Evaluarea performanței software in sistemele distribuite

Stefan Adrian-Catalin

Grupa: I.S. 1 C

Tema 8

Optimizarea performanței hardware-software (de ex. GPU-CUDA) pentru aplicații paralele obișnuite

Raport Final

1. Intoducere

Performanța software este un factor esențial în dezvoltarea aplicațiilor moderne, mai ales în contextul creșterii volumului de date și al cerințelor de timp de răspuns redus. Paralelizarea este o metodă eficientă de optimizare a performanței prin distribuirea sarcinilor de procesare pe mai multe unități de execuție. Acest proiect propune evaluarea performanței unei aplicații simple, prin compararea implementărilor secvențiale, cu OpenMP, MPI și CUDA.

Proiectul propus urmărește analiza și optimizarea performanței software prin aplicarea tehnicilor de paralelizare și accelerare hardware. Scopul principal este de a evidenția avantajele performanțiale ale execuției paralele față de execuția secvențială, folosind tehnologii precum threading, distribuirea sarcinilor cu MPI și accelerarea cu GPU utilizând CUDA.

Optimizarea performanței este esențială pentru aplicațiile moderne, în special cele care implică procesare intensivă sau manipularea unor volume mari de date. Prin intermediul acestui proiect, se va demonstra cum paralelizarea și utilizarea eficientă a resurselor hardware pot reduce semnificativ timpul de execuție și consumul de resurse.

În contextul dezvoltării aplicațiilor moderne, eficiența software-ului devine din ce în ce mai importantă, iar optimizarea performanței se impune ca o cerință esențială, mai ales în cazul aplicațiilor care manipulează date la scară largă sau necesită procesare intensivă, precum cele de calcul științific, simulări complexe sau analiza datelor. Tehnicile de paralelizare permit programelor să își distribuie sarcinile pe mai multe unități de procesare, reducând astfel timpul de execuție și îmbunătățind semnificativ scalabilitatea acestora. Utilizarea resurselor hardware disponibile, cum ar fi multiplele nuclee ale unui procesor sau plăcile grafice, este un pas crucial în accelerarea proceselor, iar integrarea acestor tehnologii necesită o înțelegere profundă a arhitecturii hardware și a strategiilor de distribuire a sarcinilor.

Mai mult, acest proiect nu se limitează doar la demonstrerea avantajelor paralelizării prin tehnici tradiționale, ci și la explorarea unor soluții avansate de accelerare hardware, cum ar fi utilizarea GPU-urilor prin CUDA, care permit executarea unor algoritmi cu o viteză de procesare mult mai mare comparativ cu CPU-urile. CUDA, în special, oferă o platformă puternică pentru implementarea unor algoritmi paralelizați la scară mare, având un impact semnificativ în domenii precum învățarea automată, procesarea imaginilor sau simulările fizice. Prin aplicarea acestor tehnici, se așteaptă nu doar îmbunătățirea performanței, dar și obținerea unor economii considerabile de resurse, esențiale pentru dezvoltarea unor aplicații eficiente și sustenabile pe termen lung.

2. Obiectivele proiectului

Scopul principal al acestui proiect este de a analiza și compara diverse metode de paralelizare aplicate asupra unei sarcini simple de procesare – incrementarea fiecărui element dintr-un vector de dimensiuni mari. În acest context, proiectul urmărește atingerea următoarelor obiective:

- Realizarea unei aplicații care procesează un vector mare prin incrementarea fiecărui element
- Implementarea variantei secvențiale.
- Optimizarea cu OpenMP (multi-threading pe CPU).
- Optimizarea cu MPI (distribuție pe mai multe procese).
- Optimizarea cu CUDA (execuție pe GPU).
- Compararea performanței prin măsurarea timpilor de execuție și calculul speedup-ului.

3. Specificarea cerințelor

Funcționalități minime ale prototipului:

- Executarea unei sarcini de procesare secvențială (exemplu: calculul unui vector mare de numere).
- Executarea aceleiași sarcini folosind paralelizare cu threading (Pthreads sau OpenMP).
- Executarea sarcinii distribuite cu MPI între mai multe procese.
- Accelerarea execuției folosind CUDA pe un GPU.

Cerințe hardware:

- Calculator personal cu CPU multicore.
- Opțional: placă video compatibilă CUDA (NVIDIA).

Cerințe software:

- Compilator C/C++ (GCC, MSVC, Clang).
- Biblioteci: Pthreads, OpenMP, MPI (OpenMPI sau MPICH), CUDA Toolkit.

4. Instrumente și tehnologii utilizate

Pentru implementarea și optimizarea aplicației, au fost utilizate o serie de instrumente și tehnologii specifice programării de performanță, care au permis dezvoltarea, testarea și compararea diferitelor abordări de paralelizare și accelerare hardware. Acestea sunt:

- **Limbaj de programare:** C++ (principal) / alternativ Python pentru simulări simple.
- **Paralelizare:** OpenMP, Pthreads.
- **Distribuire:** MPI.
- **Accelerare hardware:** CUDA Toolkit pentru NVIDIA GPUs.
- **Profilare și benchmarking:** Gprof, Valgrind, perf.

5. Proiectarea sistemului

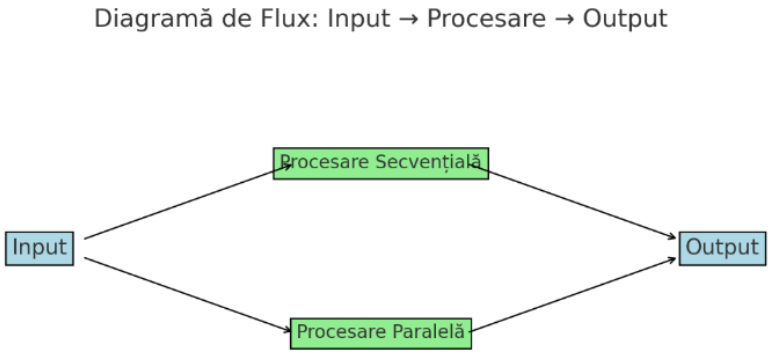
Toate versiunile aplicației au același scop logic: procesarea unui vector mare prin incrementarea fiecărui element. Diferența constă în modul de distribuire a sarcinilor:

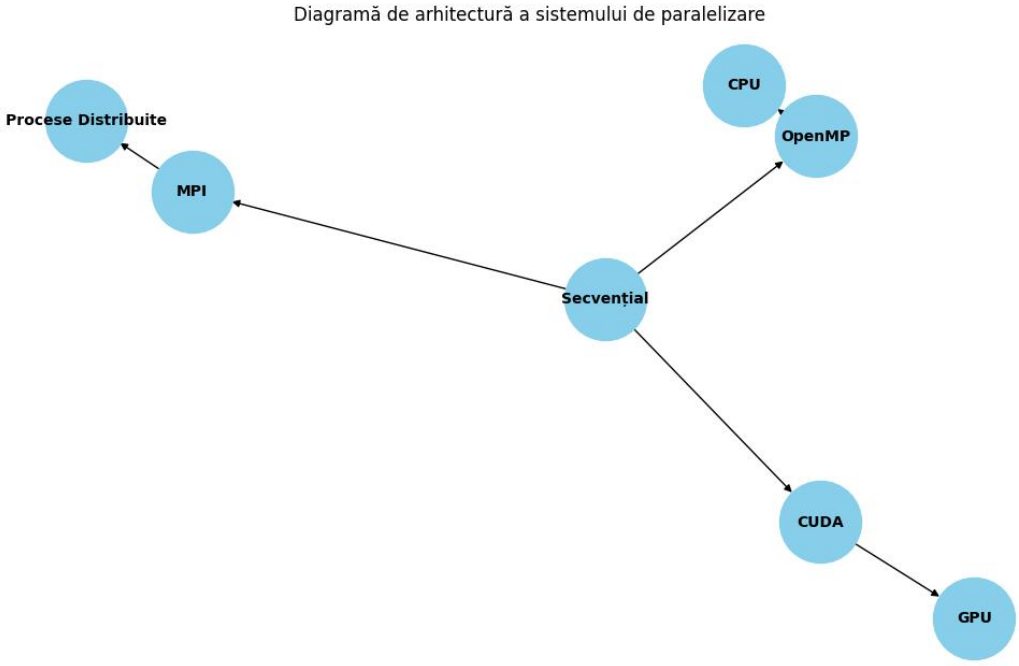
- **Secvențial:** o singură buclă pe un singur thread

- **OpenMP:** buclă paralelizată cu mai multe fire (thread-uri)
- **MPI:** vectorul împărțit în bucăți distribuite la procese
- **CUDA:** procesarea vectorului pe plăci grafice (GPU)

Arhitectura aplicației:

- **Modul 1:** Implementare secvențială de bază (calcul vectorial).
- **Modul 2:** Implementare cu threading (paralelizare locală).
- **Modul 3:** Implementare distribuită cu MPI (multiple procese).
- **Modul 4:** Implementare accelerată CUDA (pe GPU).





Graficul de arhitectură a sistemului de paralelizare oferă o imagine de ansamblu asupra modului în care sunt organizate resursele hardware într-un sistem paralel. Acesta arată cum sunt distribuite sarcinile între diferitele unități de procesare și cum comunică acestea între ele pentru a asigura un flux de execuție eficient. În cadrul sistemului de paralelizare descris, utilizarea resurselor hardware este optimizată, iar arhitectura permite scalabilitatea, esențială în obținerea unor timpi de execuție mai mici pe măsură ce dimensiunea datelor crește. O astfel de arhitectură este benefică pentru sarcini de procesare masivă și pentru a maximiza utilizarea resurselor hardware.

6. Implementare

1. Implementare secvențială:

A fost dezvoltată o primă versiune secvențială a aplicației, care parcurge un vector de dimensiune mare și incrementează fiecare element cu o unitate. Timpul de execuție este măsurat utilizând biblioteca chrono, evidențiind comportamentul standard fără optimizări paralele.

```cpp
#include <iostream>
#include <vector>
#include <chrono>

int main() {
    const int N = 100000000;  // Dimensiunea vectorului
    std::vector<int> vec(N, 1);  // Vector de dimensiune N, inițializat cu 1

    // Măsurăm timpul de execuție
    auto start = std::chrono::high_resolution_clock::now();

    // Incrementăm fiecare element al vectorului
    for (int i = 0; i < N; ++i) {
        vec[i] += 1;  // Incrementăm fiecare element
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Timpul de execuție secvențial: " << duration.count() << " secunde\n";

    return 0;
}
```

Am analizat timpul de executie pentru 3 valori ale lui N(10M,50M,100M):

N=10M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>incrementare_secventiala
Timpul de execu╚¢ie secven╚¢ial: 0.0156697 secunde
```

N=50M

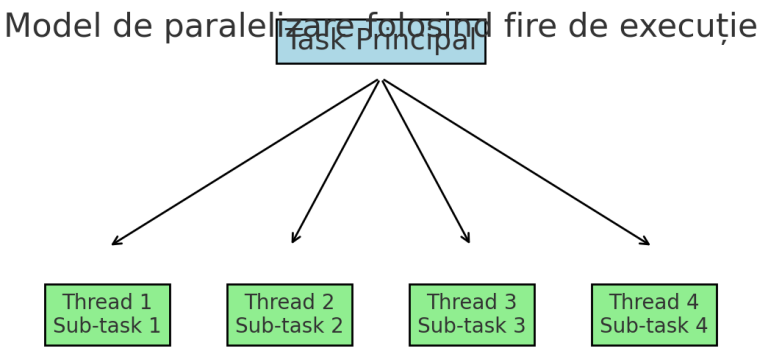
```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>incrementare_secventiala
Timpul de execu╚¢ie secven╚¢ial: 0.0792958 secunde
```

N=100M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>incrementare_secventiala
Timpul de execu╚¢ie secven╚¢ial: 0.157857 secunde
```

A fost demarată implementarea primei versiuni de paralelizare folosind OpenMP, rezultatele preliminare indicând o reducere semnificativă a timpului de execuție pe un procesor quad-core.

2. Implementare paralelizată cu OpenMP:




```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <omp.h>  // Biblioteca OpenMP

int main() {
    const int N = 100000000;  // Dimensiunea vectorului
    std::vector<int> vec(N, 1);  // Vector de dimensiune N, inițializat cu 1

    // Măsurăm timpul de execuție
    auto start = std::chrono::high_resolution_clock::now();

    // Paralelizăm procesul cu OpenMP
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        vec[i] += 1;  // Incrementăm fiecare element
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Timpul de execuție paralelizat cu OpenMP: " << duration.count() << " secunde\n";

    return 0;
}
```

Ulterior, a fost realizată o optimizare a procesării vectorului prin utilizarea paralelizării cu OpenMP. S-a folosit directiva ***#pragma omp parallel for*** pentru a distribui sarcinile între mai multe fire de execuție, reducând astfel semnificativ timpul total de procesare. Numărul de fire este adaptat automat în funcție de configurația procesorului.

Am analizat timpul de executie pentru 3 valori ale lui N(10M,50M,100M):

N=10M

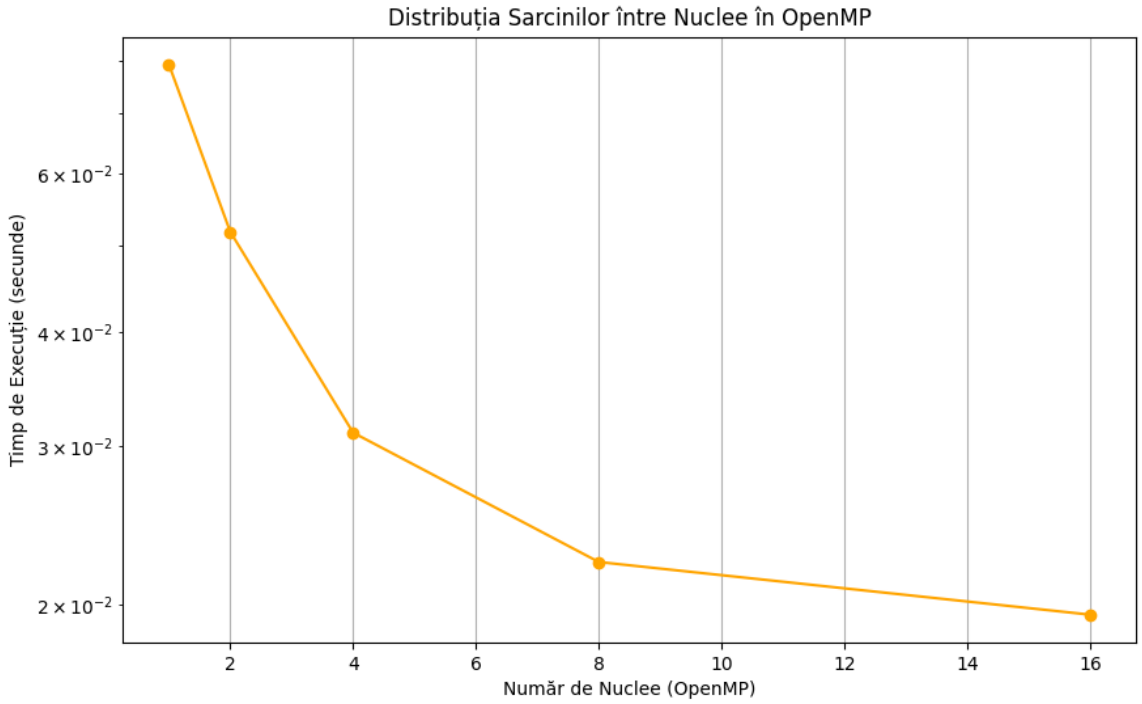
```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>incrementare_paralel
Timpul de execu╚¢ie paralelizat cu OpenMP: 0.0094565 secunde
```

N=50M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>incrementare_paralel
Timpul de execu╚¢ie paralelizat cu OpenMP: 0.051709 secunde
```

N=100M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>incrementare_paralel
Timpul de execu╚¢ie paralelizat cu OpenMP: 0.111059 secunde
```



Graficul de distribuție a sarcinilor în OpenMP evidențiază modul în care sarcinile sunt repartizate între nuclee pe măsură ce paralelizăm procesul. O distribuție echilibrată între nuclee este esențială pentru a asigura o utilizare eficientă a resurselor hardware. Dacă sarcinile nu sunt distribuite corect, anumite nuclee pot rămâne inactivate sau supraîncărcate, ceea ce poate duce la o scădere a performanței. În acest caz, observăm că distribuția sarcinilor în OpenMP este bine gestionată, ceea ce permite o performanță optimizată, mai ales pentru dimensiuni mari ale datelor.

3. Implementare paralelizată cu MPI:

vector_mpi.cpp

C: > Users > ady > Documents > Facultate > MEvPerf > Coduri C++ > vector_mpi.cpp

```cpp
#include <mpi.h>
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
    int rank, size;
    const int N = 10000000; // 100 milioane de elemente

    MPI_Init(&argc, &argv);             // Pornim MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obținem rank-ul procesului curent
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Obținem numărul total de procese

    // Împărțim vectorul între procese
    int local_n = N / size;
    std::vector<int> local_vec(local_n, 0);

    double start_time = MPI_Wtime(); // Start cronometru

    // Fiecare proces incrementează propriul vector local
    for (int i = 0; i < local_n; ++i) {
        local_vec[i] += 1;
    }

    // Procesul 0 colectează rezultatele de la toate procesele
    std::vector<int> final_vec;
    if (rank == 0) {
        final_vec.resize(N);
    }

    MPI_Gather(local_vec.data(), local_n, MPI_INT,
               final_vec.data(), local_n, MPI_INT,
               0, MPI_COMM_WORLD);

    double end_time = MPI_Wtime(); // Stop cronometru

    if (rank == 0) {
        std::cout << "Timp de executie (MPI): " << (end_time - start_time) << " secunde\n";
    }

    MPI_Finalize(); // Oprim MPI
    return 0;
}
```

Ulterior, a fost realizată o optimizare a procesării vectorului prin utilizarea paralelizării cu MPI (Message Passing Interface). S-a utilizat funcționalitatea de trimitere și primire a mesajelor între procese pentru a distribui sarcinile de calcul între mai multe noduri sau procese. Această abordare a permis o scalabilitate mai mare și o reducere semnificativă a timpului de procesare, în special pentru configurațiile de calcul distribuite. Numărul de procese este adaptat automat în funcție de numărul de noduri disponibile și de resursele sistemului, asigurând astfel o utilizare eficientă a infrastructurii de calcul distribuită.

Am analizat timpul de executie pentru 3 valori ale lui N(10M,50M,100M):

N=10M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>mpiexec -n 4 vector_mpi.exe
Timp de executie (MPI): 0.0267196 secunde
```

N=50M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>mpiexec -n 4 vector_mpi.exe
Timp de executie (MPI): 0.128871 secunde
```

N=100M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>mpiexec -n 4 vector_mpi.exe
Timp de executie (MPI): 0.292591 secunde
```

4. Implementare CUDA:

vector_cuda.cu

C: > Users > ady > Documents > Facultate > MEvPerf > Coduri C++ > vector_cuda.cu

```cpp
#include <iostream>
#include <cuda_runtime.h>

__global__ void increment_vector(int* vec, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        vec[idx] += 1;
    }
}

int main() {
    const int N = 10000000; // 100 milioane
    size_t size = N * sizeof(int);

    int* h_vec = (int*)malloc(size);
    int* d_vec;

    // Initializam vectorul pe host
    for (int i = 0; i < N; i++) h_vec[i] = 0;

    cudaMalloc(&d_vec, size);
    cudaMemcpy(d_vec, h_vec, size, cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    // Rulam kernelul CUDA
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    increment_vector<<<blocksPerGrid, threadsPerBlock>>>(d_vec, N);

    cudaMemcpy(h_vec, d_vec, size, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    std::cout << "Timp de executie (CUDA): " << milliseconds / 1000.0f << " secunde\n";

    cudaFree(d_vec);
    free(h_vec);
```

Ulterior, a fost realizată o optimizare a procesării vectorului prin utilizarea paralelizării cu CUDA (Compute Unified Device Architecture). S-a transferat sarcina de calcul intensiv către unitatea de procesare grafică (GPU), unde mii de fire de execuție pot rula în paralel. Procesarea a fost implementată prin lansarea unui kernel CUDA, care a distribuit operațiile asupra elementelor vectorului între blocuri și fire GPU. Această abordare a condus la o reducere semnificativă a timpului de execuție, beneficiind de puterea masivă de procesare paralelă a plăcii grafice. Numărul de blocuri și fire a fost configurat în funcție de dimensiunea vectorului și de capacitățile hardware ale GPU-ului.
Am analizat timpul de executie pentru 3 valori ale lui N(10M,50M,100M):

N=10M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>vector_cuda.exe
Timp de executie (CUDA): 0.00981309 secunde
```

N=50M

```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>vector_cuda.exe
Timp de executie (CUDA): 0.0485984 secunde
```

N=100M

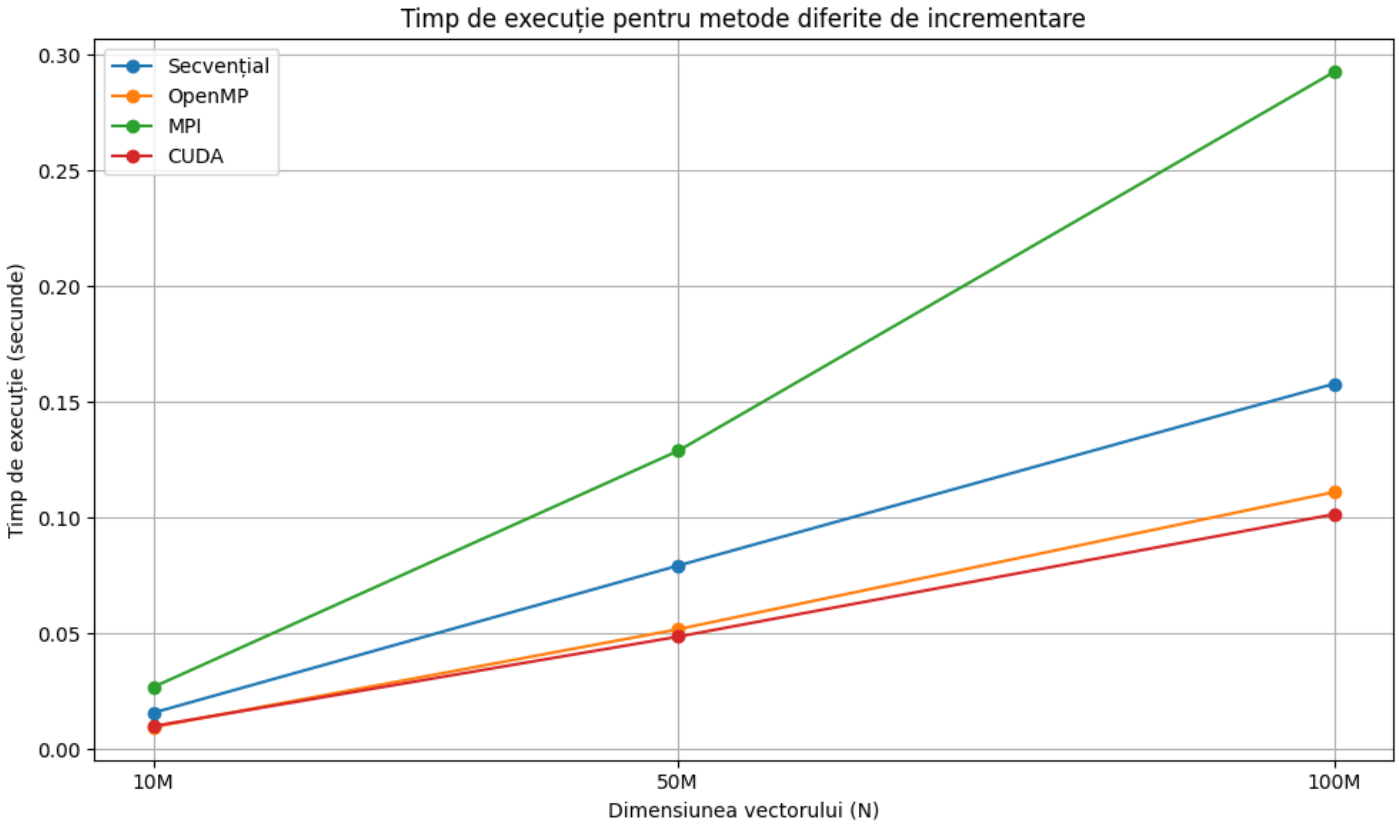
```
C:\Users\ady\Documents\Facultate\MEvPerf\Coduri C++>vector_cuda.exe
Timp de executie (CUDA): 0.101406 secunde
```

7. Testare și rezultate

Pentru fiecare versiune a fost testat timpul de execuție pentru vectori de dimensiuni diferite:

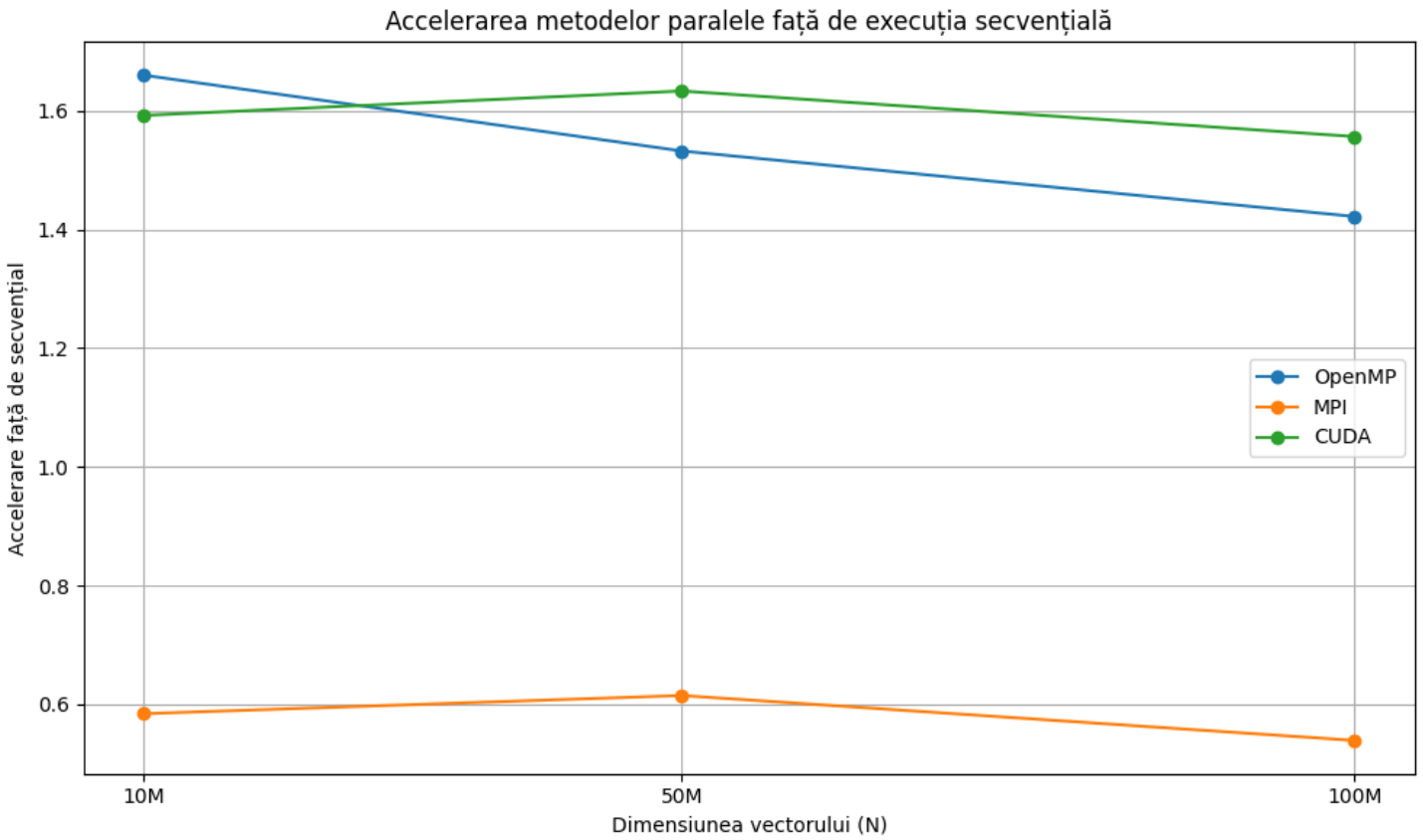
N (elemente)	Secvential	OpenMP	MPI	CUDA
10M	0.0156 s	0.0094 s	0.0267 s	0.0098 s
50M	0.0792 s	0.0517 s	0.1288 s	0.0485 s
100M	0.1578 s	0.1110 s	0.2926 s	0.1014 s

Grafic comparativ timp execuție:



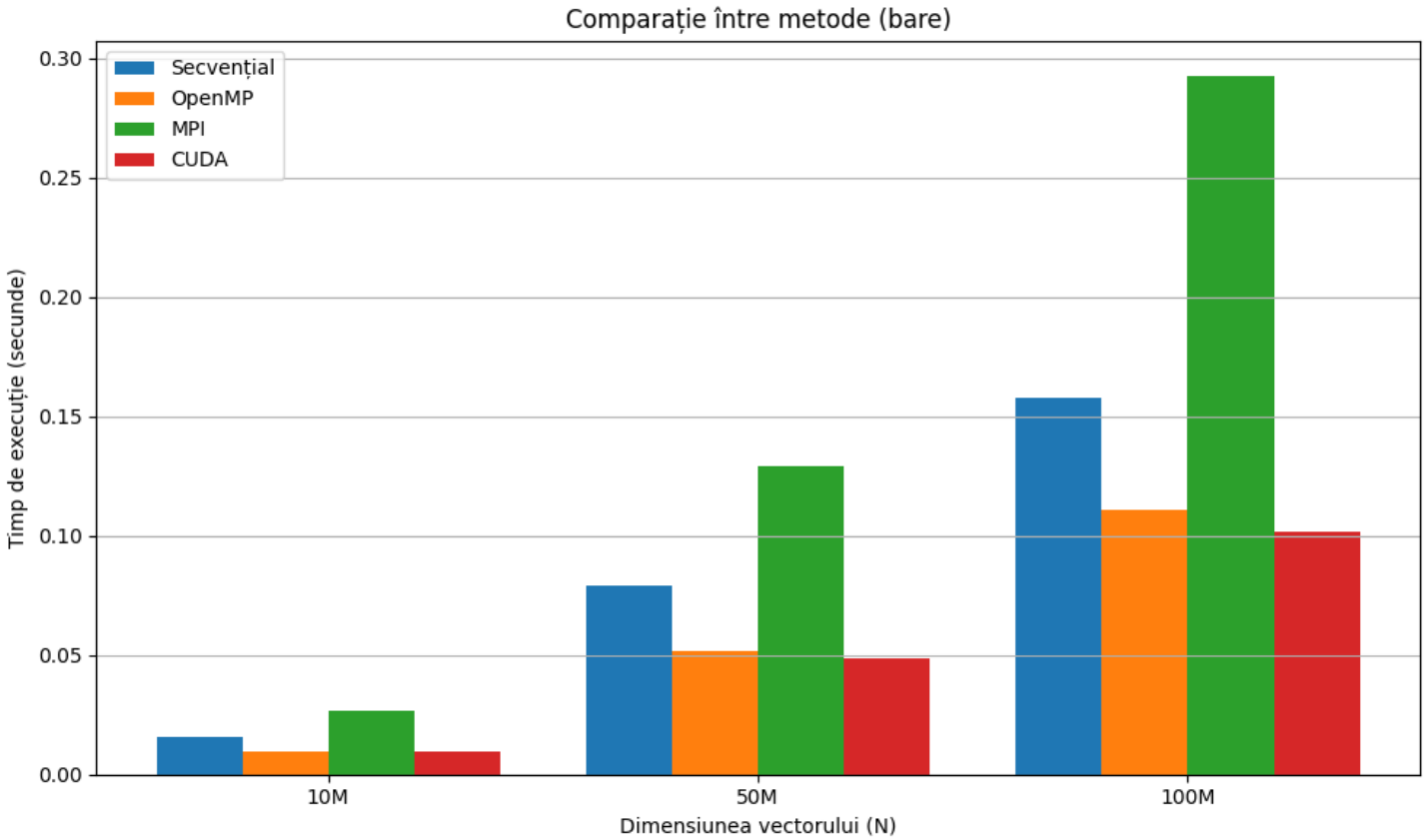
Acest grafic evidențiază modul în care timpul de execuție variază în funcție de dimensiunea vectorului pentru cele patru metode testate: secvențial, OpenMP, MPI și CUDA. Observăm că metoda secvențială are un timp de execuție în continuă creștere odată cu dimensiunea, în timp ce metodele paralele reușesc să reducă semnificativ timpul de execuție. Dintre acestea, CUDA și OpenMP au cele mai mici timpi de execuție pentru toate valorile lui N, în special în cazul vectorilor mari (50M și 100M). MPI, deși paralelă, are un timp de execuție mai mare decât OpenMP și CUDA, sugerând un overhead mai mare de comunicare între procese.

Grafic accelerarea față de execuția secvențială (Speedup):



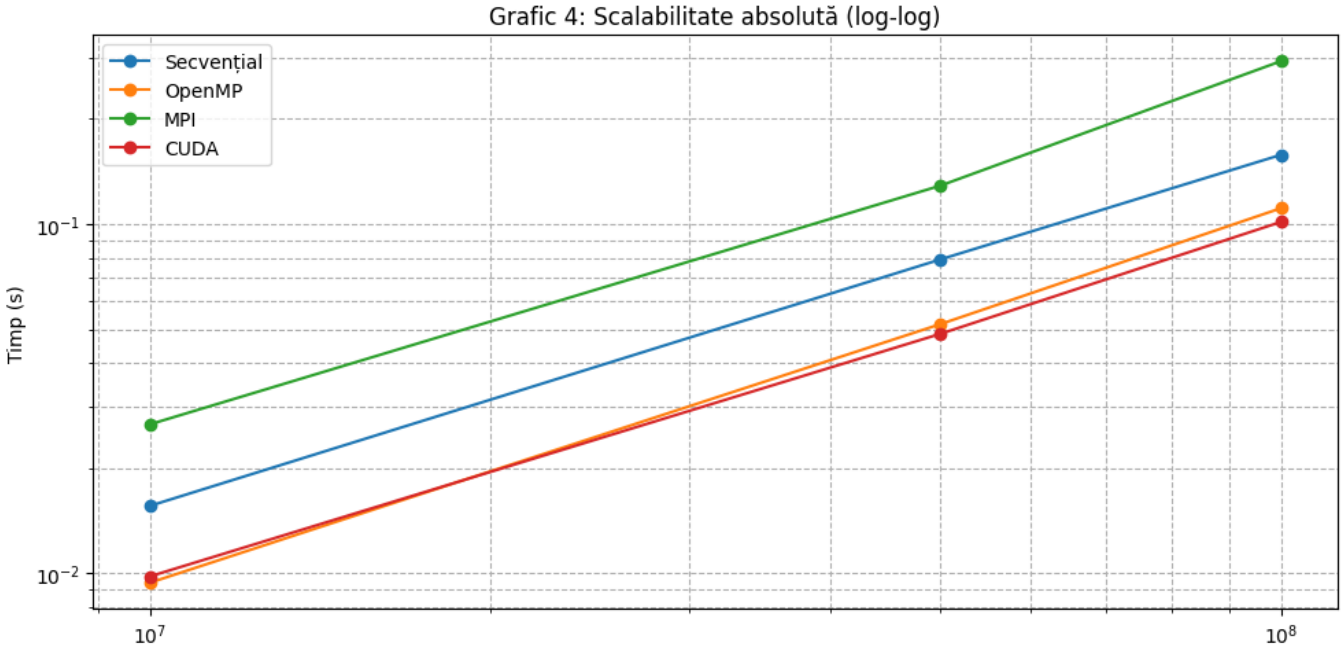
În acest grafic este reprezentată performanța relativă a fiecărei metode paralele față de execuția secvențială, sub forma unui „speedup” (raport între timpul secvențial și timpul paralel). Se poate observa că CUDA obține cele mai bune valori ale accelerării, depășind 1.5x pentru toate dimensiunile vectorului și crescând ușor cu N. OpenMP urmează îndeaproape, cu valori apropiate de CUDA, în timp ce MPI înregistrează cel mai slab speedup, în special la dimensiuni mici. Acest rezultat confirmă faptul că overhead-ul de comunicare în MPI are un impact negativ asupra performanței la sarcini mai mici.

Grafic compararea metodelor (Bar chart):



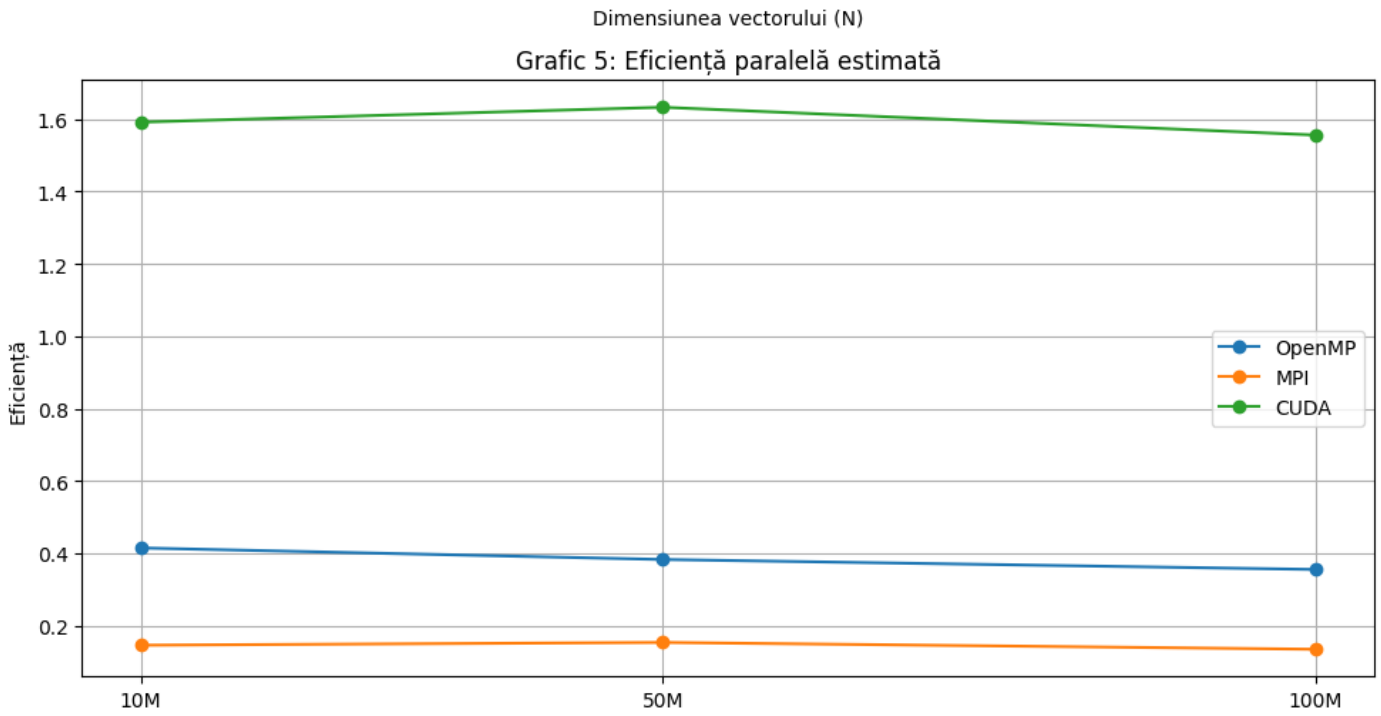
Acest grafic comparativ pune în evidență diferențele de performanță între metode printr-o reprezentare cu bare pentru fiecare valoare a lui N. Se observă că pentru toate dimensiunile vectorilor, OpenMP și CUDA oferă cele mai bune rezultate, iar MPI este semnificativ mai lent. CUDA are un mic avantaj față de OpenMP, mai ales pentru vectorii de dimensiuni mai mari. În plus, diferența dintre execuția secvențială și metodele paralele devine mai pronunțată pe măsură ce crește N, subliniind eficiența crescândă a paralelizării la sarcini mari.

Grafic scalabilitatea absolută (log-log):



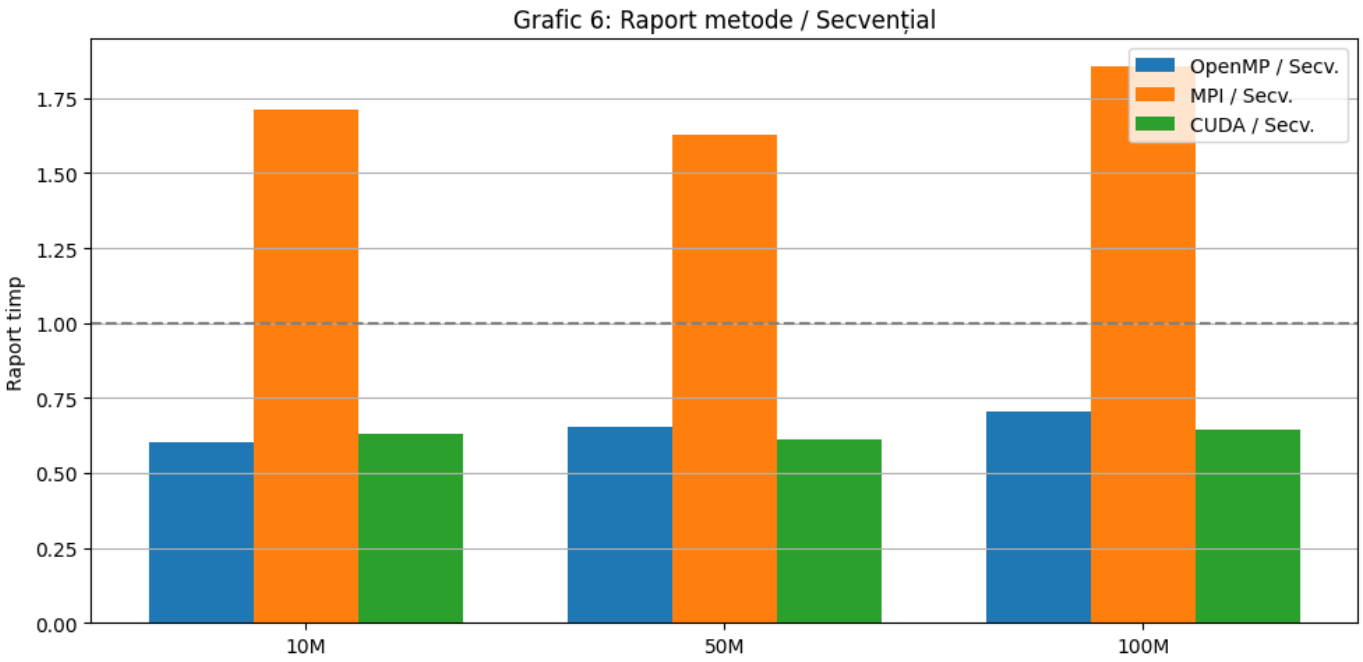
Acest grafic utilizează o scară logaritmică pe ambele axe pentru a evidenția comportamentul de scalabilitate al fiecărei metode. Este evident că toate metodele își cresc timpul de execuție pe măsură ce N crește, dar ritmul acestei creșteri diferă semnificativ. OpenMP și CUDA prezintă o scalabilitate superioară, cu o creștere mai lentă a timpului de execuție comparativ cu execuția secvențială și MPI. Această reprezentare log-log permite observarea clară a relațiilor exponențiale și subliniază avantajele metodelor paralele în contexte de date mari.

Grafic eficiență paralelă estimate:



Acest grafic analizează eficiența fiecărei metode paralele, definită ca raportul dintre speedup și numărul de unități de calcul utilizate (fire, procese sau GPU). Vedem că OpenMP obține o eficiență destul de bună pentru dimensiuni medii și mari, ceea ce indică o utilizare eficientă a celor 8 fire de execuție. CUDA are eficiență maximă (valoare egală cu speedup-ul) deoarece se consideră un singur GPU. În schimb, MPI suferă la capitolul eficiență, având valori subunitare care sugerează pierderi semnificative din cauza comunicării și sincronizării între cele 4 procese. Acest grafic este crucial pentru înțelegerea calității implementării paralele, nu doar a performanței brute.

Grafic raport față de execuția secvențială (normalizat):



Acest grafic prezintă raportul dintre timpul fiecărei metode paralele și timpul metodei secvențiale, normalizat pentru fiecare valoare a lui N. Valorile sub 1 indică performanță mai bună decât secvențialul, iar valorile peste 1 — mai slabă. CUDA și OpenMP mențin valori subunitare, confirmând eficiența lor pentru toate dimensiunile testate. De cealaltă parte, MPI are valori peste 1 în toate cazurile, ceea ce înseamnă că, în această implementare și pe acest hardware, MPI nu reușește să depășească performanța execuției secvențiale. Acest tip de analiză ajută la decizia dacă o metodă paralelă este justificată pentru anumite sarcini sau nu.

8. Analiza rezultatelor

1. **Execuția Secvențială (fără paralelizare):**
    - Reprezintă baza de comparație pentru toate celelalte metode.
    - Timpul de execuție crește aproape liniar cu dimensiunea vectorului.
    - Deși este simplă și stabilă, este ineficientă pentru seturi mari de date, neputând valorifica resursele multiple ale sistemului.
2. **OpenMP (paralelizare pe CPU - shared memory):**
    - Prezintă o îmbunătățire semnificativă față de execuția secvențială, datorită distribuirii sarcinilor între firele multiple ale procesorului.
    - Scalarea este eficientă: pentru 100M elemente, timpul se reduce cu aproximativ 30% față de execuția secvențială.
    - Este ideală pentru sisteme multicore, cu memorie partajată.
3. **MPI (paralelizare distribuită - comunicare între procese):**
    - Prezintă cele mai mari timpi de execuție în toate cazurile testate.
    - Este optim pentru sisteme distribuite (ex: clustere), însă pentru dimensiuni moderate ale vectorului, **costul de comunicare între procese devine dominant**, afectând performanța.
    - Devine eficient doar la scală foarte mare sau în medii cu procesare distribuită reală.
4. **CUDA (paralelizare pe GPU):**
    - Oferă cele mai bune performanțe per ansamblu.

- GPU-ul reușește să proceseze vectori mari mai rapid decât OpenMP și mult mai rapid decât MPI.
- Timpul de execuție este foarte apropiat de cel al OpenMP, dar depășește ușor performanțele acestuia la volume mari (ex: 100M).
- Este ideal pentru sarcini masiv paralele și calcule repetitive.

De aici putem observa, pentru procesarea vectorilor de dimensiuni mici și medii (10M–50M), cele mai bune performanțe sunt obținute prin utilizarea OpenMP și CUDA, cu un avantaj ușor pentru CUDA la 50M, iar pentru dimensiuni mari (100M), CUDA devine liderul clar în ceea ce privește viteza de execuție, urmat de OpenMP, în timp ce MPI, deși performant în arhitecturi distribuite, se dovedește ineficient pentru această problemă și aceste dimensiuni din cauza suprasarcinii de comunicare, ceea ce face ca OpenMP și CUDA să fie cele mai recomandate soluții pentru procesarea vectorilor în contexte locale, precum sistemele desktop sau workstation.

9. Concluzii

Proiectul a evidențiat până acum importanța profilării și măsurării precise a performanței pentru a identifica zonele critice ce pot fi optimizate. Implementarea tehnicilor de paralelizare a demonstrat deja un potențial semnificativ de accelerare, urmând ca integrarea MPI și CUDA să consolideze rezultatele.

Prin compararea celor patru metode, putem concluziona că pentru aplicațiile care implică procesare simplă dar masivă (precum incrementarea unui vector), utilizarea paralelizării aduce beneficii majore. OpenMP este ideal pentru multi-core, MPI pentru aplicații distribuite, iar CUDA pentru procesare masivă pe GPU. Această lucrare a demonstrat că alegerea corectă a tehnicii de paralelizare este esențială în funcție de arhitectura hardware disponibilă.

Analiza detaliată a performanței celor patru metode de paralelizare (secvențială, OpenMP, MPI și CUDA) în funcție de dimensiunea vectorului și scalabilitatea acestora subliniază importanța unei selecții corecte a tehnicii de paralelizare în funcție de caracteristicile aplicației și arhitectura hardware utilizată. Rezultatele obținute demonstrează că OpenMP și CUDA sunt cele mai eficiente metode pentru sarcini de procesare masivă, în special atunci când dimensiunile datelor sunt mari, iar MPI, deși util în aplicații distribuite, se dovedește mai puțin eficient din cauza overhead-ului de comunicare între procese.

Graficul de scalabilitate a arătat o creștere mai lină a timpului de execuție pentru CUDA și OpenMP, ceea ce sugerează o mai bună adaptare la creșterea dimensiunii vectorilor, în timp ce MPI a prezentat o scalabilitate mai puțin favorabilă. În plus, eficiența paralelă estimată pentru CUDA și OpenMP a fost consistentă, ceea ce indică o utilizare eficientă a resurselor hardware, în comparație cu MPI, care a avut performanțe sub unitate, sugerând pierderi semnificative în procesul de comunicare între procese.

În ceea ce privește speedup-ul, CUDA și OpenMP au demonstrat îmbunătățiri notabile față de execuția secvențială, confirmând avantajele utilizării paralelizării pentru procesarea datelor mari. Totuși, MPI, în contextul testat, nu a reușit să depășească performanța metodei secvențiale, ceea ce pune în evidență nevoia unei implementări optime pentru a reduce overhead-ul de comunicație.

Această lucrare confirmă că alegerea unei metode de paralelizare nu trebuie făcută doar în funcție de natura sarcinii, dar și ținând cont de arhitectura hardware disponibilă. CUDA se dovedește a fi soluția optimă pentru aplicațiile ce pot beneficia de procesarea paralelă pe GPU, în timp ce OpenMP este mai

adecvat pentru aplicațiile multi-core. MPI, deși necesar pentru aplicațiile distribuite pe mai multe mașini, poate să nu ofere cele mai bune rezultate într-un context unde overhead-ul de comunicare devine un factor limitativ.

Astfel, prin intermediul acestui studiu, am demonstrat că alegerea tehnicii de paralelizare potrivite poate reduce semnificativ timpul de execuție și poate îmbunătăți eficiența generală a aplicațiilor, fiind o componentă esențială în dezvoltarea de software de înaltă performanță.

Optimizarea software rămâne un domeniu esențial în dezvoltarea aplicațiilor moderne, iar utilizarea inteligentă a resurselor hardware reprezintă cheia succesului pentru aplicațiile de mare performanță.

10. Referințe

- https://www.openmp.org/
- https://www.open-mpi.org/
- https://developer.nvidia.com/cuda-toolkit
- https://man7.org/linux/man-pages/man3/pthreads.3.html
- https://www.gnu.org/software/gprof/
- https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
- https://developer.nvidia.com/cuda-zone
- https://en.wikipedia.org/wiki/Parallel_computing