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 intermediar

1. Intoducere

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

- Implementarea unei aplicații simple care poate fi rulată în mod secvențial și paralelizat.
- Compararea timpilor de execuție între variantele secvențiale și paralele.
- Utilizarea firelor de execuție (threading) pentru paralelizare locală.
- Utilizarea MPI pentru distribuirea sarcinilor pe mai multe noduri.
- Folosirea CUDA pentru accelerarea execuției cu ajutorul GPU-ului.
- Analiza performanței și realizarea de grafice comparative.

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.

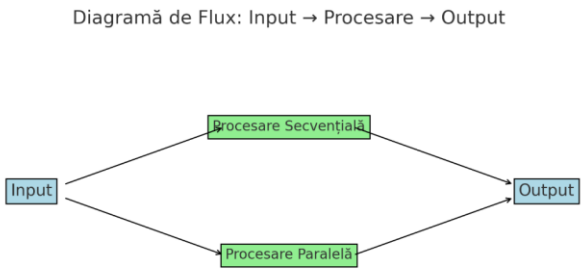
4. Instrumente și tehnologii utilizate

- **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

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).



6. Implementare minimală

Până în acest moment, a fost realizată implementarea secvențială a algoritmului de procesare. Algoritmul presupune parcurgerea unui vector de dimensiune mare și aplicarea unei funcții simple asupra fiecărui element (ex: incrementarea valorii).

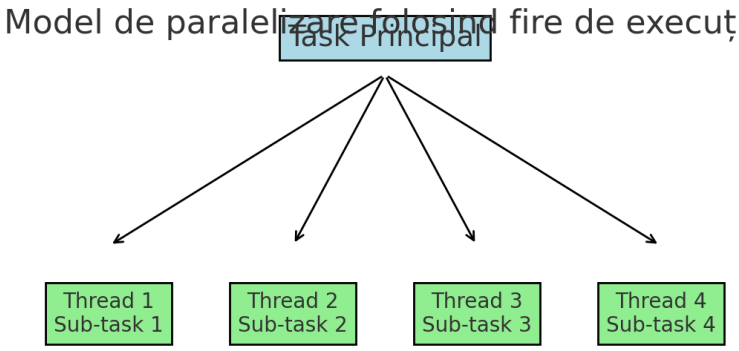
```cpp
#include <iostream>
#include <vector>
#include <chrono>

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;
}
```

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.

Exemplu de cod secvențial:

```cpp
for (int i = 0; i < n; i++) {
    v[i] = v[i] + 1;
}
```

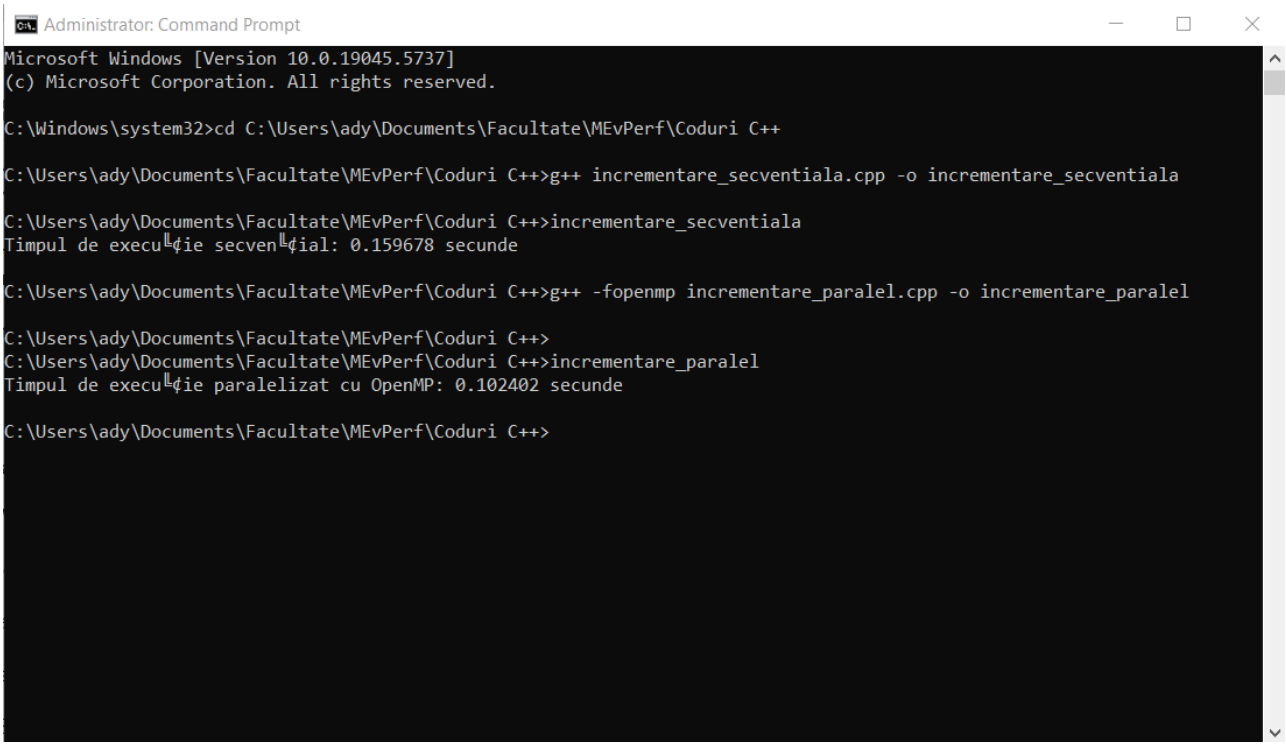
A fost demarată implementarea primei versiuni de paralelizare folosind OpenMP, rezultatele preliminare indicând o reducere semnificativă a timpului de execuție pe un procesor quad-core.




```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;
}
```

Implementare paralelizată cu OpenMP:
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.

Rezultate preliminare:
Pentru a evalua impactul optimizării prin paralelizare, au fost rulate ambele variante ale aplicației (secvențială și paralelizată cu OpenMP). Timpul de execuție a fost măsurat utilizând biblioteca chrono, iar rezultatele obținute sunt ilustrate în figura de mai jos.



Observație importantă:

- Timp secvențial: **0.159678 secunde**
- Timp paralelizat OpenMP: **0.102402 secunde**

Se observă că optimizarea prin paralelizare a redus timpul total de execuție cu aproximativ 36%. Această reducere demonstrează eficiența utilizării firelor de execuție multiple în procesarea datelor pe un procesor multicore.

7. Prototip de sistem

Prototip realizat:

- Cod secvențial complet funcțional.
- Cod paralelizat cu OpenMP funcțional pentru procesare pe mai multe fire.

Etape viitoare:

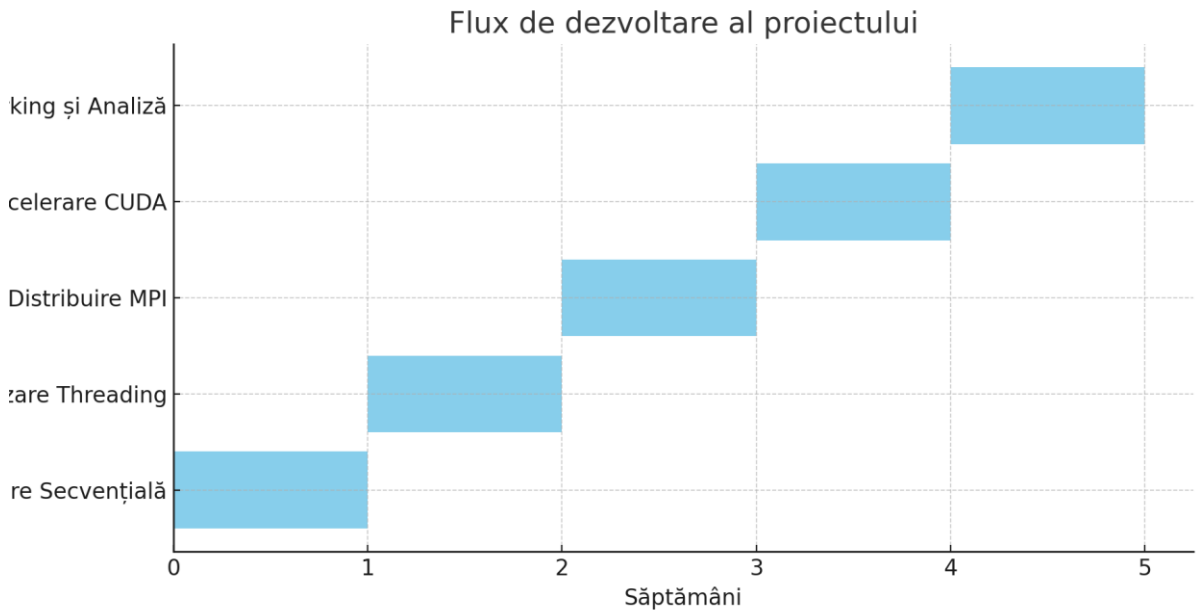
- Finalizarea distribuției sarcinilor cu MPI.
- Integrarea execuției accelerate cu CUDA.

Probleme întâlnite:

- Necesitatea optimizării încărcării firelor pentru a evita situațiile de "overhead".
- Necesitatea sincronizării eficiente între procesele MPI.

8. Plan de lucru pentru finalizarea proiectului

- T1: Finalizarea implementării versiunii MPI (Săptămâna 1-2).
- T2: Implementarea versiunii CUDA (Săptămâna 2-3).
- T3: Colectarea datelor de performanță (benchmarking) (Săptămâna 3-4).
- T4: Analiza rezultatelor și realizarea graficelor comparative (Săptămâna 4).
- T5: Scrierea raportului final și pregătirea prezentării (Săptămâna 5).



9. Concluzii parțiale

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.

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/