.3050	5.05x	3.1415926521
500000000	8	0.2480	6.21x	3.1415926515
500000000	12	0.1790	8.60x	3.1415926518
1000000000	1	3.0620	1.00x	3.1415926450
1000000000	2	1.5390	1.99x	3.1415926459
1000000000	4	0.8510	3.60x	3.1415926478
1000000000	6	0.5990	5.11x	3.1415926498
1000000000	8	0.4710	6.50x	3.1415926517
1000000000	12	0.3320	9.22x	3.1415926530

Tabela 4. Koha e ekzekutimit dhe Speed-up të metodës 4

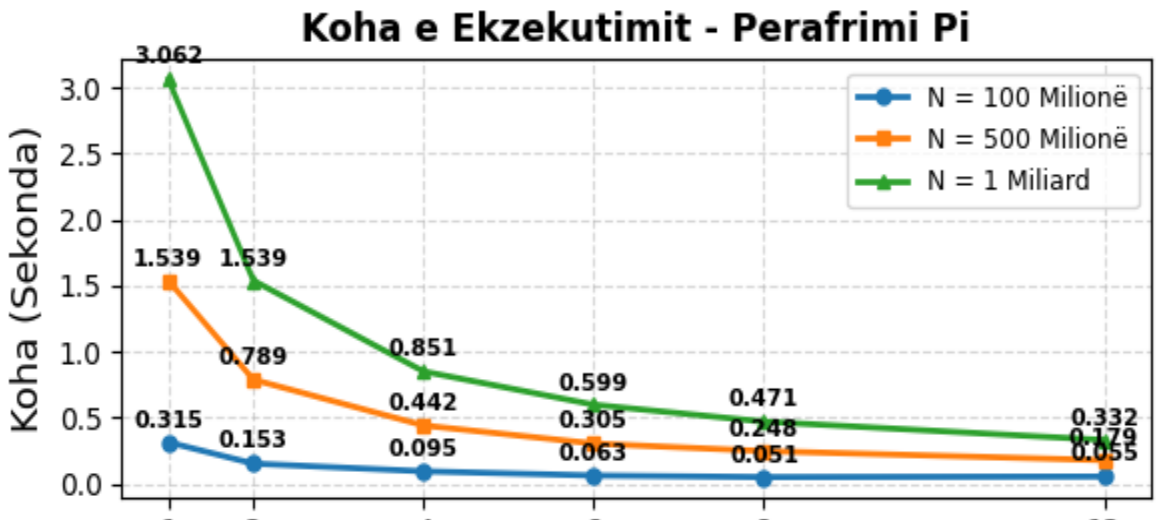


Figura 10 Grafiku i kohës së ekzekutimit për Metodën 4

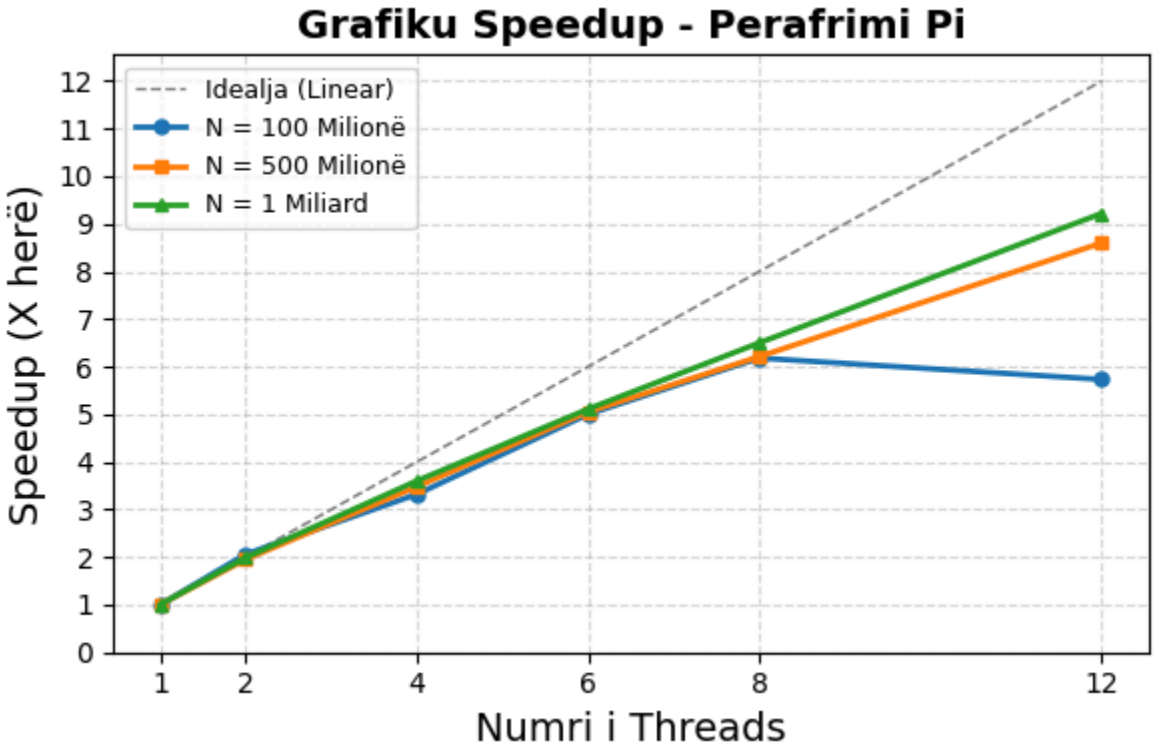


Figura 11
Grafiku i analizës së Speed-up për Metodën 4

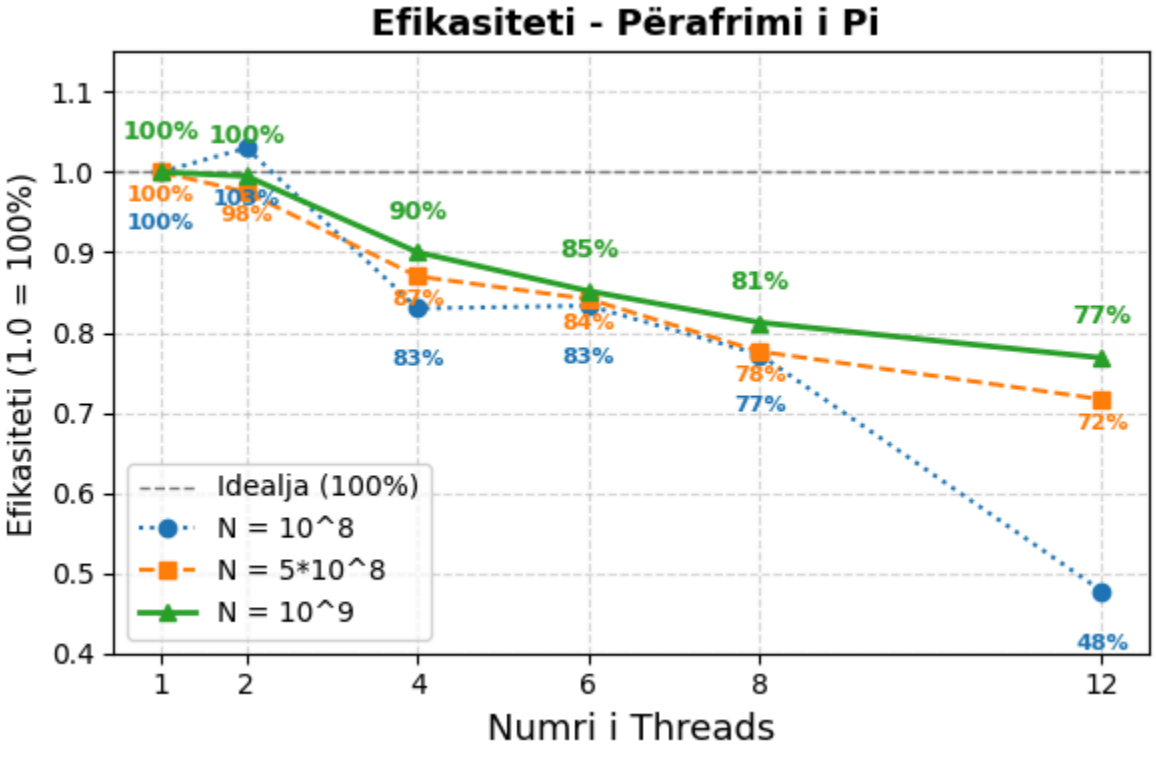


Figura 12
Grafiku i Analizës së Efikasitetit për Metodën 1

2. Përfundime

Grafiku i kohës për llogaritjen e Pi-së tregon qartë fuqinë e përpunimit paralel në detyra thjesht llogaritëse Për N=1 Miliard iteracione, koha ra nga **3.06 sekonda** (Serial) në **0.33 sekonda** (me 12 Threads). Kjo tregon se shtimi i thread-ve e ndan punën në mënyrë pothuajse të barabartë, duke e bërë programin rreth **9 herë më të shpejtë**.

Për N=100 Milionë, koha e ekzekutimit është shumë e ulët (nën 0.06s). Vëmë re se kur kalojmë nga 8 në 12 threads, koha rritet paksa (nga 0.051s në 0.055s). Kjo ndodh sepse koha që harxhon sistemi për të koordinuar 12 threads është më e madhe se koha që fitohet nga llogaritja për një problem kaq "të vogël".

Përfundime për Grafikun e Speedup-it: Meqenëse nuk ka vektorë për t'u lexuar nga RAM-i, procesori nuk "rri kot" asnjë nanosekondë. Ai vetëm llogarit. Kjo e lejon t'i afrohet shumë **Vijës Ideale** të grafikut.

Po të shikojmëi vijën blu (N=100M) ajo arrin kulmin te 8 Threads (6.18x) dhe pastaj bie te 12 Threads (5.73x).Ky fenomen quhet **Parallel Overhead**. Kur puna është e pakët, kostoja menaxheriale e ndarjes së punës dhe mbledhjes së rezultateve nga 12 threads është më e lartë se përfitimi. Kjo tregon se për probleme të vogla, nuk ia vlen të përdorësh të gjitha resurset e kompjuterit.

Analiza e Efektshmerise

Në ngarkesa të ulëta (me 2 Threads), vumë re se efikasiteti kaloi 100% (**1.03**). Kjo na tregoi se në praktikë, procesorët modernë nuk janë statikë; ata përdorin teknologji si **Turbo Boost** për të rritur shpejtësinë kur pak bërthama janë aktive, duke na dhënë rezultate më të mira se ç'parashikon matematika.Grafiku na mësoi gjithashtu se paralelizmi nuk është gjithmonë zgjidhja.Për problemin e vogël (N=10^8), efikasiteti ra nën 50% me 12 Threads.

3. Shtojca 4

```
#include <stdlib.h>
#include <omp.h>
#include <iostream>
#include <math.h>
#include <iomanip>

using namespace std;

int main()
{
    long long N_values[] = { 100000000, 500000000, 1000000000 };
    int total_N_tests = 3;
    int threads_to_test[] = { 1, 2, 4, 6, 8, 12 };
    int total_thread_tests = 6;
    cout << setw(15) << "Iteracione (N)"
         << setw(10) << "Threads"
         << setw(15) << "Koha (sek)"
         << setw(15) << "Speedup"
```

```
     << setw(18) << "Vlera Pi" << endl;
for (int i = 0; i < total_N_tests; i++) {
    long long N = N_values[i];
    double serial_time = 0.0;
    for (int j = 0; j < total_thread_tests; j++) {
        int num_threads = threads_to_test[j];
        omp_set_num_threads(num_threads);
        double s = 0.0;
        double pi = 0.0;
        double start_time = omp_get_wtime();
        #pragma omp parallel for reduction(+:s)
        for (long long k = 0; k < N; k++)
        {

            double x = (double)(k + 1);
            s = s + 1.0 / (x * x);
        }

        pi = sqrt(6 * s);
        double end_time = omp_get_wtime();
        double time_taken = end_time - start_time;
        if (num_threads == 1) serial_time = time_taken;
        double speedup = 0.0;
        if(time_taken > 0) speedup = serial_time / time_taken;
        cout << setw(15) << N
             << setw(10) << num_threads
             << setw(15) << fixed << setprecision(4) << time_taken
             << setw(14) << setprecision(2) << speedup << "x"
             << setw(18) << setprecision(10) << pi << endl;
    }

}
cout << "Testimi perfundoi" << endl;
int x; cin >> x;
return 0;
```

7. Metoda 5 – Metoda e Jakobit

1. Përshkrimi Metodës

Ky kod zbaton **Metodën e Jakobit** në mënyrë paralele duke përdorur OpenMP për të gjetur zgjidhjen numerike të një sistemi linear ekuacionesh të formës Ax = b.

Si funksionon kodi ?

Kodi fillimisht inicializon matricën e koeficientëve A, vektorin e anës së djathtë b, dhe përafrimin fillestar të zgjidhjes x .

- **Matrica A (Tridiagonale):** Krijohet një matricë N x N (ku N merr vlerat 500, 1000, 1500 për testim) me strukturë specifike:
  - $A_{i,i}$ = 2 (Diagonalja kryesore)
  - $A_{i,j}$ = -1 (Diagonalet ngjitur)
  - $A_{i,j}$ = 0 (Elementet e tjera)
- **Vektorët:** b inicializohet me 1 dhe përafrimi fillestar x inicializohet me 0.

Pjesa kryesore e kodit është cikli while, i cili kryen iteracionet derisa gabimi (eps) të jetë më i vogël se toleranca. Paralelizimi realizohet në dy hapa:

1. Llogaritja: Përdoret #pragma omp parallel for reduction(max: eps) për të llogaritur vlerat e reja **y** dhe për të gjetur gabimin maksimal global.
2. Përditësimi: Përdoret #pragma omp parallel for për të kopjuar vlerat nga **y** te **x.**

Formula Matematike

Në çdo hap iterativ, llogaritet një përafrim i ri i zgjidhjes, y (shënuar si $x_i^{(k+1)}$), duke përdorur përafrimin e vjetër x (shënuar si $x_j^{(k)}$).

Për çdo komponent i, llogaritja bëhet:

Formula e Metodës së Jakobit:

$$y_i = \frac{1}{A_{i,i}} \left(b_i - \sum_{j \neq i}^{N} A_{i,j} x_j^{(k)} \right)$$

Shpjegimi funksioneve

omp_set_num_threads(int num); I tregon sistemit se sa threads duhet të përdorë.

#pragma omp parallel for. Tregon që ciklin for që vjen menjëherë pas kësaj kaluzole mos ta ekzekutoj vetëm me një procesor, por ta ndaje në mënyrë të barabartë midis të gjithë threads që ka në dispozicion."

reduction(max: eps) Kur shumë threads duhet të kontrollojnë dhe modifikojnë të njëjtin variabël

(eps), krijohet konflikt. reduction i jep secilit thread një kopje lokale të eps. Në fund, OpenMP krahason të gjitha kopjet dhe mban vlerën Maksimum

schedule(static) Përcakton mënyrën se si ndahen rreshtat e matricës. static do të thotë që ndarja bëhet fiks që në fillim.

fabs(double x) Kthen vlerën absolute (modulin) të një numri.

Permasa	Threads	Koha (sek)	Speedup	Iteracione
500	1	4.0570	1.00x	5000
500	2	2.5470	1.59x	5000
500	4	1.9360	2.10x	5000
500	6	1.5270	2.66x	5000
500	8	1.2900	3.14x	5000
500	12	1.2760	3.18x	5000

1000	1	17.0870	1.00x	5000
1000	2	14.2870	1.20x	5000
1000	4	7.8890	2.17x	5000
1000	6	5.7300	2.98x	5000
1000	8	5.0740	3.37x	5000
1000	12	4.1040	4.16x	5000
1500	1	55.4430	1.00x	5000
1500	2	18.2120	3.04x	5000
1500	4	14.8930	3.72x	5000
1500	6	11.9790	4.63x	5000
1500	8	7.6030	7.29x	5000
1500	12	6.0780	9.12x	5000

Tabela 5. Koha e ekzekutimit dhe Speed-up të metodës 5

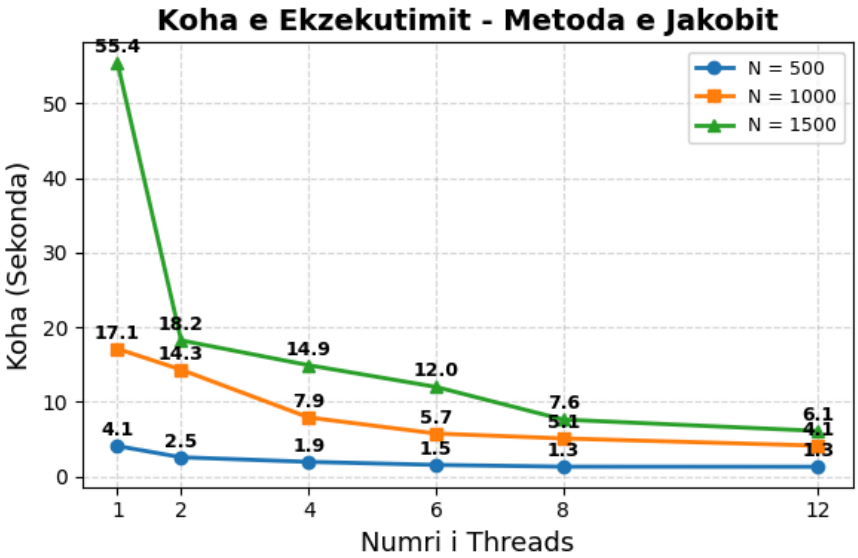


Figura 13 Grafiku i kohës së ekzekutimit për Metodën 5

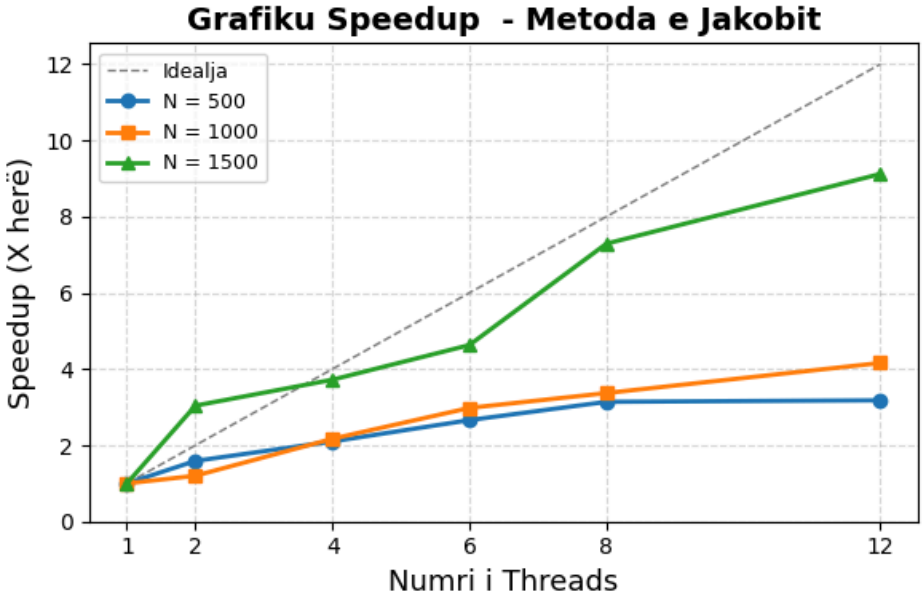


Figura 14 Grafiku i analizës Speed-up për Metodën 4

së

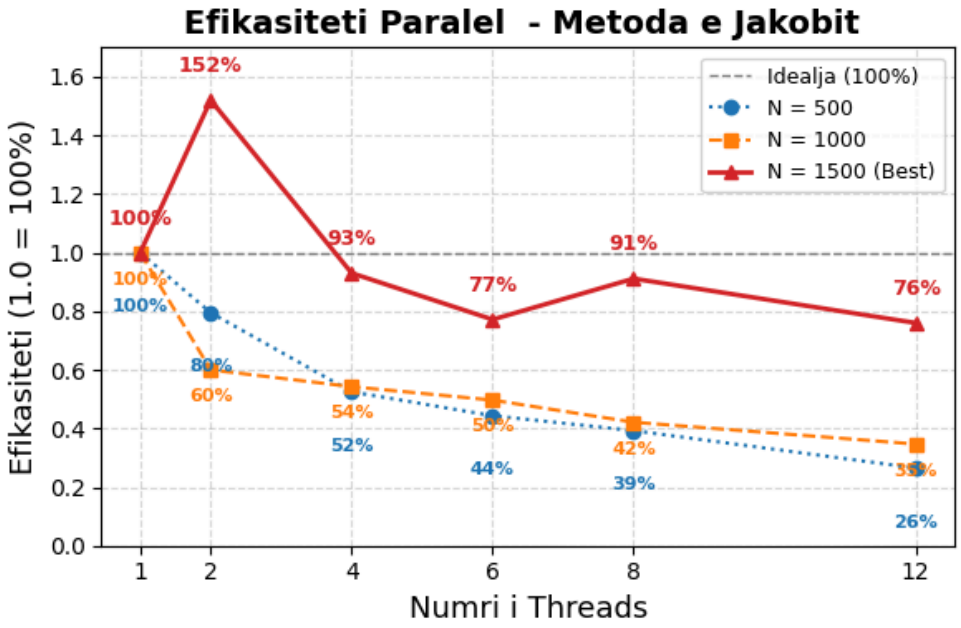


Figura 15 Grafiku i Analizës së Efikasitetit për Metodën 5

2. Përfundime

Grafiku i kohës tregon një përmirësim drastik të performancës kur rritet madhësia e problemit.

Për rastin më të rëndë (N=1500), koha e llogaritjes ra nga **55.44 sekonda** (Serial) në vetëm **6.07 sekonda** (me 12 Threads). Kjo konfirmon se metoda Jacobi është shumë e përshtatshme për paralelizim.Për N=500, lakorja e kohës sheshon pas 8 Threads (bie nga 1.29s në 1.27s). Kjo tregon se për sisteme të vogla, kostoja e menaxhimit të threads (Overhead) fillon të dominojë mbi fitimin nga llogaritja.

Te N=1500, kur kalojmë nga 1 në 2 Threads, Speedup-i nuk është 2.0x, por **3.04x**. Gjithashtu te 4 Threads është **3.72x** (afër ideales).Kjo ndodh për shkak të efektit të **Cache Memory**. Kur matrica e madhe ndahet në pjesë më të vogla për shumë bërthama, të dhënat aksesohen shumë më shpejt nga memoria.

Analiza e Efikasitetit

Rezultati më i rëndësishëm i këtij eksperimenti vërehet te kurba e kuqe (N=1500).Në kalimin nga 1 në 2 Threads, efikasiteti kërceu në vlerën **1.52 (ose 152%)**.Kjo tregon se sistemi performoi më mirë se sa parashikimi teorik ideal.Për matricën e vogël (N=500), shohim një rënie të shpejtë të efikasitetit, duke arritur në **0.27 (27%)** me 12 Threads.Kjo konfirmon se për probleme të vogla, **Kostoja e Paralelizmit (Overhead)** dominon. Koha që OpenMP shpenzon për të krijuar threads, për t'i sinkronizuar ata te reduction, dhe për të ndarë punën, është më e madhe se koha që fitohet nga llogaritja. Përdorimi i 12 bërthamave për N=500 është shpërdorim resursesh.

3. Shtojca 5

```
#include <omp.h>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <stdlib.h>
using namespace std;
#define TOLERANCE 1e-9
#define MAX_ITER 5000
int main() {
    int N_values[] = { 500, 1000, 1500 };
    int total_N = 3;
    int threads_to_test[] = { 1, 2, 4, 6, 8, 12 };
    int total_T = 6;
    endl;
    cout << setw(10) << "Permasa"
         << setw(10) << "Threads"
         << setw(15) << "Koha (sek)"
         << setw(15) << "Speedup"
         << setw(15) << "Iteracione" << endl;
    for (int t = 0; t < total_N; t++) {
        int N = N_values[t];
        double *A = new double[N * N];
        double *b = new double[N];
        double *x = new double[N];
        double *y = new double[N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (i == j)
                    A[i * N + j] = 2.0;
                else if (i == (j - 1) || i == (j + 1))
                    A[i * N + j] = -1.0;
                else
                    A[i * N + j] = 0.0;
            }
            b[i] = 1.0;
            x[i] = 0.0;
        }
        double serial_time = 0.0;
        for (int j_test = 0; j_test < total_T; j_test++) {
            int num_threads = threads_to_test[j_test];
            omp_set_num_threads(num_threads);
            for(int k=0; k<N; k++) x[k] = 0.0;
            double start_time = omp_get_wtime();
            int iter = 0;
            double eps = TOLERANCE + 1.0;
            while (eps > TOLERANCE && iter < MAX_ITER) {
                eps = 0.0;
                #pragma omp parallel for reduction(max: eps) schedule(static)
                for (int i = 0; i < N; i++) {
                    double s = 0.0;
                    for (int j = 0; j < N; j++) {
                        if (i != j) {
                            s += A[i * N + j] * x[j];
                        }
                    }
                    y[i] = (b[i] - s) / A[i * N + i];
                    double v = fabs(y[i] - x[i]);
```

```
                    if (v > eps) eps = v;
                }
                #pragma omp parallel for schedule(static)
                for (int i = 0; i < N; i++) {
                    x[i] = y[i];
                }
                iter++;
            }
            double end_time = omp_get_wtime();
            double time_taken = end_time - start_time;
            if (num_threads == 1) serial_time = time_taken;
            double speedup = 0.0;
            if (time_taken > 0) speedup = serial_time / time_taken;

            cout << setw(10) << N
                 << setw(10) << num_threads
                 << setw(15) << fixed << setprecision(4) << time_taken
                 << setw(14) << setprecision(2) << speedup << "x"
                 << setw(15) << iter << endl;
        }
        delete[] A;
        delete[] b;
        delete[] x;
        delete[] y;
    }
    cout << "Testimi perfundoi." << endl;
    int k; cin >> k;
    return 0;
}
```

8. Metoda 6 – Metoda e Durand-Kerner

1. Përshkrimi Metodës

Metoda **Durand-Kerner** është një algoritëm numerik që shërben për të **gjetur të gjitha rrënjët e një polinomi njëkohësisht**.Ndryshe nga metodat klasike (si Newton-Raphson) që gjejnë një zgjidhje, pastaj pjesëtojnë polinomin për të gjetur tjetrën, kjo metodë i gjen të 12-ta zgjidhjet (në rastin tonë) në të njëjtën kohë.

1. Problemi që zgjidh

Ne kemi një ekuacion të shkallës së 12-të (te funksioni f(x) me pow(x, 12)).Matematikisht, ky ekuacion ka saktësisht 12 zgjidhje (rrënjë). Disa mund të jenë numra realë, disa kompleksë.

2. Si punon algoritmi

Supozojmë se kemi 12 numra të çfarëdoshëm . Ne duam t'i "shtyjme" këta numra dalëngadalë derisa të bëhen rrënjët e vërteta të ekuacioni.

Formula e Metodës së Durand-Kerner:

$$x_i^{e\,re} = x_i - \frac{f(x_i)}{\prod_{j \neq i}(x_i - x_j)}$$

$f(x_i)$ është vlera e polinomit (që duam të bëhet 0).Pjesa e poshtme llogarit distancën e asaj rrënjë nga të gjitha rrënjët e tjera të hamendësuara.

Komponentët Kryesorë të Kodit

Funksioni i Polinomit **f(x):**

- Kodi definon një polinom të shkallës së 12-të.
- Qëllimi matematik është të gjenden vlerat e x për të cilat f(x) = 0.

Inicializimi (Vektori **x**):

- Krijohet një vektor **x[n]** me 12 elemente.
- Meqenëse metoda është iterative, ajo kërkon vlera fillestare . Kodi i mbush këto vlera me numra të tipit **i + 1.5** (p.sh., 1.5, 2.5, etj.) për të shmangur mbivendosjen e rrënjëve në fillim.

Llogaritja Paralele (OpenMP):

- Ne përdorim direktivën #pragma omp parallel për të krijuar një grup threads. Ndryshe nga ndarja automatike, këtu secili thread llogarit manualisht cilat rrënjë i takojnë (p.sh., Thread 0 merr gjysmën e parë, Thread 1 gjysmën e dytë).
- Një element kritik në kod është komanda #pragma omp barrier. Meqenëse formula e Durand-Kerner kërkon pozicionet e të gjitha rrënjëve të tjera për të llogaritur saktë lëvizjen, asnjë thread nuk lejohet të vazhdojë te hapi tjetër pa përfunduar të gjithë llogaritjet e hapit aktual. Kjo siguron që matematika të mbetet e saktë ndërkohë që puna bëhet paralelisht.
- `shared (x, y, eps, it, gab_max)` Kjo është e domosdoshme sepse çdo Thread duhet të ketë akses leximi në pozicionet e rrënjëve që po llogariten nga Threads e tjerë për të formuar prodhimin $\prod_{j \neq i}(x_i - x_j)$

Threads	Koha (s)	Speedup	Iteracione
1	**0.0300**	**1.00x**	**9721**
2	**0.1910**	**0.16x**	**3240**
4	**2.4450**	**0.01x**	**20000**
6	**1.0180**	**0.03x**	**5974**
8	**2.5670**	**0.01x**	**11671**
12	**6.4980**	**0.00x**	**20000**

Rrrenjet e gjetura (Zgjidhjet):
x[0] = 5.00
x[1] = 2.00
x[2] = 4.00
x[3] = 3.00
x[4] = 6.00
x[5] = 7.00
x[6] = 8.00
x[7] = 9.00
x[8] = 10.00
x[9] = 13.00
x[10] = 11.00
x[11] = 12.00

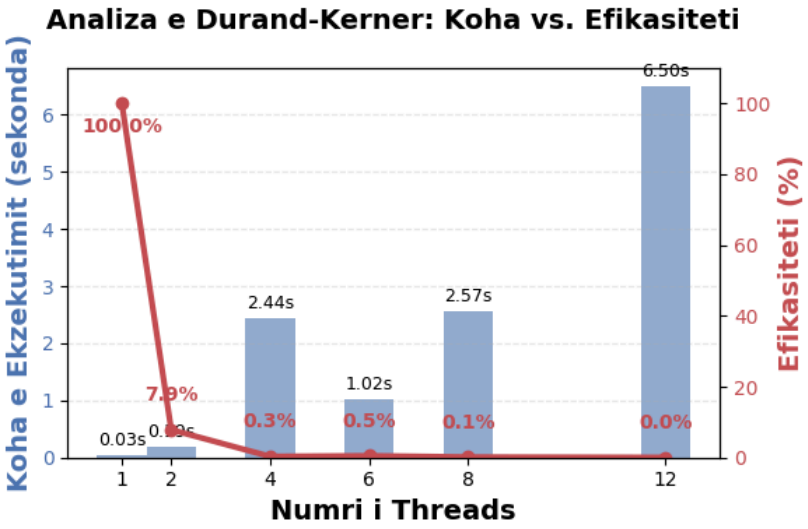


Figura 16 Analiza e grafikut Koha vs Efikasiteti per Metoden 6

2. Përfundime

Eksperimenti i gjetjes së rrënjëve të polinomit shërbeu si një demonstrim i shkëlqyer i **kufijve të paralelizmit**. Rezultatet treguan një rritje të kohës së ekzekutimit me shtimin e Threads (nga 0.03s në 6.50s), duke na çuar në këto konkluzione teknike:

Arsyeja kryesore e performancës së dobët është **Granulariteti i Imët :**
Për N=12, çdo Thread ka shumë pak punë për të bërë (llogaritja e 1 ose 2 rrënjëve).
Koha që i duhet sistemit për të krijuar Threads, për t'i shpërndarë dhe për t'i menaxhuar ata (Overhead), është shumë herë më e madhe se koha e vetë llogaritjes matematikore. Në këtë rast, kostoja e menaxhimit tejkaloi fitimin nga ndarja e punës.

Algoritmi Durand-Kerner kërkon sinkronizim të fortë. Komanda #pragma omp barrier u ekzekutua mijëra herë (për çdo iteracion). Në çdo hap, Threads e shpejtë u detyruan të prisnin, duke shkaktuar humbje masive të ciklit të procesorit. Për probleme të vogla, kostoja e ndalimit dhe rinisjes së Threads është shkatërruese për efikasitetin. Për madhësi të vogla problemi (N=12), ekzekutimi Serial është superior. Metoda Durand-Kerner me OpenMP do të ishte efikase vetëm për polinome të shkallëve shumë të larta (p.sh., N > 1000), ku koha e llogaritjes justifikon koston e sinkronizimit."

3. Shtojca 6

```
#include <omp.h>
#include <iostream>
#include <cmath>
#include <iomanip>
#include <vector>
using namespace std;
#define n 12
double f(double x) {
```

```
    return pow(x, 12)
        - 90 * pow(x, 11)
        + 3641 * pow(x, 10)
        - 87450 * pow(x, 9)
        + 1387023 * pow(x, 8)
        - 15282630 * pow(x, 7)
        + 119753843 * pow(x, 6)
        - 671189310 * pow(x, 5)
        + 2664929476 * pow(x, 4)
        - 7292774280 * pow(x, 3)
        + 13020978816 * pow(x, 2)
        - 13575738240 * x
        + 6227020800;
}

int main()
{
    int thread_counts[] = { 1, 2, 4, 6, 8, 12 };
    int num_tests = 6;
    double x[n], y[n];
    double tol = 0.5 * pow(10, -9);
    double koha_serial = 0;
        cout << setw(10) << "Threads" << setw(15) << "Koha (s)" << setw(15) <<
"Speedup" << setw(15) << "Iteracione" << endl;
    for (int t = 0; t < num_tests; t++) {
        int num_threads = thread_counts[t];

        int i, j, it = 0;
        double P = 1, eps = 2 * tol;
        for (i = 0; i < n; i++) {
            x[i] = (i + 0.4);
        }
        vector<double> gab_max(num_threads);
        omp_set_num_threads(num_threads);
        double start_time = omp_get_wtime();
        #pragma omp parallel shared (x, y, eps, it, gab_max) private(i, j, P)
        {
            int threadNum = omp_get_thread_num();
            int fillimi = n * threadNum / num_threads;
            int fundi = n * (threadNum + 1) / num_threads;
            double gab;
            while (eps > tol && it < 20000)
            {
                gab_max[threadNum] = 0;
                for (i = fillimi; i < fundi; i++)
                {
                    P = 1;
                    for (int j = 0; j < n; j++) {
                        if (i != j) {
                            P = P * (x[i] - x[j]);
                        }
                    }
                    if (abs(P) < 1e-20) P = 1e-20;
                    y[i] = x[i] - f(x[i]) / P;
```

```
                    gab = abs(y[i] - x[i]);
                    if (gab > gab_max[threadNum])
                        gab_max[threadNum] = gab;
                    x[i] = y[i];
                }
                #pragma omp barrier
                #pragma omp master
                {
                    it++;
                    eps = 0;
                    for (i = 0; i < num_threads; i++)
                    {
                        if (eps < gab_max[i])
                            eps = gab_max[i];
                    }
                }

                #pragma omp barrier
            }
        }
        double time_taken = omp_get_wtime() - start_time;
        if (num_threads == 1) {
            koha_serial = time_taken;
        }
        double speedup = koha_serial / time_taken;
        cout << setw(10) << num_threads
             << setw(15) << fixed << setprecision(4) << time_taken
             << setw(14) << fixed << setprecision(2) << speedup << "x"
             << setw(15) << it << endl;
    }

    cout << "\nRrenjet e gjetura (Zgjidhjet):" << endl;
    for (int i = 0; i < n; i++)
        cout << "x[" << i << "] = " << x[i] << endl;

    return 0;
}
```

9. Metoda 7 – Metoda e Trapezit

1. Përshkrimi Metodës

Kjo metodë bën llogaritjen e përafërt të **integralit të caktuar** duke përdorur **Rregullin e Trapezit**.

Kodi llogarit integralin e funksionit $f(x) = \frac{1}{x}$ në intervalin nga a=1 në b=2 . Matematikisht dimë që:

$\int_1^2 \frac{1}{x}\,dx = ln(2) - ln(1) \approx 0.693147$. Kompjuteri e zgjidh këtë duke e ndarë sipërfaqen në 10, 100 dhe 500 milionë trapeza të vegjël vertikalë dhe duke mbledhur sipërfaqet e tyre.

Ndarja e Punës: Kemi ciklin **for (int i = 1; i < n - 1; i++)**. Meqë n=100 milionë, OpenMP e ndan këtë varg gjigant në 4 pjesë (nga 25 milionë për çdo thread).

Në vend që të gjithë threads të shkojnë te një variabël i vetëm **s**, secili thread ka "vend" të vetin.

- Thread 0 mbush `sum[0]`.
- Thread 1 mbush `sum[1]`.
- Thread 2 mbush `sum[2]`.
- Thread 3 mbush `sum[3]`. Kjo bën që ata të punojnë plotësisht të pavarur, pa pritur njëri-tjetrin.

Formula e Metodës së Trapezit:

$$I \approx \frac{h}{2}[f(a) + 2 * shuma + f(b)]$$

Madhesia	Threads	Koha (sek)	Speedup
10000000	1	0.0310	1.00x
10000000	2	0.0320	0.97x
10000000	4	0.0480	0.65x
10000000	6	0.0510	0.61x
10000000	8	0.0320	0.97x
10000000	12	0.0440	0.70x
100000000	1	0.2560	1.00x
100000000	2	0.4110	0.62x
100000000	4	0.2370	1.08x
100000000	6	0.3530	0.73x
100000000	8	0.3970	0.64x
100000000	12	0.3720	0.69x
500000000	1	1.3170	1.00x
500000000	2	1.9890	0.66x
500000000	4	1.2610	1.04x
500000000	6	1.6750	0.79x
500000000	8	1.9560	0.67x
500000000	12	1.9080	0.69x

Tabela 6. Koha e ekzekutimit dhe Speedup i metodës 7

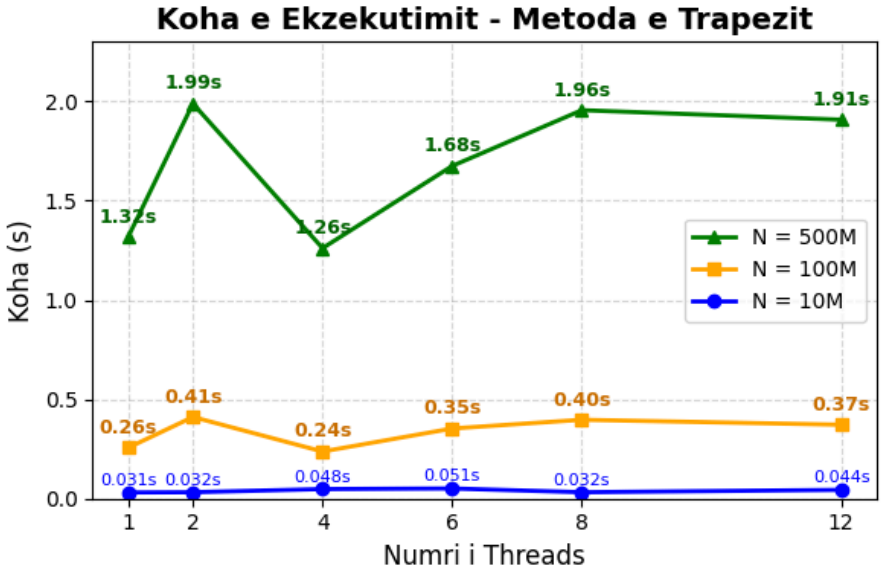


Figura 17 Grafiku i Analizës së Kohes se Ekzekutimit për Metodën 7

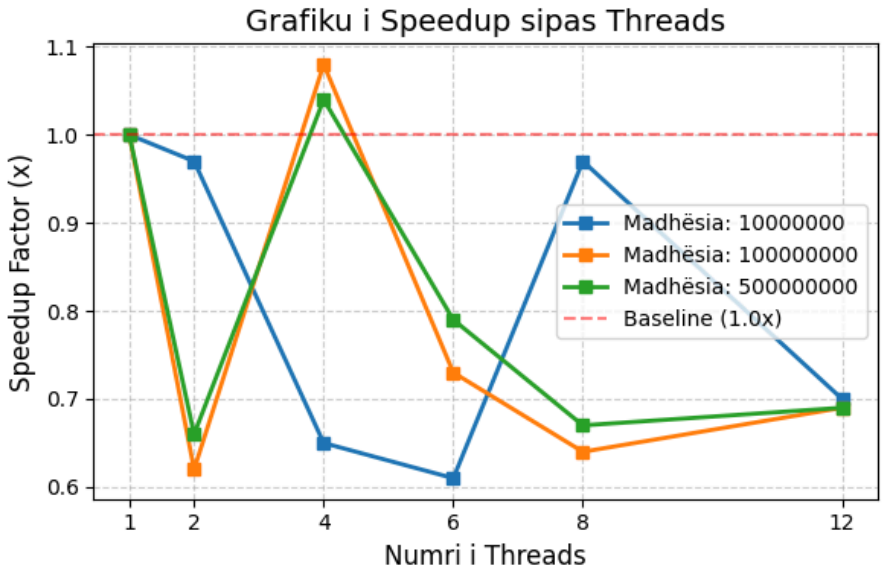


Figura 18 Grafiku i Analizës së Speed-Up për Metodën 7

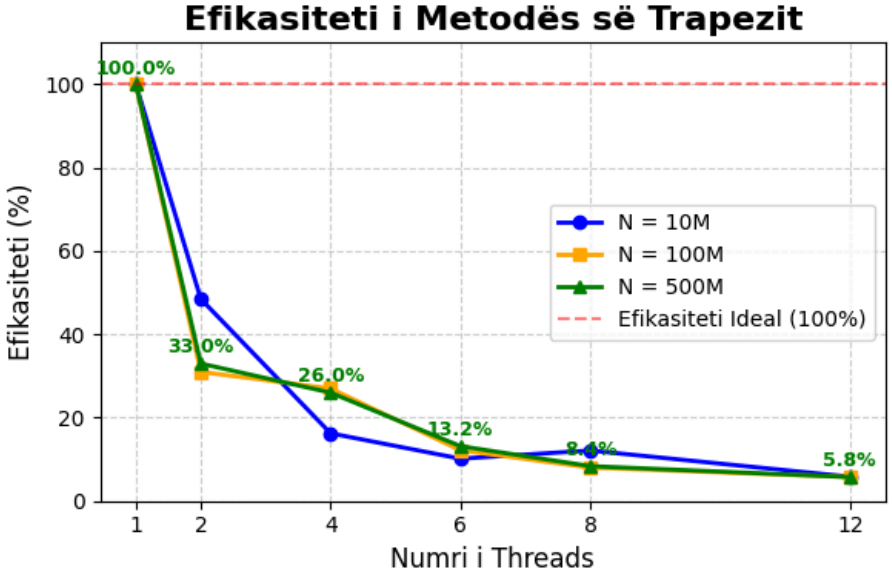


Figura 19 Grafiku i Analizës së Efikasitetit për Metodën 7

2. Përfundime

Grafiku tregon një sjellje jo të zakonshme për sistemet paralele: në vend që koha të ulet me shtimin e Threads, ajo ka tendencë të rritet ose të luhatet.Kjo rritje e kohës shkaktohet nga **Overhead (Kostoja e Menaxhimit)**. Funksioni f(x)=1/x është aq i lehtë për t'u llogaritur, saqë procesori shpenzon më shumë kohë duke krijuar Threads dhe duke menaxhuar memorien e vektorit sum[], sesa duke bërë llogaritjen efektive.
Grafiku i Speedup tregon se performanca qëndron kryesisht nën vijën bazë **1.0x** (vija e kuqe).Vëmë re një "Pikë Optimale" te **4 Threads** për N=100M dhe N=500M, ku Speedup arrin pak mbi 1.0x (1.04x - 1.08x). Grafiku paraqet një rënie drastike dhe të vazhdueshme.Kjo rënie tregon se shtimi i burimeve (Threads) është shpërdorim për këtë lloj problemi. Një efikasitet prej 10% do të thotë se 90% e fuqisë së procesorëve po harxhohet për sinkronizim dhe komunikim, dhe vetëm 10% po përdoret për matematikë. Kjo konfirmon ligjin e **Amdahl-it**: Kur pjesa e llogaritjes është shumë e vogël krahasuar me kohën e komunikimit, paralelizmi nuk ia vlen.

3. Shtojca 7

```
#include <omp.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>
using namespace std;
// Funksioni f(x) = 1/x
double f(double x) {
    return 1.0 / x;
}
int main() {
    long long N_list[] = { 10000000, 100000000, 500000000 };
    int num_N = 3;
    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_threads_count = 6;
    double a = 1, b = 2;
    cout << "      Madhesia     Threads     Koha (sek)      Speedup" << endl;
    for (int i = 0; i < num_N; i++) {
        long long n = N_list[i];
        double h = (b - a) / n;
        double koha_serial = 0;
        for (int j = 0; j < num_threads_count; j++) {
            int num_threads = threads_list[j];
            vector<double> sum(num_threads);
            double s = 0;
            omp_set_num_threads(num_threads);
            double start = omp_get_wtime();
            #pragma omp parallel
            {
                int id = omp_get_thread_num();
                sum[id] = 0;
```

```
                #pragma omp for
                for (long long k = 1; k < n; k++) {
                    sum[id] += f(a + k * h);
                }
            }
            for (int k = 0; k < num_threads; k++) s += sum[k];
            volatile double result = h / 2.0 * (f(a) + 2 * s + f(b));
            double time_taken = omp_get_wtime() - start;
            if (time_taken < 1e-9) time_taken = 1e-9;
            double speedup = 0.0;
            if (num_threads == 1) {
                koha_serial = time_taken;
                speedup = 1.00;
            } else {
                speedup = koha_serial / time_taken;
            }
            cout << setw(14) << n
                 << setw(8) << num_threads
                 << setw(14) << fixed << setprecision(4) << time_taken
                 << setw(14) << fixed << setprecision(2) << speedup << "x" << endl;
        }
    }

    // system("pause");
    return 0;
}
```

10. Metoda 8 – Algoritmi i renditjes QuickSort

1. Përshkrimi Metodës

QuickSort është një algoritëm i tipit "Përça dhe Sundo".

- **Zgjedhja e Pivotit :** Algoritmi zgjedh elementin e parë si pikë referimi.
- **Ndarja :**
  - Indeksi **`i`** lëviz djathtas për të gjetur numra më të mëdhenj se pivoti.
  - Indeksi **`j`** lëviz majtas për të gjetur numra më të vegjël se pivoti.
  - Këta numra ndërrohen me njëri-tjetrin .
  - Në fund, pivoti vendoset në pozicionin e tij përfundimtar (indeksi **`j`**), ku të gjithë numrat majtas janë më të vegjël dhe të gjithë numrat djathtas janë më të mëdhenj.

2. Pjesa Paralele

Në vend që të thërrasë funksionin për pjesën e majtë dhe të djathtë njëra pas tjetrës (si në QuickSort normal), ky kod përpiqet t'i bëjë të dyja njëkohësisht.

`#pragma omp parallel sections`: Krijon një grup threads për të ekzekutuar blloqe të ndryshme kodi.
`#pragma omp section`: Përcakton punën për secilin thread.

Kompleksiteti kohor

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

T(N): Koha për të renditur N numra.

$2T\left(\frac{N}{2}\right)$: Algoritmi e ndan problemin në **2** nën-probleme me gjysmën e madhësisë N/2.

+N: Është koha që duhet për të bërë kalimin linear për të ndarë numrat majtas/djathtas pivotit.

Madhesia	Threads	Koha (sek)	Speedup
1000000	1	0.0940	1.00x
1000000	2	0.1140	0.82x
1000000	4	0.0950	0.99x
1000000	6	0.1090	0.86x
1000000	8	0.0880	1.07x
1000000	12	0.1100	0.85x
5000000	1	0.8490	1.00x
5000000	2	0.8170	1.04x
5000000	4	0.8320	1.02x
5000000	6	0.8320	1.02x
5000000	8	0.8240	1.03x
5000000	12	0.8230	1.03x
10000000	1	2.4880	1.00x
10000000	2	2.4200	1.03x
10000000	4	2.4600	1.01x
10000000	6	3.4960	0.71x
10000000	8	3.9420	0.63x
10000000	12	3.9220	0.63x

Tabela 7. Koha e ekzekutimit dhe Speedup i metodës 8

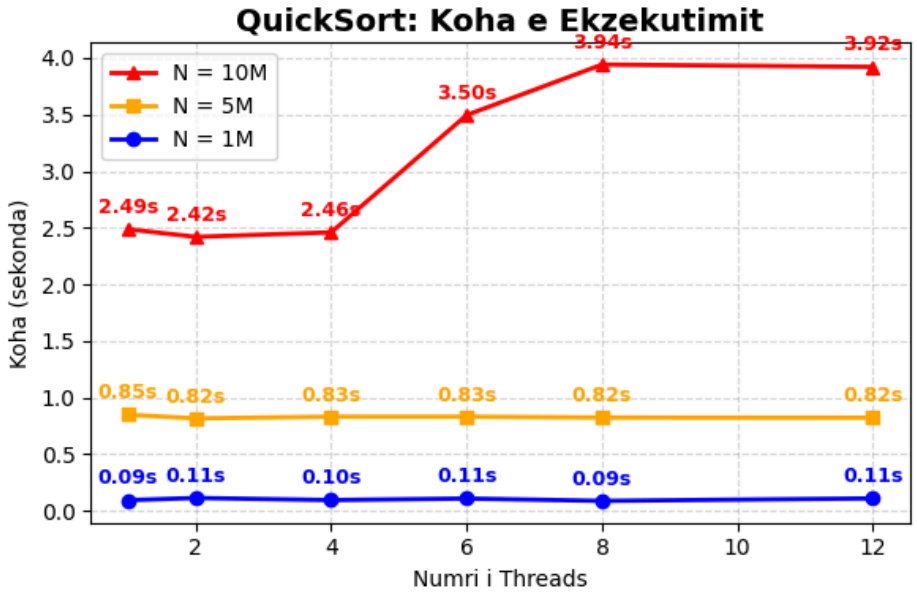


Figura 20 Grafiku i kohës së ekzekutimit për Thread 1-5 per algoritmin e renditjes QuickSort

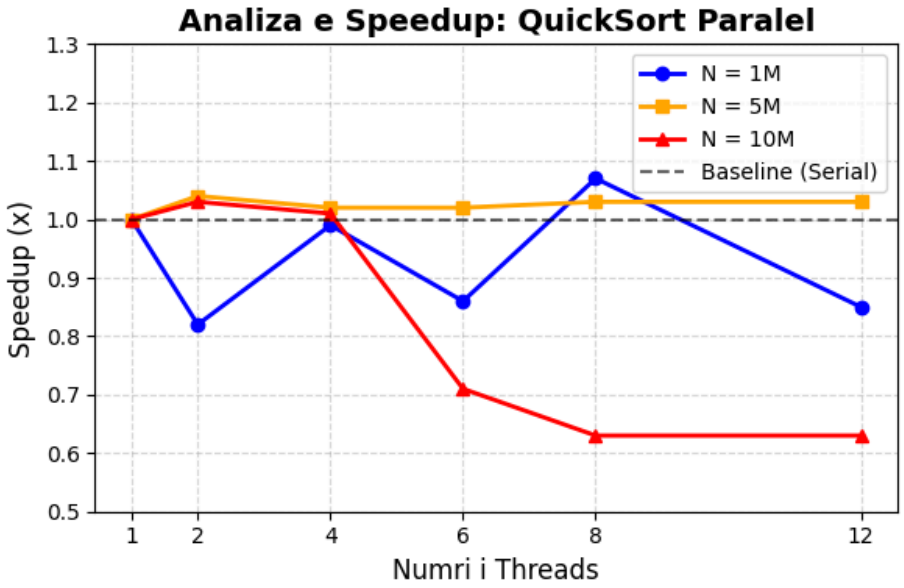


Figura 21 Grafiku i Analizës së Speed-Up për Metodën 8

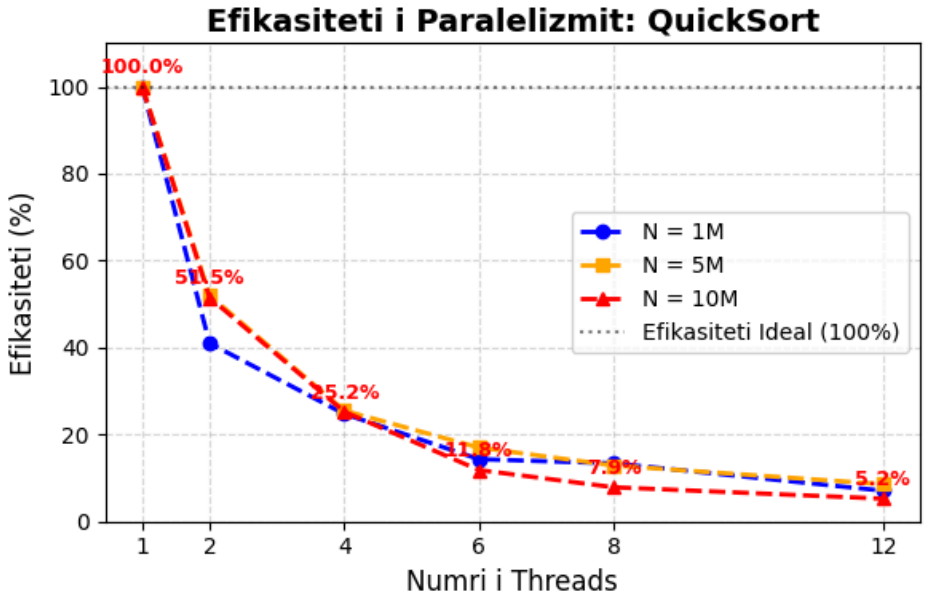


Figura 22 Grafiku i Analizës së Efikasitetit për Metodën 8

2. Përfundime

Grafiku tregon se për madhësi mesatare (N = 1M dhe N = 5M), koha e ekzekutimit mbetet pothuajse konstante pavarësisht shtimit të threads. Megjithatë, te madhësia maksimale **N = 10 milionë**, vërehet një rritje e papritur dhe e fortë e kohës pas kalimit të pragut prej 4 threads.Ky fenomen ndodh për shkak të **Saturimit të Memories (Memory Bandwidth)**. QuickSort kërkon lëvizje të vazhdueshme të të dhënave në RAM. Kur 12 threads përpiqen të shkruajnë dhe lexojnë 40MB të dhëna në të njëjtën kohë, krijohet një "shishe e ngushtë" në procesor, duke e ngadalësuar sistemin në vend që ta përshpejtojë.

Grafiku i Speedup tregon se algoritmi QuickSort paralel lufton për të kaluar kufirin e performancës seriale

(vija 1.0x).QuickSort ka një pjesë seriale shumë të rëndësishme: **Partition (Ndarja)**. Derisa pivoti i parë të vendoset në vend, të gjithë threads e tjerë qëndrojnë bosh. Kjo pjesë seriale kufizon Speedup-in maksimal që mund të arrijmë.
Vumë re se te N = 5M, Speedup-i ishte më i qëndrueshmi (afër 1.0), gjë që tregon një ekuilibër midis përfitimit llogaritës dhe koston e menaxhimit të threads. Te N = 1M, rezultatet ishin të paqëndrueshme për shkak të kohës së shkurtër të ekzekutimit që ndikohej nga proceset e sistemit, ndërsa te N = 10M, vumë re një degradim të performancës mbi 4 threads për shkak të (Memory Bandwidth Bottleneck)."
 Efikasiteti nis nga 100% (me 1 thread) dhe bie me shpejtësi drejt 5% - 8% kur përdoren 12 threads.Një efikasitet prej 5.2% (për N = 10M me 12 threads) do të thotë se pothuajse e gjithë fuqia llogaritëse po "digjet" kot për menaxhimin e rekursionit dhe për të pritur radhën në memorien RAM.

3. Shtojca 8

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <iomanip>
#include <iostream>

using namespace std;
void quicksort(int a[], int el1, int elF) {
   int i, j, celes, temp;
   if (el1 < elF) {
      celes = el1;
      i = el1;
      j = elF;
      while (i < j) {
         while (a[i] <= a[celes] && i < elF) i++;
         while (a[j] > a[celes]) j--;
         if (i < j) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }
      }
      temp = a[celes];
      a[celes] = a[j];
      a[j] = temp;

      if (j - el1 > 1000) {
         #pragma omp parallel sections
         {
            #pragma omp section
            quicksort(a, el1, j - 1);

            #pragma omp section
            quicksort(a, j + 1, elF);
         }
      } else {

         quicksort(a, el1, j - 1);
         quicksort(a, j + 1, elF);
      }
   }
}

int main() {
   long N_list[] = { 100000, 500000, 1000000 };
   int threads_list[] = { 1, 2, 4, 6, 8, 12 };

   for (int n_idx = 0; n_idx < 3; n_idx++) {
      long current_N = N_list[n_idx];
      double koha_serial = 0;

      for (int t_idx = 0; t_idx < 6; t_idx++) {
         int current_threads = threads_list[t_idx];

         int* a = (int*)malloc(current_N *
sizeof(int));
         srand(300);
         for (int i = 0; i < current_N; i++) a[i] =
rand();

         omp_set_num_threads(current_threads);
         double fillimi = omp_get_wtime();

         quicksort(a, 0, current_N - 1);

         double ke = omp_get_wtime() - fillimi;

         if (current_threads == 1) koha_serial = ke;
         double speedup = koha_serial / ke;

         printf("%14ld %8d %14.4f %14.2fx\n",
current_N, current_threads, ke, speedup);

         free(a);
      }
   }

   return 0;}
```

11. Metoda 9 – Metoda për llogaritjen e n!

Të shkruhet nje program ne OpenMP për të llogaritur n! për n = 100, 1000, 10000, 100000 dhe duke përdorur numrin e threads 1, 2, 3, 4 .Të llogaritet koha, speed up, efektshmëria dhe masa e efektshmërisë.

1. Përshkrimi Metodës

Kjo metodë llogarit gjetjen e faktorialit të një numri në ekzekutimin paralel .Kam përdor direktivën **#pragma omp parallel for reduction(*:fact).**
Kjo teknikë ndan automatikisht iteracionet e ciklit midis threads të disponueshme.

Klauzola **reduction** siguron saktësinë e rezultatit duke krijuar kopje lokale të variablit për çdo thread dhe duke i shumëzuar ato në fund, duke shmangur kështu konfliktet e memories .Për shkak se llogaritja e një faktoriali të vetëm është tepër e shpejtë (nanosekonda) për t'u matur saktë nga ora e sistemit, u implementua një teknikë e **ngarkesës sintetike**. Algoritmi ekzekutohet brenda një cikli prej **20,000 përsëritjesh** për çdo matje, duke gjeneruar një kohë totale të matshme dhe të qëndrueshme.
Tabela e rezultateve përmban metrikat e mëposhtme:

Koha (T_p): Koha që duhet për të kryer llogaritjen me p threads.

Speed-up (S): Tregon sa herë më shpejt punon programi krahasuar me versionin serial.

$S = \frac{T_{serial}}{T_{paralele}}$

Efektshmëria (E): Tregon sa mirë po përdoren burimet (threads). $E=\frac{S}{P}$ (p është numri i threads).

Masa e Efektshmërisë: Është thjesht efektshmëria e shprehur në përqindje (E * 100).

N	Threads	Koha (s)	Speedup	Efikasiteti	Masa (%)
100	1	0.073000	1.00	1.00	100.00 %
100	2	0.525000	0.14	0.07	6.95 %
100	3	0.635000	0.11	0.04	3.83 %
100	4	0.635000	0.11	0.03	2.87 %
1000	1	0.189000	1.00	1.00	100.00 %
1000	2	0.558000	0.34	0.17	16.94 %
1000	3	0.589000	0.32	0.11	10.70 %
1000	4	0.657000	0.29	0.07	7.19 %
10000	1	1.264000	1.00	1.00	100.00 %
10000	2	1.206000	1.05	0.52	52.40 %
10000	3	1.025000	1.23	0.41	41.11 %
10000	4	1.103000	1.15	0.29	28.65 %
100000	1	11.981000	1.00	1.00	100.00 %
100000	2	6.743000	1.78	0.89	88.84 %
100000	3	5.514000	2.17	0.72	72.43 %
100000	4	5.629000	2.13	0.53	53.21 %

Tabela 8. Koha e ekzekutimit, Efektshmerise dhe Speedup i metodës 9

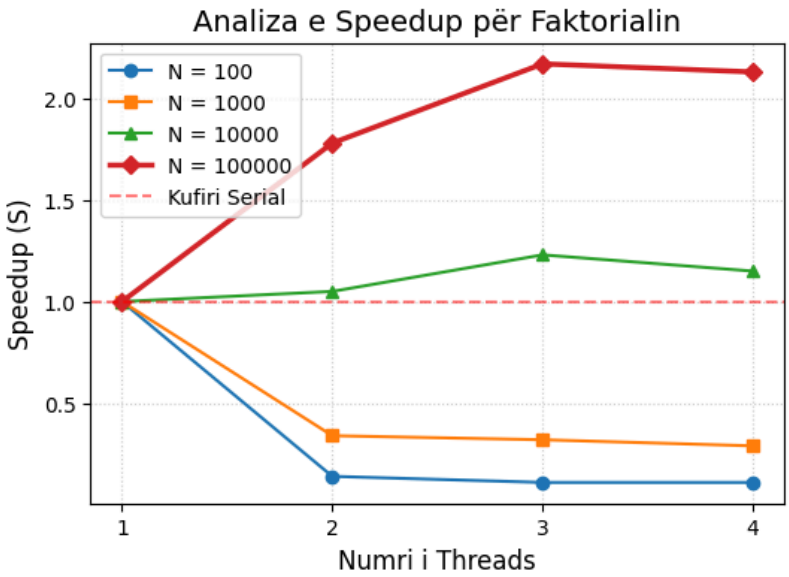


Figura 23 Grafiku i analizës së SpeedUp për Metoden 9

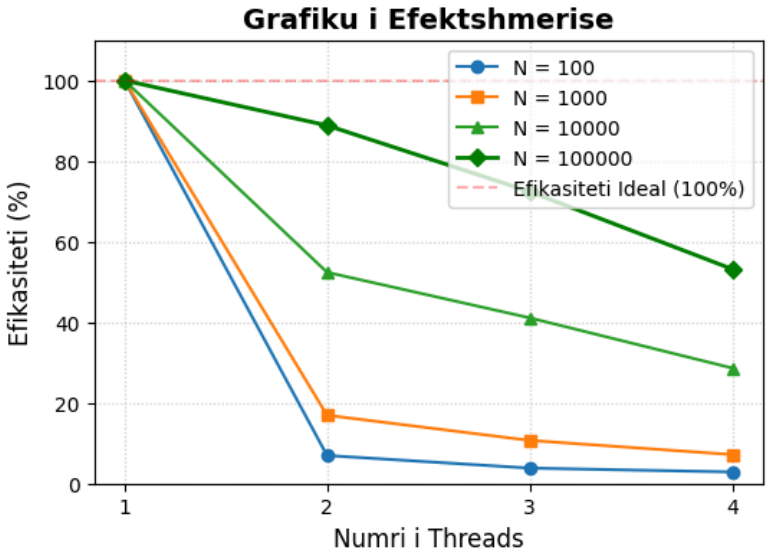


Figura 24 Grafiku i analizës së Efektshmerise për Metoden 9

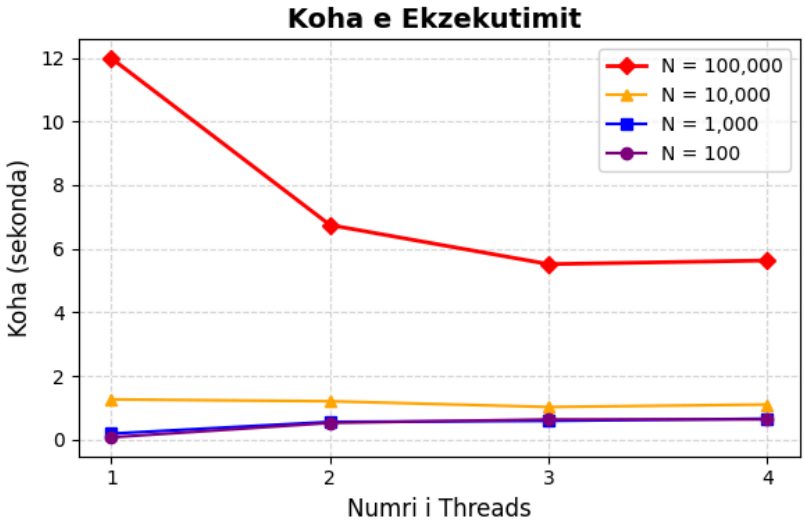


Figura 25 Grafiku i analizës së Kohes se Ekzekutimit për Metoden 9

2. Përfundime

Për vlerat e vogla të N (100 dhe 1,000), koha e ekzekutimit **rritet** me shtimin e threads në vend që të ulet. Kjo tregon se koha që harxhon sistemi për të krijuar dhe sinkronizuar threads është më e madhe se vetë puna llogaritëse.Vetëm te N = 100,000 vërejmë një rënie të ndjeshme të kohës (nga **11.9s** në **5.5s**). Kjo vërteton se paralelizmi kërkon një "ngarkesë kritike" pune për të qenë efektiv.
Pika e Ngopjes: Te N = 100,000, koha ulet deri në 3 threads, por pëson një rritje të lehtë te 4 threads. Kjo sugjeron që bërthama e katërt shton më shumë vonesa komunikimi sesa fuqi përpunuese.

Speedup Negativ (S < 1): Për ëN = 100ë dhe ëN = 1,000ë, Speedup-i është shumë nën vlerën 1.0 (deri në **0.11**). Kjo quhet "shkallëzim negativ" dhe ndodh kur kostoja e paralelizmit dominon mbi llogaritjen.

Për të gjitha vlerat e N, efektshmëria bie me shtimin e numrit të threads. Kjo është normale, pasi me rritjen e numrit threads, harxhohet më shumë kohë për t'i organizuar ata sesa për të bërë punën.

3. Shtojca 9

```
#include <iostream>
#include <omp.h>
#include <iomanip>
#include <vector>

using namespace std;
#define LOAD_LOOP 20000

int main() {
    int n_values[] = { 100, 1000, 10000, 100000 };
    int thread_counts[] = { 1, 2, 3, 4 };

    cout << left << setw(8) << "N"
         << setw(10) << "Threads"
         << setw(15) << "Koha (s)"
         << setw(12) << "Speedup"
         << setw(15) << "Efikasiteti"
         << "Masa (%)" << endl;

    for (int n : n_values) {
        double t_serial = 0;

        for (int p : thread_counts) {
            omp_set_num_threads(p);

            double start_time = omp_get_wtime();

            for (int k = 0; k < LOAD_LOOP; k++) {
                long double fact = 1.0;

                #pragma omp parallel for reduction(*:fact)
                for (int j = 1; j <= n; j++) {
                    fact *= j;
                }
            }

            double end_time = omp_get_wtime();
            double time_taken = end_time - start_time;

            if (p == 1) {
                t_serial = time_taken;
            }

            double speedup = t_serial / time_taken;
            double efficiency = speedup / p;
            double efficiency_percent = efficiency *
100.0;

            cout << left << setw(8) << n
                 << setw(10) << p
                 << fixed << setprecision(6) << setw(15) <<
time_taken
                 << setprecision(2) << setw(12) << speedup
                 << setw(15) << efficiency
                 << setprecision(2) << efficiency_percent <<
" %" << endl;
        }
    }
    return 0;
}
```

12. Metoda 10 – Llogaritja e vlerës së π duke përdorur metodën e përbërë të trapezit

Të llogaritet vlera e π në $\int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4}$ duke përdorur e pëbërë të trapezit .

b) Të shkruhet kodi, por duke përdorur metodën e përbërë të Simpsonit .

1. Përshkrimi Metodës së përbëre të trapezit

Qëllimi është të llogarisim vlerën e π duke përdorur integralin:

$$\int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4} \qquad \Rightarrow \qquad \pi = 4 * \int_0^1 \sqrt{1-x^2}\,dx$$

Gjeometrikisht, kjo llogarit sipërfaqen e çerekut të rrethit . Duke e shumëzuar rezultatin me 4, gjen vlerën e plotë të pi-së. E ndan boshtin x nga 0 në 1 në miliona copa të vogla (trapezë). Sa më shumë copa (N), aq më e saktë del vlera e pi.Komanda kyçe është **#pragma omp parallel for reduction(+:sum).**
reduction: Siguron që secili thread të mbledhë pjesën e vet dhe në fund të bashkohen të gjitha në një shumë totale pa gabime.

N	Threads	Koha (s)	Speedup (X)	Efikasiteti (%)	PI e Llogaritur
10000000	1	0.11800	1.00	100.00	3.1415926536
10000000	2	0.06600	1.79	89.39	3.1415926536
10000000	4	0.02900	4.07	101.72	3.1415926536
10000000	6	0.02400	4.92	81.94	3.1415926536
10000000	8	0.01600	7.37	92.19	3.1415926536
10000000	12	0.01700	6.94	57.84	3.1415926536
100000000	1	0.95300	1.00	100.00	3.1415926536
100000000	2	0.46700	2.04	102.03	3.1415926536
100000000	4	0.25300	3.77	94.17	3.1415926536
100000000	6	0.17700	5.38	89.74	3.1415926536
100000000	8	0.14300	6.66	83.30	3.1415926536
100000000	12	0.09300	10.25	85.39	3.1415926536
500000000	1	4.66100	1.00	100.00	3.1415926536
500000000	2	2.33100	2.00	99.98	3.1415926536
500000000	4	1.21200	3.85	96.14	3.1415926536
500000000	6	0.82300	5.66	94.39	3.1415926536
500000000	8	0.64200	7.26	90.75	3.1415926536
500000000	12	0.63100	7.39	61.56	3.1415926536

Tabela 9. Koha e ekzekutimit, Efektshmerise dhe Speedup i metodës 10

2. Shtojca 10

```
#include <iostream>
#include <omp.h>
#include <cmath>
```

```cpp
#include <iomanip>

using namespace std;
double f(double x) {
    return sqrt(1.0 - x * x);
}

int main() {
    long long n_values[] = { 10000000, 100000000, 500000000 };
    int num_experiments = 3;

    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_threads_configs = 6;
    cout << left << setw(12) << "N"
         << setw(10) << "Threads"
         << setw(14) << "Koha (s)"
         << setw(14) << "Speedup (X)"
         << setw(18) << "Efikasiteti (%)"
         << "PI e Llogaritur" << endl;

    for (int exp = 0; exp < num_experiments; exp++) {
        long long n = n_values[exp];

        double a = 0.0, b = 1.0;
        double h = (b - a) / n;
        double t_serial = 0;

        for (int i = 0; i < num_threads_configs; i++) {

            int t = threads_list[i];

            omp_set_num_threads(t);
            double start = omp_get_wtime();
            double sum = 0.0;

            #pragma omp parallel for reduction(+:sum)
            for (long long j = 1; j < n; j++) {
                double x = a + j * h;
                sum += f(x);
            }
            double integral = h * (0.5 * (f(a) + f(b)) + sum);
            double pi_calc = 4.0 * integral;
            double end = omp_get_wtime();
            double time_taken = end - start;

            if (time_taken < 1e-9) time_taken = 1e-9;

            if (t == 1) {
                t_serial = time_taken;
            }
            double speedup = t_serial / time_taken;
            double efficiency = (speedup / t) * 100.0;

            cout << left << setw(12) << n
                 << setw(10) << t
                 << fixed << setprecision(5) << setw(14) << time_taken
                 << fixed << setprecision(2) << setw(14) << speedup
                 << fixed << setprecision(2) << setw(18) << efficiency
                 << fixed << setprecision(10) << pi_calc << endl;
        }
```

```
    }
    cin.get();
    return 0;
}
```

3. Përshkrimi Metodës së përbëre të Simpsonit

Ndryshe nga Metoda e Trapezit, e cila përafron funksionin me vija të drejta (polinome të rendit 1), Metoda e Simpsonit përdor **parabola** (polinome të rendit 2) për të lidhur pikat. Kjo sjell një saktësi shumë më të lartë për funksione të lakuara si rrethi.

Dallimi kryesor në kod: Duhet të mbledhim dy shuma të ndara:

1. Shuma e elementeve me indeks tek (x_1, x_3...) shumëzohet me 4.
2. Shuma e elementeve me indeks çift (x_2, x_4...) shumëzohet me 2.

Çfarë ndryshon nga Trapezi?

Dy Shuma: Në vend të një sum të vetëm, kemi sum_odd dhe sum_even.

- **sum_odd** mbledh f(x_1), f(x_3), f(x_5)...
- **sum_even** mbledh f(x_2), f(x_4), f(x_6)...

Përdorim cikle me hapin i += 2 për të shmangur kontrollin e kushtëzuar if (i % 2 != 0), i cili do të ngadalësonte procesorin.Kam përdorur direktivën nowait midis dy cikleve for. Kjo lejon që threads të cilët përfundojnë llogaritjen e indekseve teke të mos presin në barrierë, por të vazhdojnë menjëherë me llogaritjen e indekseve çift, duke rritur efikasitetin e përdorimit të CPU-së.

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(x_0) + 4\sum_{i=1,3,5,}^{N-1} f(x_i) + 2\sum_{i=2,4,6..,}^{N-2} f(x_i) + f(x_N)\right]$$

$$I \approx \frac{h}{3} * (\, f(a) + f(b) + 4\ (\text{Shuma e indekseve tek}) + 2\ (\text{Shuma e indekseve cift}))$$

N	Threads	Koha (s)	Speedup (X)	Efikasiteti (%)	PI e Llogaritur
10000000	1	0.09100	1.00	100.00	3.1415926536
10000000	2	0.05300	1.72	85.85	3.1415926536
10000000	4	0.02400	3.79	94.79	3.1415926536
10000000	6	0.01800	5.06	84.26	3.1415926536
10000000	8	0.01600	5.69	71.09	3.1415926536
10000000	12	0.03000	3.03	25.28	3.1415926536
100000000	1	0.77300	1.00	100.00	3.1415926536
100000000	2	0.39500	1.96	97.85	3.1415926536
100000000	4	0.20600	3.75	93.81	3.1415926536
100000000	6	0.15300	5.05	84.20	3.1415926536
100000000	8	0.11800	6.55	81.89	3.1415926536
100000000	12	0.10100	7.65	63.78	3.1415926536
500000000	1	4.00800	1.00	100.00	3.1415926536

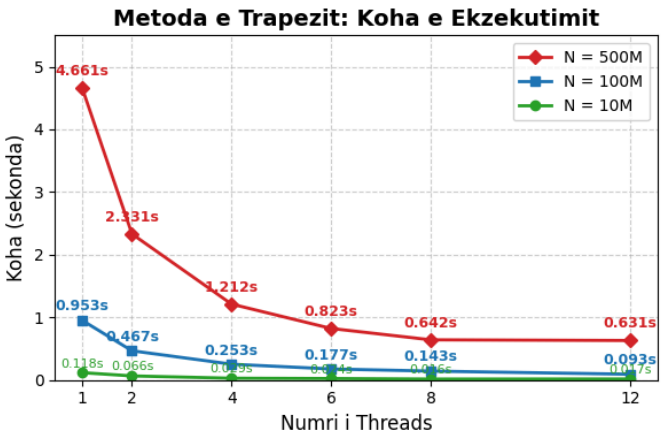
```
500000000   2         3.05000       1.31          65.70              3.1415926536
500000000   4         1.56200       2.57          64.15              3.1415926536
500000000   6         1.04500       3.84          63.92              3.1415926536
500000000   8         0.77000       5.21          65.06              3.1415926536
500000000   12        0.53700       7.46          62.20              3.1415926536
-------------------------------------------------------------------------------
```

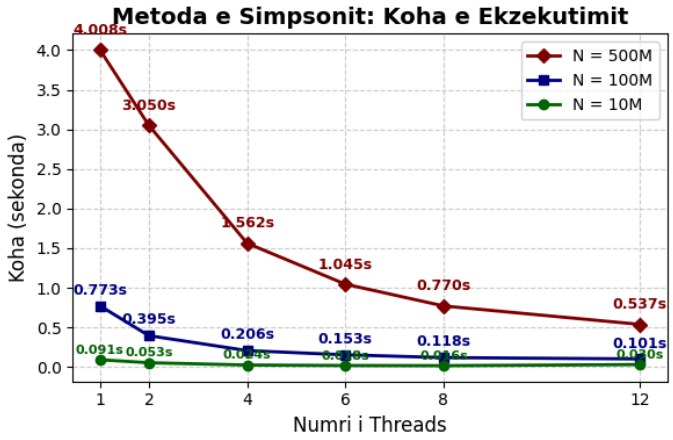
Tabela 10. Tabela e llogaritjes për kohën dhe speed up në llogaritjen e pi-së me anë të metodës së Simpsonit

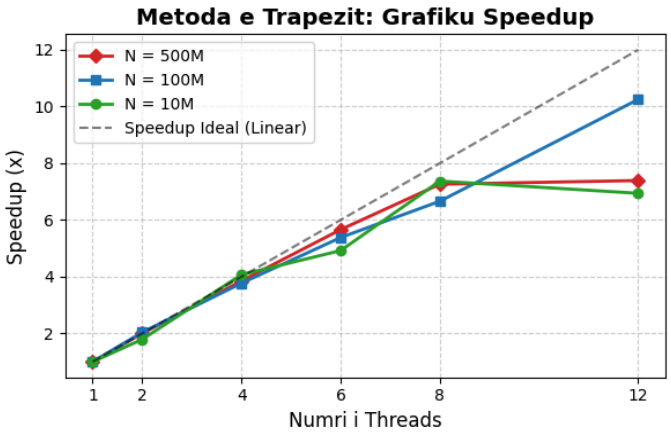
4. Shtojca 11

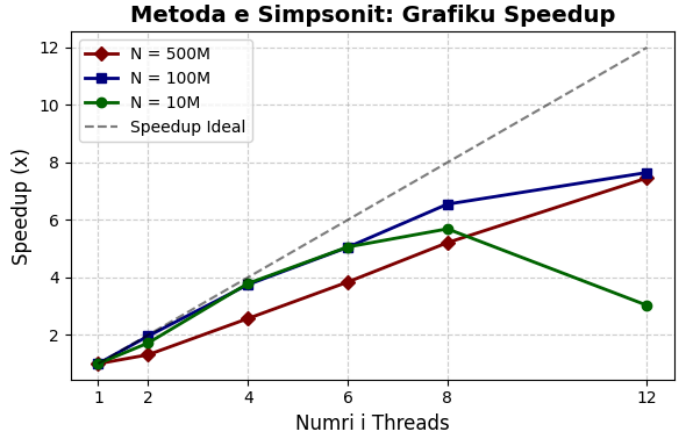
```
#include <iostream>
#include <omp.h>
#include <cmath>
#include <iomanip>
using namespace std;
// Funksioni matematikor: y = sqrt(1 - x^2)
double f(double x) {
    return sqrt(1.0 - x * x);
}
int main() {
    long long n_values[] = { 10000000, 100000000, 500000000 };
    int num_experiments = 3;
    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_threads_configs = 6;

    cout << left << setw(12) << "N"
         << setw(10) << "Threads"
         << setw(14) << "Koha (s)"
         << setw(14) << "Speedup (X)"
         << setw(18) << "Efikasiteti (%)"
         << "PI e Llogaritur" << endl;

    for (int exp = 0; exp < num_experiments; exp++) {
        long long n = n_values[exp];
        // Metoda e Simpsonit kërkon N çift.
        if (n % 2 != 0) n++;
        double a = 0.0, b = 1.0;
        double h = (b - a) / n;
        double t_serial = 0;

        for (int i = 0; i < num_threads_configs; i++)
            int t = threads_list[i]
            omp_set_num_threads(t);
            double start = omp_get_wtime();

            double sum_odd = 0.0;  // Koeficienti 4
            double sum_even = 0.0; // Koeficienti 2

            #pragma omp parallel
            {
                // FAZA 1: Indekset TEKE (1, 3, 5...)
                // 'nowait' lejon threads të kalojnë te FAZA 2 pa pritur të tjerët
                #pragma omp for reduction(+:sum_odd) nowait
                for (long long j = 1; j < n; j += 2) {
                    sum_odd += f(a + j * h);
                }
                // FAZA 2: Indekset ÇIFT (2, 4, 6...)
                #pragma omp for reduction(+:sum_even)
```

```
            for (long long j = 2; j < n; j += 2) {
                sum_even += f(a + j * h);
            }
        }
        double integral = (h / 3.0) * (f(a) + f(b) + 4.0 * sum_odd + 2.0 * sum_even);
        double pi_calc = 4.0 * integral;
        double end = omp_get_wtime();
        double time_taken = end - start;
        if (time_taken < 1e-9) time_taken = 1e-9;
        if (t == 1) t_serial = time_taken;

        double speedup = t_serial / time_taken;
        double efficiency = (speedup / t) * 100.0;
        cout << left << setw(12) << n
             << setw(10) << t
             << fixed << setprecision(5) << setw(14) << time_taken
             << fixed << setprecision(2) << setw(14) << speedup
             << fixed << setprecision(2) << setw(18) << efficiency
             << fixed << setprecision(10) << pi_calc << endl;
    }
  }
  cin.get();
  return 0;
}
```









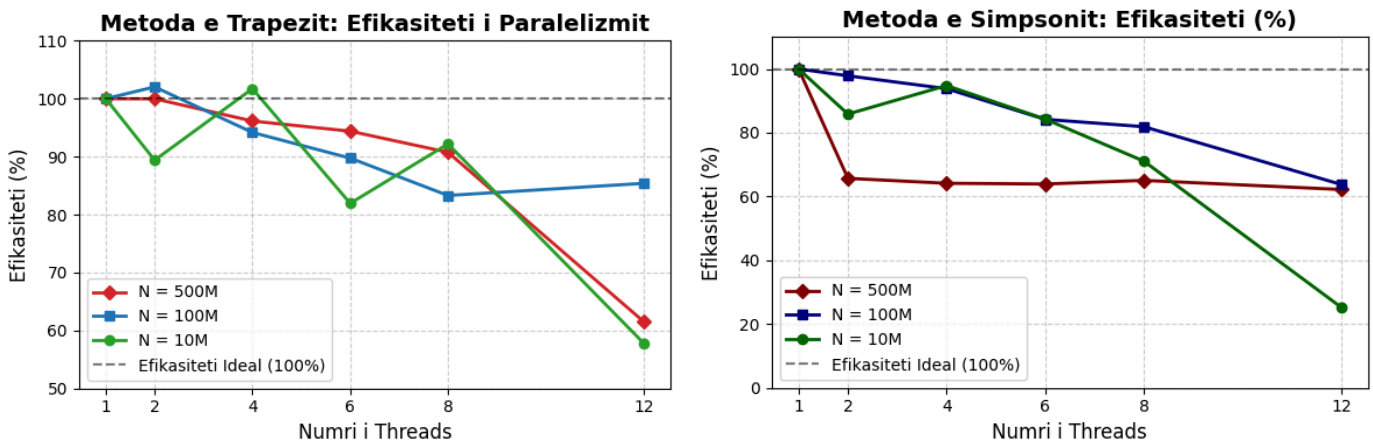


Figura 26 Krahasimi i Metodes se Trapezit me Metoden e Simpsonit

5. Përfundime

1. Performanca Seriale (1 Thread)

- **Simpsoni :** Për N=500M, Simpsoni fillon me **4.008s**.
- **Trapezi:** Për N=500M, Trapezi fillon me **4.661s**.
- **Analiza:** Çuditërisht, Simpsoni është **më i shpejtë** në ekzekutimin me 1 thread. Ndonëse Simpsoni ka më shumë veprime matematike (shumëzime me 4 dhe 2), struktura e tij në këtë rast është ekzekutuar më me efikasitet nga procesori .

2. Shkallëzueshmëria - Forma e Kurbës

- **Trapezi :** Kur kalon nga 1 → 2 threads, koha bie nga **4.66s —>2.33s**. Kjo është përgjysmim perfekt (**Speedup ideal 2x**).
- **Simpsoni:** Kur kalon nga 1 → 2 threads, koha bie nga **4.00s —> 3.05s**. Ky është një përmirësim modest (**Speedup 1.3x**).
- **Analiza:** Metoda e Trapezit shkallëzohet shumë më mirë. Për shkak se akseson memorien në rradhë (1, 2, 3...), OpenMP e ndan punën pa asnjë konflikt. Simpsoni, me hapin e tij i+=2, krijon pak më shumë vonesa në komunikimin midis threads dhe memories në fillim.

Grafikët e Speedup :

- **Metoda e Trapezit** është me e shkallëzueshmëris. Ajo arrin speedup mbi 10x dhe qëndron afër ideales në çdo skenar. Është algoritmi më "i shëndetshëm" për paralelizëm.
- **Metoda e Simpsonit** është e brishtë. Ajo vuan nga rënie drastike e performancës në ngarkesa të vogla (vija jeshile bie) dhe nuk arrin kurrë të njëjtin speedup maksimal si Trapezi, për shkak të qasjes jo-lineare në memorie.

Grafikët e Efektshmërisë :

Trapezi (Vija Blu - 100M): Ne 2 Threads efikasiteti është **mbi 100%** (102%).

- Tregon se metoda e Trapezit është aq e lehtë për t'u ndarë, saqë ndarja e bën procesorin të punojë më

mirë sesa kur punon vetëm.

Simpsoni (Vija Blu - 100M): Nis te 100% dhe bie menjëherë. Te 12 Threads është rreth **63%**.

- Simpsoni nuk arrin kurrë të kalojë 100%

Për probleme të vogla (N=10M), Metoda e Simpsonit është jashtëzakonisht jo-efikase me shumë threads (vetëm **25%** efikasitet). Kjo sugjeron se për këtë metodë nuk duhet të përdorim kurrë më shumë se 4-6 threads për N të vogla.

Metoda e Trapezit arrin të ruajë efikasitet mbi **90%** në shumicën e konfigurimeve, madje duke kaluar edhe **100%** në raste specifike (Super-linear efficiency).

13. Metoda 11 - Përafrimi algorimit të prodhimit skalar për gjetjen e vlerës së polinomit në një pikë

1. Përshkrimi Metodës

Një polinom i shkallës N, P(x) =$a_0 + a_1x + a_2x^2 + \ldots\ldots + a_nx^n$ mund të shihet si **prodhimi**

skalar i dy vektorëve:
Vektori i Koeficientëve: A = [a_0,a_1, , , , a_n]

Vektori i Fuqive: P= [1, x, x^2, ... , x^n] . Vlera e polinomit është: $\sum_{i=0}^{N} A[i]\,P[i]$.

Si funksionon kodi ?

Algoritmi ndahet në tre zona paralele të pavarura për të maksimizuar shpejtësinë:

Inicializimi i A: Mbushet vektori i koeficientëve në paralel.

Llogaritja e P: Llogariten fuqitë e x në mënyrë të pavarur.

Reduktimi : Kryhet shumëzimi element-për-element dhe mbledhja finale duke përdorur klauzolën #pragma omp parallel for reduction(+:sum).

Përdoret alokim dinamik (new double[]) për të përballuar vlera të mëdha të N (deri në 50 milionë elemente), të cilat nuk do të nxinin në Stack-un e memories

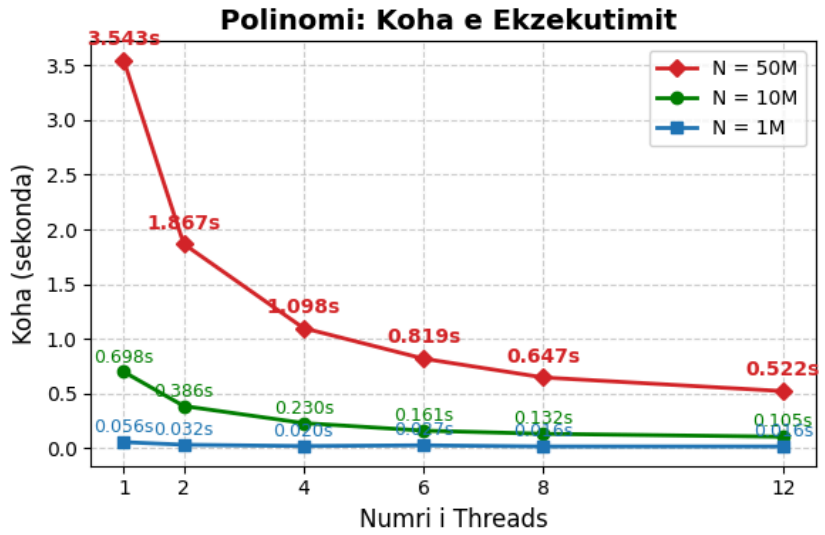


Figura 27 Grafiku i analizës së Kohes se Ekzekutimit te Metodes 11

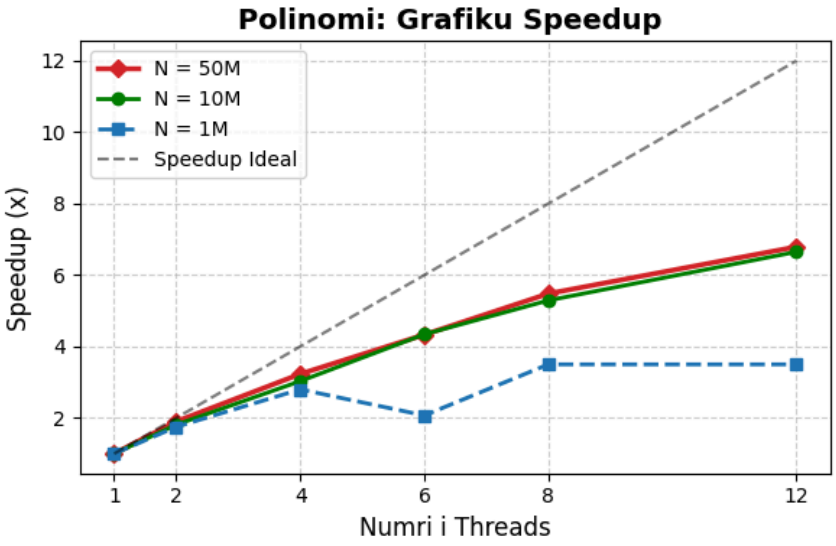


Figura 28 Grafiku i analizës së Speedup për Metoden 11

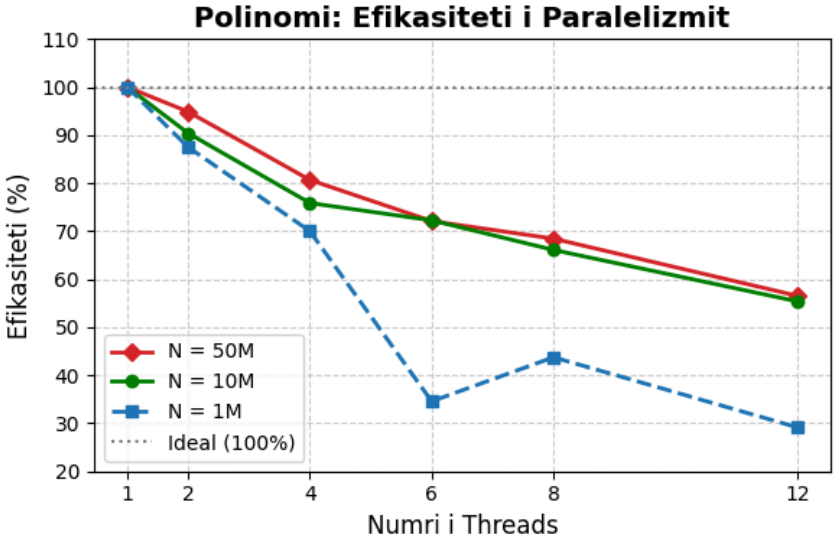


Figura 29 Grafiku i analizës së Efektshmerise për Metoden 11

2. Përfundime

Në këtë metode, ne aplikuam paralelizmin për të llogaritur vlerën e një polinomi të shkallës N duke përdorur metodën e vektorëve (Prodhimi Skalar). Testet u kryen për N=1M, 10M dhe 50M.Arritëm në këto përfundime kryesore:Metoda e Polinomit është Memory Bound (shumë memorie, pak llogaritje).Speedup-i maksimal për N=50M arriti në 6.79x me 12 Threads, dhe efikasiteti ra në 56%.Procesori është shumë i shpejtë në shumëzimin e numrave, por ai duhet të presë që RAM-i t'i sjellë të dhënat nga vektorët gjigantë A dhe P. Kur 12 threads kërkojnë të dhëna njëkohësisht, "gryka" e memories (Memory Bandwidth) bllokohet, duke penguar arritjen e një speedup-i linear (12x).Për algoritmet që manipulojnë vektorë të mëdhenj, shpejtësia e ekzekutimit diktohet jo vetëm nga fuqia llogaritëse e procesorit, por kryesisht nga shpejtësia e transferimit të të dhënave nga memoria RAM."

3. Shtojca 12

```
#include <iostream>
#include <cmath>
#include <omp.h>
#include <iomanip>

using namespace std;

int main() {
    long long n_values[] = { 1000000, 10000000, 50000000 };
    int num_experiments = 3;
    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_threads_configs = 6;

    double x = 0.9999999;

    cout << left << setw(12) << "N"
         << setw(10) << "Threads"
         << setw(14) << "Koha (s)"
         << setw(14) << "Speedup (X)"
         << setw(18) << "Efikasiteti (%)"
         << "Vlera e Polinomit" << endl;

    for (int exp = 0; exp < num_experiments; exp++) {
        long long N = n_values[exp];

        double* A = new double[N];
        double* P_x = new double[N];

        double t_serial = 0;

        for (int i = 0; i < num_threads_configs; i++) {

            int t = threads_list[i];
            omp_set_num_threads(t);

            double start = omp_get_wtime();
            #pragma omp parallel for
            for (long long j = 0; j < N; j++) {
                A[j] = 1.000001;
            }
            #pragma omp parallel for
```

```
            for (long long j = 0; j < N; j++) {
                P_x[j] = pow(x, j);
            }
            double vlera_polinomit = 0.0;
            #pragma omp parallel for reduction(+:vlera_polinomit)
            for (long long j = 0; j < N; j++) {
                vlera_polinomit += A[j] * P_x[j];
            }

            double end = omp_get_wtime();
            double time_taken = end - start;

            if (time_taken < 1e-9) time_taken = 1e-9;

            if (t == 1) t_serial = time_taken;

            double speedup = t_serial / time_taken;
            double efficiency = (speedup / t) * 100.0;

            cout << left << setw(12) << N
                 << setw(10) << t
                 << fixed << setprecision(5) << setw(14) << time_taken
                 << fixed << setprecision(2) << setw(14) << speedup
                 << fixed << setprecision(2) << setw(18) << efficiency
                 << scientific << setprecision(5) << vlera_polinomit << endl;
        }
        delete[] A;
        delete[] P_x;
    }
    cin.get();
    return 0;
}
```

14. Metoda 12 -Metoda e Fuqisë

1. Përshkrimi Metodës

Metoda e Fuqisë është një algoritëm iterativ që përdoret për të gjetur vlerën vetiake më të madhe në vlerë absolute dhe vektorin vetiak përkatës të një matrice A. Procesi bazohet në formulën rekursive:

$$x_{k+1} = \frac{Ax_k}{||Ax_k||_\infty}$$

Si funksionon metoda?

Shumëzimi Matricë-Vektor (y = Ax) Paralelizmi bëhet duke ndarë rreshtat e matricës midis threads-ave. Çdo thread llogarit një pjesë të vektorit y.

Duhet gjetur elementi maksimal absolut në vektorin y.

Përdoret #pragma omp parallel for reduction(max:max_val) për të gjetur maksimumin globa

Normalizimi (x = y / λ):Vektori y pjesëtohet me vlerën maksimale (λ) për të marrë vektorin e ri x. Ky hap është teresisht paralel pasi çdo element llogaritet në mënyrë të pavarur.

N	Threads	Koha (s)	Speedup (X)	Efikasiteti (%)
5000	1	1.8770	1.00	100.00
5000	2	0.9550	1.97	98.27
5000	4	0.7600	2.47	61.74
5000	6	0.5710	3.29	54.79
5000	8	0.4870	3.85	48.18
5000	12	0.3770	4.98	41.49
-> Rezultati (Lambda) per N=5000 ishte: 5004.61765				
10000	1	7.2730	1.00	100.00
10000	2	3.8600	1.88	94.21
10000	4	2.9260	2.49	62.14
10000	6	2.1590	3.37	56.14
10000	8	1.7530	4.15	51.86
10000	12	1.4270	5.10	42.47
-> Rezultati (Lambda) per N=10000 ishte: 10005.28320				
20000	1	28.9310	1.00	100.00
20000	2	15.2030	1.90	95.15
20000	4	11.0110	2.63	65.69
20000	6	8.4610	3.42	56.99
20000	8	6.8930	4.20	52.46
20000	12	5.7100	5.07	42.22
-> Rezultati (Lambda) per N=20000 ishte: 20005.95906				

Tabela 11. Koha e ekzekutimit, Efektshmerise dhe Speedup i metodës 12

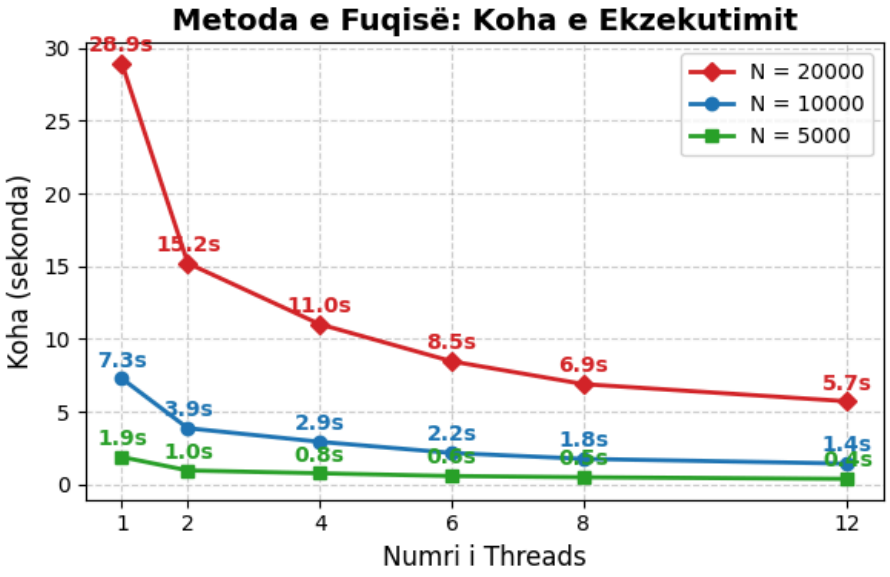


Figura 30 Grafiku i analizës se Kohes se Ekzekutimit per Metoden 12

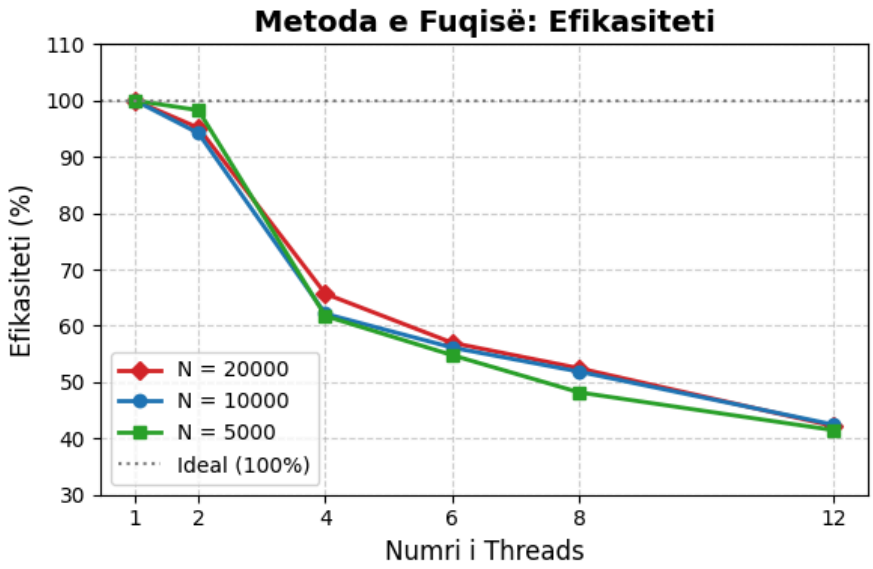


Figura 31 Grafiku i analizës së Efektshmerise për Metoden 12

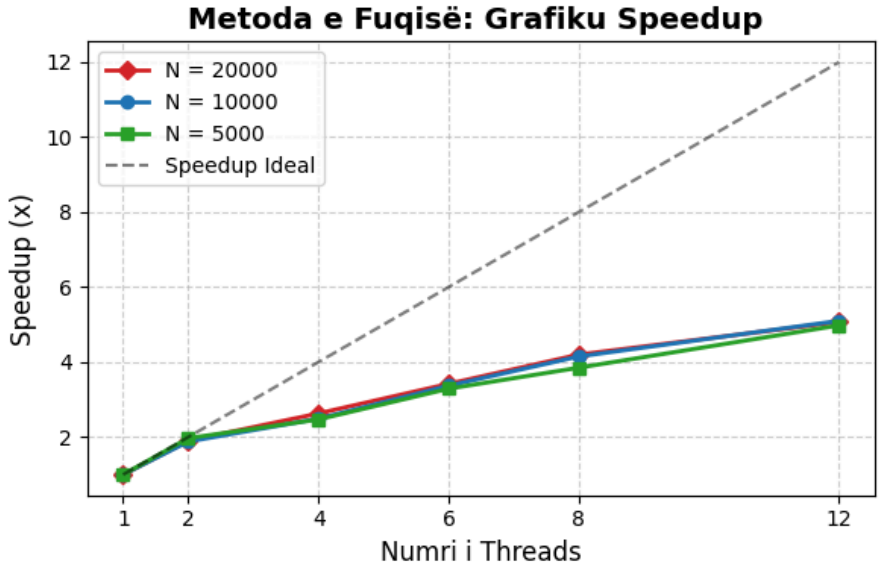


Figura 32 Grafiku i analizës së Speedup per Metoden 12

2. Përfundime

Pavarësisht se përdorëm 12 threads, Speedup-i maksimal u stabilizua rreth vlerës **5.1x** (jo 12x).Efikasiteti ra në mënyrë të vazhdueshme ndërsa shtonim threads, duke përfunduar në rreth **42%**. Kjo tregon se shtimi i pakufizuar i threads nuk sjell rritje lineare të shpejtësisë për këtë lloj algoritmi.

Ndryshe nga metodat e thjeshta (si Trapezi), Metoda e Fuqisë është **Iterative** dhe kërkon sinkronizim të shpeshtë. Në çdo iteracion, threads-at duhet të ndalojnë dhe të presin njëri-tjetrin **3 herë**:

1. Pas shumëzimit Matricë-Vektor (për të kompletuar vektorin y).
2. Pas gjetjes së Maksimumit (Reduction).
3. Pas Normalizimit të vektorit.
   Kjo "pritje" e detyruar është arsyeja pse procesori nuk mund të punojë me kapacitet 100% gjatë gjithë kohës.

Edhe pse arritëm ta ulim kohën e ekzekutimit nga **29 sekonda në 5.7 sekonda** (për N=20000), nevoja për komunikim dhe sinkronizim të vazhdueshëm midis threads-ave e bën të pamundur arritjen e një efikasiteti perfekt linear."

3. Shtojca 13

```
#include <iostream>
#include <vector>
#include <cmath>
#include <omp.h>
#include <iomanip>

using namespace std;
// Funksion qe gjeneron elementin A[i][j]
inline double get_A(int i, int j) {
    if (i == j) return i + 1.0;
    return 1.0 / (abs(i - j) + 1.0);
}
int main() {
    int n_values[] = { 5000, 10000, 20000 };
    int num_experiments = 3;

    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_threads_configs = 6;

    const int max_iter = 20;
    const double tol = 1e-6;

    cout << left << setw(10) << "N"
         << setw(10) << "Threads"
         << setw(14) << "Koha (s)"
         << setw(14) << "Speedup (X)"
         << "Efikasiteti (%)" << endl;

    for (int exp = 0; exp < num_experiments; exp++) {
        int n = n_values[exp];

        vector<double> x(n, 1.0);
        vector<double> y(n);

        double t_serial = 0;
        double final_lambda = 0;

        for (int t_idx = 0; t_idx < num_threads_configs; t_idx++) {
            int num_threads = threads_list[t_idx];
            omp_set_num_threads(num_threads);

            fill(x.begin(), x.end(), 1.0);
            double lambda_new = 0;

            double start_time = omp_get_wtime();
```

```
            for (int iter = 0; iter < max_iter; iter++) {

                #pragma omp parallel for schedule(static)
                for (int i = 0; i < n; i++) {
                    double sum = 0.0;
                    for (int j = 0; j < n; j++) {
                        sum += get_A(i, j) * x[j];
                    }
                    y[i] = sum;
                }
                double max_val = 0.0;
                #pragma omp parallel for reduction(max:max_val)
                for (int i = 0; i < n; i++) {
                    double val = fabs(y[i]);
                    if (val > max_val) max_val = val;
                }
                lambda_new = max_val;
                #pragma omp parallel for schedule(static)
                for (int i = 0; i < n; i++) {
                    x[i] = y[i] / lambda_new;
                }
            }

            double end_time = omp_get_wtime();
            double time_taken = end_time - start_time;
            if (time_taken < 1e-9) time_taken = 1e-9;

            if (num_threads == 1) t_serial = time_taken;
            double speedup = t_serial / time_taken;
            double efficiency = (speedup / num_threads) * 100.0;

            final_lambda = lambda_new;
            cout << left << setw(10) << n
                 << setw(10) << num_threads
                 << fixed << setprecision(4) << setw(14) << time_taken
                 << fixed << setprecision(2) << setw(14) << speedup
                 << fixed << setprecision(2) << efficiency << endl;
        }
        cout << "   -> Rezultati (Lambda) per N=" << n << " ishte: " << fixed <<
setprecision(5) << final_lambda << endl;
    }
    cin.get();
    return 0;
}
```

15. Metoda 13 -Metoda e Përgjysmimit

1. Përshkrimi Metodës

Metoda e Përgjysmimit është një algoritëm numerik për gjetjen e rrënjës së një ekuacioni f(x) = 0.

1. **Intervali Fillestar:** Nisemi me një interval [a, b] ku jemi të sigurt që ndodhet te pakten nje rrënje. Kushti është që funksioni të ketë shenja të kundërta në skaje: $f(a) * f(b) < 0$.

2. **Pika e Mesit:** Gjejmë mesin e intervalit: $c = \frac{a+b}{2}$

Testimi:

1. Nëse f(c) = 0, gjetëm rrënjën.
2. Nëse $f(a) * f(c) < 0$, rrënja është në gjysmën e majtë [a, c]. Intervali i ri bëhet [a, c]
3. Nëse $f(a) * f(c) < 0$, rrënja është në gjysmën e djathtë [c, b]. Intervali i ri bëhet [c, b].

Procesi përsëritet derisa intervali të bëhet jashtëzakonisht i vogël.

Si funksionon paralelizimi ?

Ndryshe nga metoda klasike që e ndan intervalin vetëm në dy pjesë, ky kod e ndan intervalin aktual [a, b] në aq pjesë sa ka Threads **(NUM_Threads)**.Direktiva **#pragma omp parallel for** krijon threads dhe i cakton secilit një indeks **i.**

- Secili thread llogarit kufijtë e vet lokalë: x0 (fillimi) dhe x1 (fundi).
- Brenda këtij nën-intervali të vogël, thread-i bën një kontroll "mini-përgjysmimi". Ai gjen mesin c të pjesës së tij dhe kontrollon nëse rrënja ndodhet aty.
  Dallimi: Serial vs Paralel
- **Serial:** Në çdo hap, intervali zvogëlohet me faktorin **2** (përgjysmohet).
- **Paralel:** Në çdo hap (iteracion të ciklit **while)**, intervali zvogëlohet me një faktor shumë më të madh (afërsisht 2 * NUM_Threads).

Logjika e Algoritmit tonë :
Inizializimi:

- Përcakton intervalin fillestar [a, b] (-1 deri në 0).
- Kontrollon kushtin e konvergjencës $f(a) * f(b) < 0$. Nëse shenjat janë të njëjta, ndalon sepse nuk ka siguri që ka rrënjë aty.

Cikli While :

- Kodi vazhdon të ekzekutohet derisa gabimi (eps) të jetë më i vogël se toleranca e përcaktuar (tol = 0.00000001)

Paralelizimi:

- Ndryshe nga metoda klasike që e ndan intervalin vetëm në dy pjesë (në mes), ky kod e ndan intervalin aktual [a, b] në aq pjesë sa ka Threads (**NUM_Threads**).
- h = (b - a) / NUM_Threads;: Llogarit gjerësinë e nën-intervalit për çdo thread.
- #pragma omp parallel for: Krijon threads dhe secili merr përsipër të analizojë një nën-interval specifik (x0 deri në x1).

Funksioni yne : $f(x) = sin(x) + \frac{sin(2x)}{2} + \frac{sin(3x)}{3} + \ldots + \frac{sin(1000x)}{1000}$

Threads	Koha (s)	Speedup (X)	Iteracione	Rrenja
1	0.22100	1.00	1000	1.55000
2	0.01600	13.81	25	2.14075
4	0.00400	55.25	13	2.14075
6	0.00700	31.57	10	2.14075
8	0.00700	31.57	9	2.14075
12	0.00400	55.25	7	2.14075

Tabela 12. Koha e ekzekutimit, Speedup dhe numri iteracioneve per metodën 13

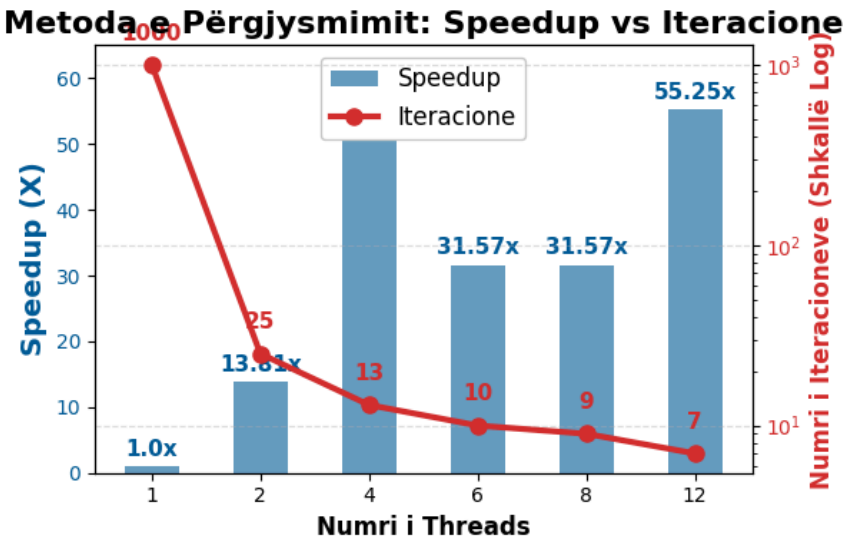


Figura 33 Grafiku i analizës per Metoden e Pergjysmimit

2. Përfundime

Metoda Seriale (1 Thread): E ndante intervalin në 2 pjesë. Pas **1000 iteracioneve** (limiti i vendosur), ajo ende nuk arriti të konvergjonte te rrënja e saktë, duke dhënë një rezultat të gabuar (x ≈ 1.55).
Metoda Paralele (12 Threads): Duke e ndarë intervalin në **12 nën-intervale** në çdo hap, hapësira e kërkimit u ngushtua. Kjo lejoi gjetjen e saktë të rrënjës (x ≈ 2.14075) në vetëm **7 iteracione**.

Ne vumë re një Speedup prej **55x** me vetëm 4 threads.Ky rezultat, që teorikisht duket i pamundur (mbi 4x), shpjegohet me faktin se ne krahasuam një proces të gjatë e të pasuksesshëm (serial) me një proces ultra-të shkurtër e të saktë ne paralel

Për numrin e lartë të threads (4, 6, 8, 12), kohët e ekzekutimit ishin aq të vogla (**~0.004 sekonda**) saqë arritën kufirin e saktësisë së orës së sistemit. Kjo solli vlera identike të Speedup-it për konfigurime të ndryshme, duke treguar se për probleme të kësaj natyre, kemi arritur zgjidhjen e menjëhershme.

Ky testim tregoi se Paralelizmi është zgjidhja ideale për problemet e kërkimit të rrënjëve në funksione me kosto të lartë llogaritëse . Duke kaluar nga 'Bisection' (ndarje në 2) në 'Multisection' (ndarje në N), ne

arritëm ta kthenim një proces që dështoi pas 1000 hapash në një proces që u zgjidh saktë në vetëm 7 hapa."

3. Shtojca 14

```
#include <iostream>
#include <omp.h>
#include <cmath>
#include <iomanip>

using namespace std;

// FUNKSIONI: Shuma e nje Serie Fourier (Sinjal Valor)
// Matematikisht: f(x) = shuma(sin(k*x)/k) - 0.5
double f_real(double x) {
    double sum = 0.0;
    // Mbledhim 1000 harmonika te vales (ngarkese mesatare reale)
    for (int k = 1; k <= 1000; k++) {
        sum += sin(k * x) / k;
    }
    return sum - 0.5; // Kerkojme ku sinjali ka vleren 0.5
}
int main() {
    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_configs = 6;
    // Toleranca standarde inxhinierike
    const double tol = 1e-7;
    const int max_iter = 1000;
    cout << left << setw(10) << "Threads"
         << setw(14) << "Koha (s)"
         << setw(14) << "Speedup (X)"
         << setw(14) << "Iteracione"
         << "Rrenja" << endl;

    double t_serial = 0;

    for (int t_idx = 0; t_idx < num_configs; t_idx++) {
        int num_threads = threads_list[t_idx];

        // Intervali ku dime qe ka rrenje per kete funksion
        double a = 0.1;
        double b = 3.0;
        double eps = 1.0;
        int iter = 0;

        omp_set_num_threads(num_threads);

        double start_time = omp_get_wtime();

        while (eps > tol && iter < max_iter) {
            iter++;
            double h = (b - a) / num_threads;
            double current_a = a;
```

```
            bool found = false;

            #pragma omp parallel
            {
                // Optimizim: Nese u gjet, threads te tjere ndalojne
                if (!found) {
                    int id = omp_get_thread_num();
                    // Ndarja e intervalit ne N pjesë
                    double x0 = current_a + id * h;
                    double x1 = x0 + h;
                    // Llogaritja e funksionit real
                    double fx0 = f_real(x0);
                    double fx1 = f_real(x1);
                    // Kontrolli i Bolzanos
                    if (fx0 * fx1 < 0) {
                        #pragma omp critical
                        {
                            if (!found) { // Vetem i pari fiton te drejten te
ndryshoje a,b
                                a = x0;
                                b = x1;
                                found = true;
                            }
                        }
                    }
                }
            }
            eps = b - a;
        }
        double root = (a + b) / 2.0;
        double time_taken = omp_get_wtime() - start_time;

        // Siguri minimale per pjesetimin me zero
        if (time_taken < 1e-9) time_taken = 1e-9;

        if (num_threads == 1) t_serial = time_taken;
        double speedup = t_serial / time_taken;

        cout << left << setw(10) << num_threads
             << fixed << setprecision(5) << setw(14) << time_taken
             << fixed << setprecision(2) << setw(14) << speedup
             << setw(14) << iter
             << fixed << setprecision(5) << root << endl;
    }
    cin.get();
    return 0;
}
```

16. Metoda 14 -Algoritmi i renditjes Bubble Sort

1. Përshkrimi Metodës

Algoritmi klasik Bubble Sort krahason dhe shkëmben elementet fqinjë A[i] dhe A[i+1] duke përshkuar vargun nga fillimi në fund. Ky proces është serial, pasi krahasimi i radhës varet nga rezultati i shkëmbimit të mëparshëm.

Për të mundësuar paralelizmin, ne përdorim metodën **Teke-Çifte**. Ideja është të ndajmë procesin e renditjes në dy faza të pavarura që përsëriten:

1. **Faza Teke :** Krahasojmë dhe shkëmbejmë të gjitha çiftet me indeks tek (A[1], A[2]), (A[3], A[4]),,,. Të gjitha këto çifte janë të pavarura nga njëri-tjetri, prandaj mund të ekzekutohen në paralel nga threads të ndryshëm.
2. **Faza Çifte :** Krahasojmë dhe shkëmbejmë të gjitha çiftet me indeks çift (A[0], A[1]), (A[2], A[3]),,, Edhe këto janë të pavarura.

```
>>> TESTIMI ME N = 10000 ELEMENTE <<<
 Threads   Koha (s)       Speedup (X)
    1         0.4920         1.00
    2         0.6020         0.82
    4         0.7400         0.66
    6         0.7560         0.65
    8         0.8790         0.56
    12        1.0520         0.47

>>> TESTIMI ME N = 30000 ELEMENTE <<<
 Threads   Koha (s)       Speedup (X)
    1         4.0150         1.00
    2         3.4080         1.18
    4         3.2110         1.25
    6         3.0780         1.30
    8         3.4690         1.16
    12        4.0280         1.00

>>> TESTIMI ME N = 50000 ELEMENTE <<<
 Threads   Koha (s)       Speedup (X)
    1         11.0430        1.00
    2         7.8410         1.41
    4         7.3420         1.50
    6         10.2990        1.07
    8         10.3590        1.07
    12        10.7210        1.03
```

Tabela 13. Koha e ekzekutimit dhe Speedup per testimet e metodës 14

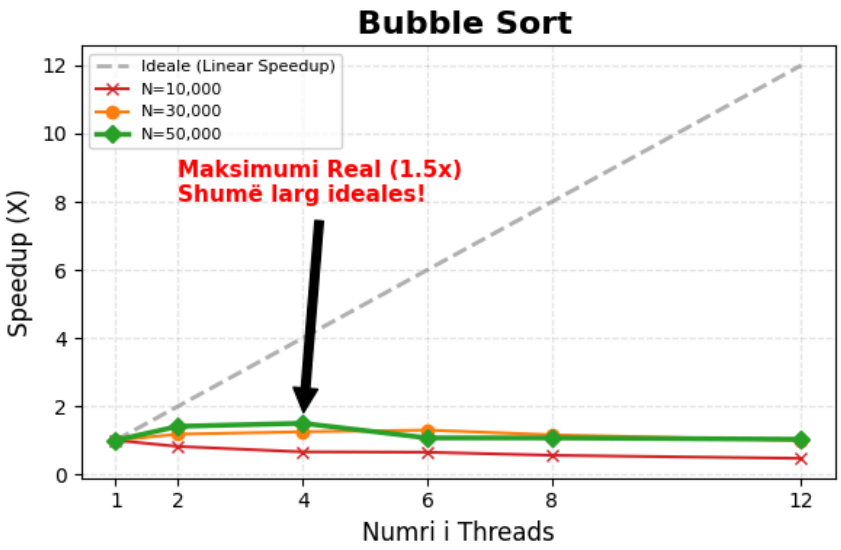


Figura 34 Grafiku i analizës së Speedup për Alogritmin e renditjes Bubble Sort

2. Përfundime

Dallimi vizual midis **Speedup Ideal** dhe Rezultatet Reale është shumë i madh. Ndërsa idealisht me 12 threads prisnim një shpejtësi afër 12x, maksimumi i arritur ishte vetëm **1.5x**.

- Kjo ndodh për shkak të natyrës së algoritmit *Odd-Even Sort*, i cili kërkon **sinkronizim të detyruar** pas çdo faze teke dhe çifte. Koha që threads harxhojnë duke pritur njëri-tjetrin dominon mbi kohën e punës efektive.

Për N=10,000 (vija e kuqe), grafiku tregon një **Speedup Negativ** (nën 1.0x).

- Sapo shtojmë threads (2, 4, 6...), koha e ekzekutimit rritet në vend që të ulet.
- Kjo vërteton se për ngarkesa të lehta, kostoja e menaxhimit të threads (krijimi, shkatërrimi dhe komunikimi) është më e lartë se vetë puna e renditjes. Në këtë rast, metoda seriale është më efikase.

Per N=50,000 (vija jeshile) arrin kulmin te **4 Threads** (1.5x Speedup) dhe më pas fillon të bjerë.

- Kjo tregon se shtimi i më shumë threads (6, 8, 12) nuk sjell përfitim. Përkundrazi, rritja e numrit të threads rrit trafikun në memorie dhe kohën e sinkronizimit, duke degraduar performancën.

3. Shtojca 15

```
#include <iostream>
#include <omp.h>
#include <vector>
#include <cstdlib>
#include <iomanip>
#include <ctime>

using namespace std;
void bubble_sort_odd_even(vector<int>& arr, int n, int num_threads) {
    omp_set_num_threads(num_threads);
```

```
    for (int i = 0; i < n; ++i) {
        // FAZA 1: Indekset Teke (1, 3, 5...)
        #pragma omp parallel for
        for (int j = 1; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
        // FAZA 2: Indekset Çifte (0, 2, 4...)
        #pragma omp parallel for
        for (int j = 0; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int main() {
    // 3 Nivele të ndryshme të N
    int N_values[] = { 10000, 30000, 50000 };
    int num_N_tests = 3;

    int threads_list[] = { 1, 2, 4, 6, 8, 12 };
    int num_configs = 6;

    srand(time(0));
    for (int n_idx = 0; n_idx < num_N_tests; n_idx++) {
        int N = N_values[n_idx];

        cout << left << setw(10) << "Threads"
             << setw(15) << "Koha (s)"
             << setw(15) << "Speedup (X)" << endl;
        cout << "-------------------------------------------------\n";

        // Gjenerojmë të dhënat vetëm një herë për këtë N
        vector<int> data(N);
        for(int i=0; i<N; i++) data[i] = rand() % 100000;

        double t_serial = 0;

        for(int t_idx = 0; t_idx < num_configs; t_idx++) {
            int t = threads_list[t_idx];

            vector<int> test_arr = data;

            double start = omp_get_wtime();
            bubble_sort_odd_even(test_arr, N, t);
            double end = omp_get_wtime();

            double duration = end - start;

            if (t == 1) t_serial = duration;
            double speedup = t_serial / duration;

            cout << left << setw(10) << t
                 << fixed << setprecision(4) << setw(15) << duration
                 << fixed << setprecision(2) << setw(15) << speedup << endl;
        }
```

```
        cout << "\n";
    cin.get();
    return 0;
}
}
```

PERFUNDIME

Në këtë raport u analizuan dhe u implementuan 14 metoda të ndryshme numerike dhe algoritmike duke përdorur gjuhën C++ dhe librarinë OpenMP. Qëllimi kryesor ishte krahasimi i performancës midis ekzekutimit Serial (1 Thread) dhe atij Paralel (2, 4, 6, 8, 12 Threads), si dhe matja e treguesve të performancës: **Speedup** dhe **Efiçencë**.

Nga eksperimentet e kryera, metodat u ndanë në 3 kategori kryesore bazuar në sjelljen e tyre:

1. Metodat me Përshpejtim Linear/Ideal

Këto janë metoda ku llogaritjet janë të pavarura nga njëra-tjetra dhe kërkojnë pak ose aspak komunikim midis procesorëve. Këtu pamë Speedup-in më të lartë, afër ideales.

- **Metodat:** *Shuma e Vektorëve, Mesatarja e Numrave Random, Përafrimi i pi, Integralet (Trapezit & Simpson).*
- **Rezultati:** Rritja e numrit të threads solli rritje pothuajse lineare të shpejtësisë. Këto janë rastet ideale për përdorimin e OpenMP.

2. Metodat Iterative dhe Algjebrike (Përshpejtim i mirë, por me kosto)

Këto metoda kërkojnë sinkronizim pas çdo iteracioni ose hapi, gjë që shton pak "overhead".

- **Metodat:** *Shumëzimi i Matricave, Metoda e Jakobit, Metoda e Fuqisë , Durand-Kerner (Polinomet).*
- **Rezultati:** Paralelizmi funksionoi mirë, por përshpejtimi nuk ishte perfekt (p.sh. me 12 threads morëm ~8-10x speedup). Kufizimi kryesor ishte aksesi në memorie (Memory Bandwidth) dhe sinkronizimi i shpeshtë.

3. Metodat Algoritmike Komplekse (Raste Speciale)

- Rasti i "Super-Linear Speedup" (Metoda e Përgjysmimit):
  Te kërkimi i rrënjëve (Seritë Fourier), paralelizmi ndryshoi logjikën e kërkimit . Kjo solli rezultate befasuese (deri në 55x Speedup), duke treguar se paralelizmi mund të përmirësojë cilësinë e algoritmit, jo vetëm shpejtësinë bruto.
- Rasti i "Vështirë për t'u Paralelizuar" (Bubble Sort):
  Te renditja, nevoja për sinkronizim pas çdo krahasimi bëri që Speedup-i të ishte minimal (max 1.5x) dhe madje negativ për vargje të vogla. Kjo demonstroi se jo çdo problem zgjidhet duke shtuar më shumë procesorë.

REFERENCAT

Për realizimin e këtij raporti dhe implementimin e algoritmeve janë shfrytëzuar burimet e mëposhtme:

[1] Mathematics Theory of Deep Learning.pdf
https://chatdata.ru/files/2407_18384.pdf

[2] Prof.Asc.Dr.Eglantina Kalluci. (2025). Leksione të lëndës "Programim Paralel dhe me Objekte". Fakulteti Shkencave te Natyres.

[3]https://gemini.google.com/u/2/app?pageId=none

[4]OpenMP Architecture Review Board. (2021). *OpenMP Application Program Interface, Version 5.1*. E aksesueshme në: https://www.openmp.org/specifications/

[5] Microsoft Learn. *OpenMP in Visual C++*. E aksesueshme në: https://learn.microsoft.com/cpp/parallel/openmp/.