UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Măsurarea timpului de execuție a proceselor în C++, Python și Rust

Structura Sistemelor de Calcul

Autor: Vlășan Darius-Ioan
Grupa: 30321

Facultatea de Automatica
si Calculatoare

Decembrie 2023

Cuprins

1 Introducere

1.1 Context

Proiectul propus se concentrează pe măsurarea timpului necesar pentru executarea proceselor scrise în limbajele de programare C++, Python și Rust. Scopul principal al acestui proiect constă în evaluarea și comparația performanței acestor limbaje în diferite situații, având în vedere aspecte precum alocarea de memorie, accesul la memorie, crearea de thread-uri, schimbul de context între thread-uri și migrația acestora între nucleele de procesor.

Pentru comunitatea dezvoltatorilor și cercetătorilor din domeniul Calculatoarea, rezultatele pot fi folosite pentru luarea unor decizii informate privind alegerea limbajului de programare potrivit pentru proiectele lor, în funcție de cerințele de performanță; pentru optimizarea codului existent, precum și dezvoltarea de noi tehnologii care folosesc punctele forte specifice fiecărui limbaj comparat.

1.2 Specificații

Testarea este efectuată pe Windows, folosind mediile integrate de dezvoltare(IDE) dezvoltate de JetBrains: CLion pentru programarea în C++, pyCharm pentru Python, și RustRover pentru limbajul de programare Rust. Specificațiile hardware ale sistemului sunt următoarele:

- Procesor: AMD Ryzen 7 5800U, cu o frecvență de 1.90GHZ
- Placa video : Radeon Graphics (integrată)
- Memorie RAM: 16.0 GB

1.3 Obiective

Pentru o evaluare cât mai obiectivă și cuprinzătoare a performanței, urmărim următoarele obiective:

- Evaluarea alocării și accesului la memorie, atât pentru cea statică, cât și pentru cea dinamică.
- Crearea de thread-uri: măsurarea timpului necesar pentru a crea și distruge thread-uri în fiecare limbaj, pentru a evalua performanța funcțiilor de gestionare a acestora.
- Context switch între thread-uri: evidențierea eficienței planificării și schimbării de context în fiecare limbaj.
- Migrarea thread-urilor între nuclee de procesor: evaluarea gestionării migratei în fiecare limbaj.

2 Studio bibliografic

2.1 Ceas Monotonic

- Este un tip de ceas care nu se modifică niciodată în sensul decrementării.
- Crește constant în valoare și nu este afectat de schimbările orare sau de sincronizarea cu rețele externe.
- Este utilizat pentru măsurarea duratei de timp între evenimente sau pentru a determina timpul scurs între două puncte în timp.
- Utilizarea sa este utilă în aplicații care necesită măsurători precise ale timpului și care trebuie să funcționeze independent de corecțiile orare ale sistemului.

2.2 Ceasul de Perete (Wall Clock)

- Este un ceas obișnuit care afișează ora și data actuale, precum un ceas de perete tradițional.
- Se sincronizează cu ora locală a sistemului și poate fi afectat de schimbările orare.
- Este preferat pentru măsurarea timpului de execuție pe multicore.

2.3 Alocarea memoriei

2.3.1 Alocarea statică

- În alocarea statică, spațiul de memorie este rezervat în timpul etapei de compilare și rămâne neschimbat pe întreaga durată a execuției programului.
- Variabilele statice sunt alocate într-un segment special al memoriei denumit segmentul de date.
- Aceste variabile persistă pe toată durata execuției programului și au o durată de viață prelungită.
- Exemple de alocare statică includ variabilele globale și variabilele statice locale.

2.3.2 Alocarea dinamică

- Alocarea dinamică permite programului să solicite și să elibereze spațiu de memorie în timpul execuției.
- Aceasta se realizează folosind funcții precum `malloc`, `calloc`, sau `new` (în C++).
- Principalele beneficii ale alocării dinamice includ flexibilitatea și economisirea resurselor de memorie.
- Variabilele alocate dinamic au o durată de viață scurtă și sunt eliberate atunci când nu mai sunt necesare.

2.4 Thread

Thread-urile sunt unități de execuție concurente care rulează în cadrul unui proces și împart resursele acestuia, precum memoria și descriptorii de fișiere. Thread-urile permit un program să efectueze mai multe operațiuni în paralel, îmbunătățind astfel eficiența și capacitatea de răspuns a aplicațiilor.

2.4.1 Context switch

Context switch reprezintă procesul prin care sistemul de operare întrerupe execuția unui thread și trece la execuția altui thread în cadrul aceluiași proces sau în cadrul altor procese. Acest mecanism este esențial în gestionarea execuției concurente a mai multor thread-uri și este declanșat de mai multe evenimente, precum întreruperi hardware, întreruperi de ceas (timer interrupts) sau apeluri de sistem.

Un context switch implică următoarele acțiuni:

- Salvarea contextului thread-ului curent, care include valorile înregistrărilor procesorului, registrele de stare, pointerii la stivă și alte informații relevante pentru execuție.
- Restaurarea contextului thread-ului următor, care implică încărcarea valorilor salvate ale înregistrărilor procesorului, registrelor de stare și altor date necesare pentru continuarea execuției thread-ului următor.
- Transferul de control către thread-ul următor, care reprezintă continuarea execuției acestuia de unde a fost întreruptă.

2.4.2 Migrarea thread-urilor

Migrarea thread-urilor se referă la transferul unui thread dintr-un nucleu de procesor sau de la o unitate de calcul (CPU) la alta. Aceasta se face pentru a distribui sarcinile în mod eficient pe mai multe nuclee sau CPU-uri și pentru a echilibra încărcarea sistemului. Migrarea thread-urilor este obișnuită în sistemele multiprocesor sau în medii de calcul distribuit, cum ar fi sistemele de calcul în cluster sau în cloud.

Există două tipuri principale de migrare a thread-urilor:

- Migrarea manuală: Programatorul are controlul asupra migrării thread-urilor și poate decide când și cum să mute un thread între nuclee sau CPU-uri. Acest tip de migrare poate fi folosit pentru optimizarea sarcinilor specifice sau pentru gestionarea resurselor în mod eficient.
- Migrarea automată: Sistemul de operare sau middleware-ul de gestionare a resurselor poate gestiona migrarea thread-urilor automat, bazându-se pe politici prestabilite sau pe evaluarea încărcării sistemului. Acest lucru este utilizat pentru a asigura un echilibru al încărcării și pentru a utiliza resursele disponibile în mod eficient.

Migrarea thread-urilor implică sincronizarea datelor și a contextului thread-ului, deoarece acesta poate avea date înregistrate și starea memoriei înainte de migrare. De asemenea, trebuie gestionate chestiuni precum comunicarea inter-thread și sincronizarea pentru a asigura funcționarea corectă și consistentă a thread-urilor după migrare.

3 Analiza

3.1 Măsurarea timpului de execuție

3.1.1 C++

std::chrono::high_resolution_clock este un tip de ceas în biblioteca standard C++ care oferă o rezoluție cât mai înaltă posibilă pentru a măsura timpul. Acest ceas este parte a bibliotecii chrono, introdusă începând cu standardul C++11, și oferă un cadru flexibil și portabil pentru lucrul cu timpul într-un mod consistent și precis în cadrul limbajului C++.

3.1.2 Python

Modulul time oferă funcționalități pentru manipularea timpului și măsurarea duratei. Una dintre funcțiile importante din acest modul este time.time(). Această funcție returnează timpul curent în secunde, măsurat de la un moment de referință fix (denumit epoch). Momentul de referință este 1 ianuarie 1970 la miezul nopții (UTC), iar time.time() furnizează timpul curent în secunde de la acea dată.

3.1.3 Rust

Std::time::Instant este un ceas nedescrescător. Acesta returnează Ok(Durata) în caz de succes si Err în caz de eșec.

3.2 Memoria dinamică

3.2.1 C++

Management-ul memoriei în C++ implică alocarea și dealocarea memoriei de către programator. Alocarea se realizează prin operatorul `new`, iar dealocarea prin `delete`.

3.2.2 Python

În Python, gestionarea memoriei este realizată în mare parte automat, datorită sistemului de gestionare a memoriei cunoscut sub numele de "Garbage Collection". Echivalentul alocării dinamice este apelarea constructorului, având în vedere că totul este un obiect.

3.2.3 Rust

În Rust, gestionarea memoriei dinamică este realizată cu precizie și securitate, datorită tipului `Box<T>` și a sistemului său de gestionare a proprietății de memorie. Acest tip inteligent permite alocarea și eliberarea automată a memoriei, evitând erorile comune asociate cu gestionarea memoriei, precum „double free". În plus, Rust oferă și alte tipuri, cum ar fi `Rc<T>` (Reference Counter) și `Arc<T>` (Atomic Reference Counter), pentru gestionarea partajării proprietății între multiple părți ale codului în mod concurent, asigurând o gestionare sigură a memoriei în toate situațiile.

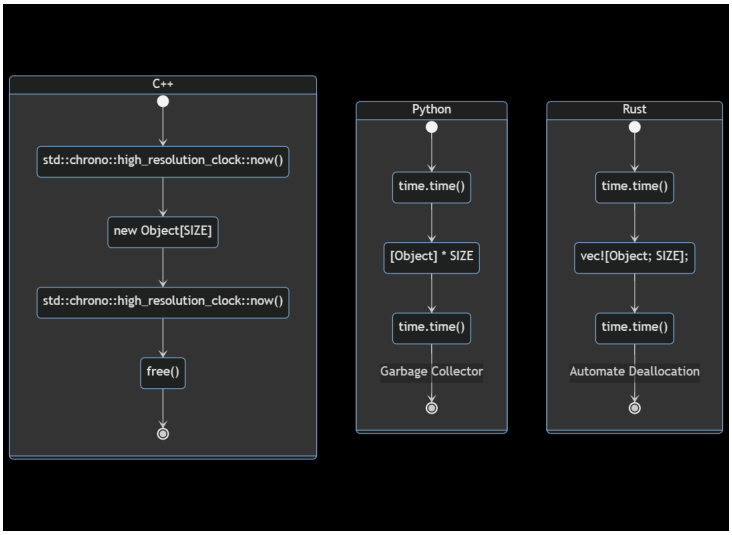

Figura 1: Alocarea dinamică

3.3 Crearea unui Thread

3.3.1 C++

`std::thread` este o clasă din biblioteca standard C++ (parte a standardului C++11 și versiunilor ulterioare) care permite crearea și gestionarea thread-urilor într-un mod portabil și eficient. Pe Windows, se folosește, de asemenea, `CreateThread` din API-ul nativ al acestui sistem de operare.

3.3.2 Python

În Python, crearea de thread-uri cu ajutorul modulului `threading` oferă o modalitate de gestionare a concurenței și de a executa cod în paralel. Python utilizează GIL (Global Interpreter Lock), care restricționează interpretorul să ruleze codul într-un singur thread la un moment dat. Acest lucru face ca thread-urile să nu fie potrivite pentru sarcini de procesare intensivă sau paralelă pe mai multe nuclee ale procesorului.

3.3.3 Rust

În limbajul Rust, crearea și gestionarea thread-urilor sunt realizate folosind modulul `std::thread`. Crearea unui thread implică definirea unei funcții care va fi executată în acel thread și utilizarea funcției `thread::spawn()` pentru a lansa thread-ul.

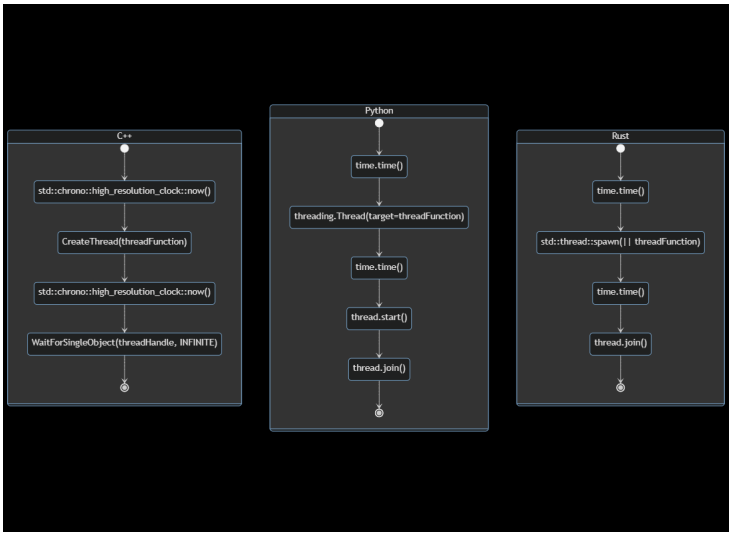

Figura 2: Crearea thread-urilor

3.4 Thread Context Switch

3.4.1 C++

În C++, comutarea de context este de obicei gestionată de planificatorul de sistem de operare. De exemplu, când folosiți `std::thread` pentru a crea și executa thread-uri, planificatorul de sistem alege când să comute contextul între thread-uri. Se pot utiliza funcții și biblioteci pentru a gestiona sincronizarea și comutarea de context în mod explicit, cum ar fi `std::condition_variable` sau `std::mutex`.

3.4.2 Python

În Python, comutarea contextului este gestionată de Blocarea Globală a Interpretorului (GIL). GIL permite unui singur fir să execute bytecode Python la un moment dat, ceea ce înseamnă că thread-urile Python nu pot utiliza în totalitate nucleele procesorului. Ca urmare, impactul comutării contextului este adesea mai puțin vizibil în Python datorită GIL, care serializează execuția firelor. O măsurare a acestei operații se poate realiza cu ajutorului modulului `psutil`. prin calcularea context-uri swtich-urilor dinaintea și după operația de sleep(0.1) (o durata scurta de timp care garanteaza cel puțin un context switch).

3.4.3 Rust

Rust, la fel ca C++, se bazează pe planificatorul de sistem de operare pentru comutarea contextului. Limbajul asigură siguranță datorită sistemului său de proprietate și împrumut, prevenind astfel cursele de date. Sistemul de proprietate din Rust reduce necesitatea blocărilor și a sincronizărilor manuale, ceea ce poate reduce costul comutării de context. Totodată, există mecanisme precum variabilele condiționale și mutex-urile.

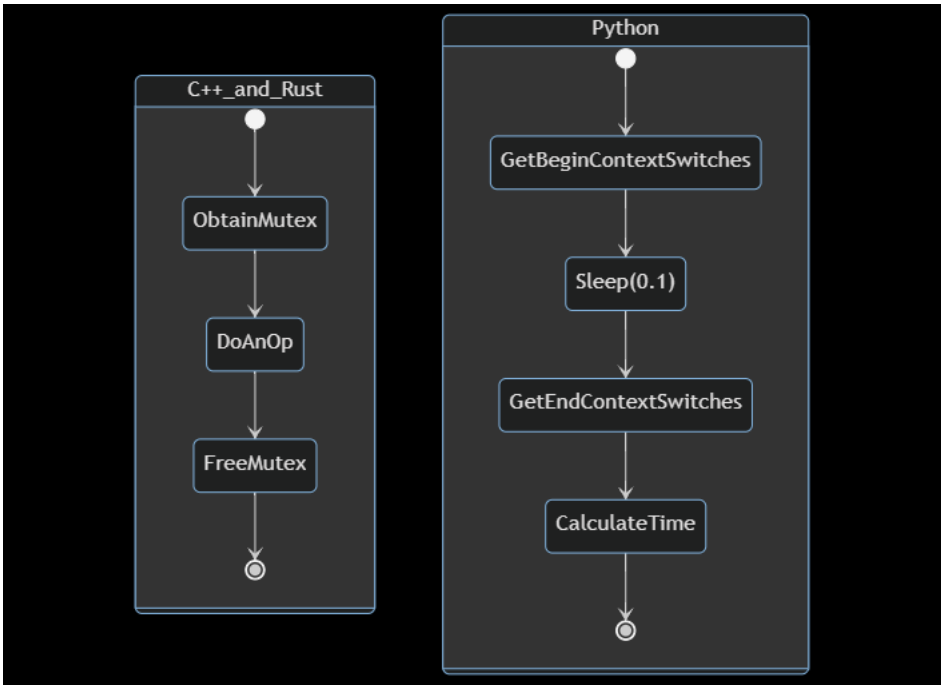

Figura 3: Context switch

3.5 Thread Migration

3.5.1 C++

Funcția `SetThreadAffinityMask` furnizează control asupra procesorului pe care un thread rulează, permițând programatorilor să optimizeze performanța aplicației prin distribuirea thread-urilor pe diferite nuclee de procesare.

3.5.2 Python

Funcția `psutil.Process(pid).cpu_affinity(new_affinity)` este utilizată în Python pentru a seta afinitatea CPU-ului pentru un anumit proces identificat prin PID (ID-ul procesului). Prin această metodă, putem controla pe care nuclee ale procesorului să ruleze procesul respectiv. Parametrul `new_affinity` specifică nucleele pe care procesul ar trebui să ruleze, iar funcția are rolul de a aplica această afinitate.

3.5.3 Rust

Funcția `set_thread_affinity(&cores)` este utilizată pentru a controla afinitatea firelor de execuție la nucleele specifice ale procesorului. Prin furnizarea unei referințe la un vector de nuclee (`cores`), programatorii pot specifica pe care nuclee să ruleze firele de execuție asociate.

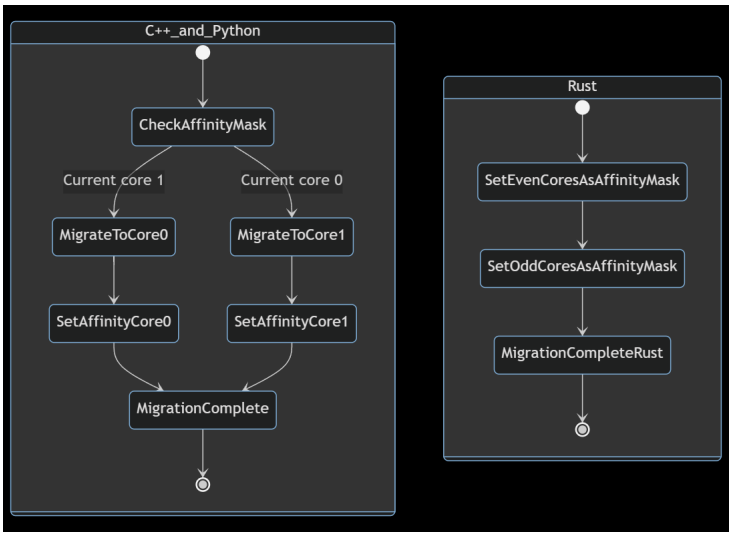

Figura 4: Migrarea thread-urilor

4 Design

4.1 Interfață Grafică

Aplicația trebuie să aibă o interfață grafică simplă și prietenoasă pentru utilizator, structurată astfel:

- **Panou de Selectare a Limbajului:** Listă de selecție pentru alegerea măsurătorii dorite.
- **Panou de Afișare a Rezultatelor:** Grafic cu performanța fiecărui limbaj împreună cu media.
- **Buton de Execuție:** Permite utilizatorului să inițieze execuția aplicației.

4.2 Backend

1. Instrumente pentru evaluarea duratei de execuție a codului sursă.
2. Module de execuție a codului: Pentru limbajele C++ și Rust, se utilizează sistemul de compilare pentru a produce un executabil. În cazul Python, se recurge la modulul `subprocess`.
3. Utilizarea fișierelor text pentru salvarea rezultatelor.
4. Componentă pentru interacțiunea cu interfața grafică: preluarea si prelucrarea datelor.

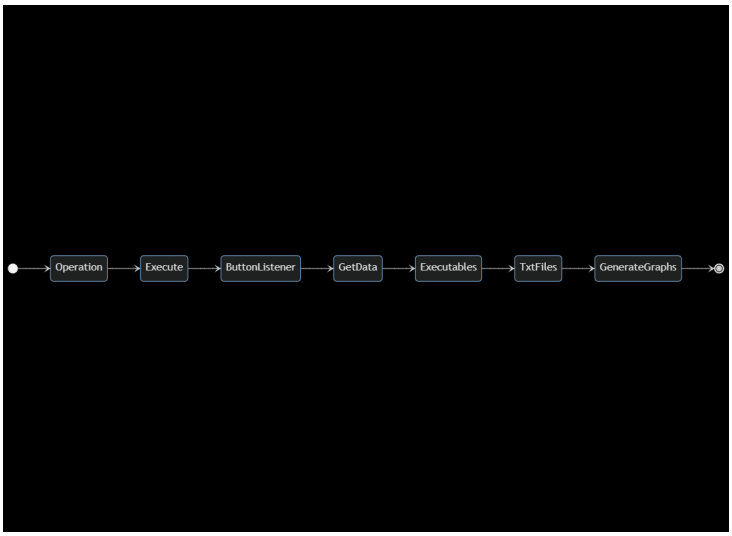

Figura 5: Ordinea execuției programului

5 Implementare

5.1 Operații cu memoria

5.1.1 C++

Codul C++ măsoară și compară timpul necesar pentru operațiile de alocare dinamică și acces la memorie, atât pentru un array static, cât și pentru unul dinamic. Mai întâi, se definește o funcție WriteNumberToFile care scrie un număr (în acest caz, timpul măsurat) într-un fișier specificat. Apoi, în funcția main, se declară un array static de 100,000 de elemente și se măsoară timpul pentru alocarea unui array dinamic de aceeași dimensiune.

Următorul bloc de cod măsoară timpul necesar pentru accesul la memorie pentru array-ul static și array-ul dinamic. Se utilizează două bucle for pentru a parcurge și a atribui valori elementelor fiecărui array. Timpul pentru aceste operațiuni este măsurat folosind biblioteca chrono, iar rezultatele sunt afișate în consolă.

La final, array-ul dinamic este șters pentru a evita memory leak-uri, iar timpul măsurat pentru fiecare operațiune este scris în fișiere de log corespunzătoare. Aceste fișiere de log conțin informații despre timpul necesar pentru alocarea dinamică, accesul la memorie statică și accesul la memorie dinamică, respectiv.

5.1.2 Python

Codul Python utilizează modulul `time` pentru a măsura și compara timpul necesar pentru operațiile de alocare a memoriei și acces la memorie, atât pentru un array static, cât și pentru unul dinamic. Funcția `write_number_to_file` scrie un număr specificat (în acest caz, timpul măsurat) într-un fișier specificat, gestionând posibilele erori de deschidere sau scriere în fișier.

5.1.3 Rust

În funcția principală `main`, un array static de 100,000 de elemente și un vector dinamic sunt inițializate cu zero. Se măsoară timpul necesar pentru alocarea dinamică de memorie și pentru accesul la memorie, atât pentru array-ul static, cât și pentru vectorul dinamic.

Măsurările timpului sunt efectuate folosind structura `Instant` din modulul `std::time`. Duratele rezultate sunt afișate în consolă și sunt, de asemenea, scrise în fișiere de log corespunzătoare (`"DynamicAlloc_log.rs.txt"`, `"StaticAccess_log.rs.txt"`, `"DynamicAccess_log.rs.txt"`).

5.2 Operații cu thread-uri și procese

5.2.1 C++

Înainte de funcția `main`, sunt incluse biblioteci necesare pentru manipularea timpului, gestionarea firelor de execuție și operarea cu fișiere. O variabilă partajată (`sharedVariable`) este definită pentru a fi modificată în mod concurent de către cele două fire de execuție. Un mutex (`mtx`) este folosit pentru a proteja regiunea critică în care se accesează variabila partajată.

Funcția `ThreadFunction` reprezintă funcția pe care o va executa fiecare fir de execuție creat. În această funcție, timpul de început este măsurat, iar apoi se realizează o iterație de 10 milioane de ori, unde variabila partajată este incrementată într-o regiune critică protejată de mutex. La final, se măsoară timpul de sfârșit al firului de execuție.

În `main`, se măsoară timpul pentru crearea a două fire de execuție cu ajutorul funcției `CreateThread`. Se măsoară și timpul asociat unui context switch prin blocarea și așteptarea mutexului pentru o perioadă scurtă de timp. Apoi, se măsoară timpul pentru migrația unui fir de execuție între nucleele CPU, ajustând afinitatea nucleelor. În cele din urmă, se așteaptă ca ambele fire de execuție să își finalizeze execuția folosind `WaitForSingleObject` și se închid handle-urile firelor de execuție cu `CloseHandle`.

5.2.2 Python

În funcția `get_context_switch_time()`, se măsoară timpul asociat unei comutări de context. Se obțin numărul de comutări de context înainte și după o scurtă perioadă de așteptare, iar diferența este folosită pentru a calcula timpul mediu pe comutare de context.

În secțiunea principală a scriptului, se măsoară timpul pentru crearea a două fire de execuție, fiecare apelând funcția `thread_function` cu argumente diferite. Se afișează durata medie pentru crearea firelor de execuție. Apoi, se măsoară timpul asociat unei comutări de context folosind funcția `get_context_switch_time()`.

În continuare, se măsoară timpul asociat migrației unui fir de execuție între nucleele CPU. Se realizează acest lucru prin schimbarea afinității procesului la un alt nucleu.

La final, scriptul așteaptă ca ambele fire de execuție să își finalizeze execuția folosind metoda `join()` și scrie duratele măsurate în fișiere de jurnal pentru analiza ulterioară.

5.2.3 Rust

Funcția `main` începe prin a măsura timpul necesar pentru a crea un fir de execuție folosind funcția `thread::spawn`. Handle-ul firului de execuție generat este stocat pentru utilizare ulterioară.

În continuare, codul măsoară timpul pentru o comutare de context prin incrementarea variabilei partajate în contextul firului de execuție principal.

Ulterior, codul măsoară timpul pentru migrația firului de execuție. Acest lucru se realizează prin setarea afinității firului de execuție pentru nucleele CPU specifice cu ajutorul funcției `set_thread_affinity`. Durata acestei operațiuni este apoi afișată.

În cele din urmă, firul de execuție principal așteaptă ca firul de execuție generat să-și finalizeze execuția folosind `handle.join().unwrap()`.

6 Testare

Prin implementarea diferitelor metode de măsurare a timpului, avem posibilitatea să comparăm rezultatele obținute prin aceste metode și să identificăm eventualele diferențe. În situația în care constatăm diferențe semnificative între valorile obținute, este recomandat să ignorăm rezultatele și să reluăm testele.

Diferențele de performanță între Python, Rust și C++ provin în principal din modul în care aceste limbaje de programare sunt proiectate și implementate, precum și din caracteristicile lor particulare.

Python, ca limbaj interpretat, este mai lent în execuție din cauza interpretării linie cu linie, iar Global Interpreter Lock (GIL) poate afecta eficiența în scenarii de multithreading. Rust se distinge prin siguranța și controlul asupra memoriei, renunțând la un Garbage Collector. Cu toate acestea, abordarea riguroasă a siguranței poate aduce o complexitate crescută. C++ oferă control detaliat asupra hardware-ului și eficiență, dar acest nivel de control aduce riscuri de erori și complexitate.

Operation	**C++**	**Python**	**Rust**
Alocare dinamica	52.974	623.99	167.682
Acces static	77.934	911.598	6173.836
Acces dinamic	52.974	1891.956	6173.836
Crearea firelor de execuție	23.288	29.124	29630.654
Context switch	5.984	597.988	12612.530
Migrarea firelor de execuție	5.922	8.554	407.371

Tabela 1: Comparație de performanță în microsecunde

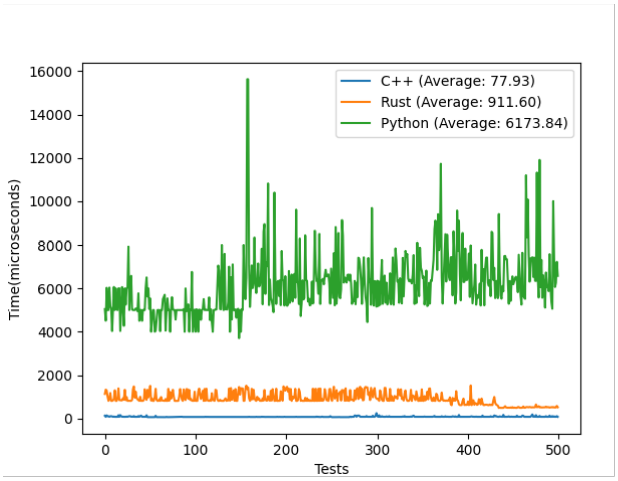

Figura 6: Acces static

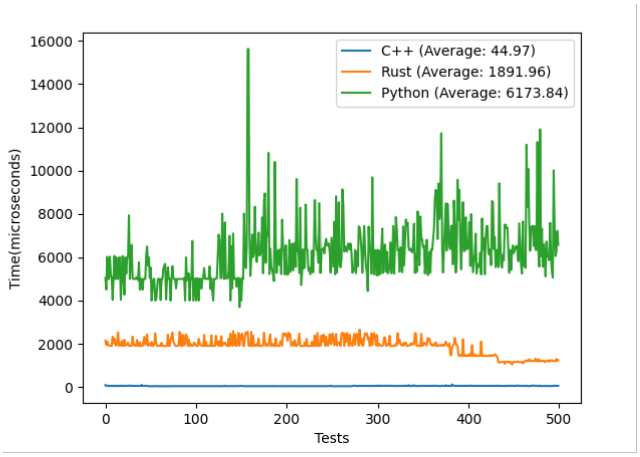

Figura 7: Acces dinamic

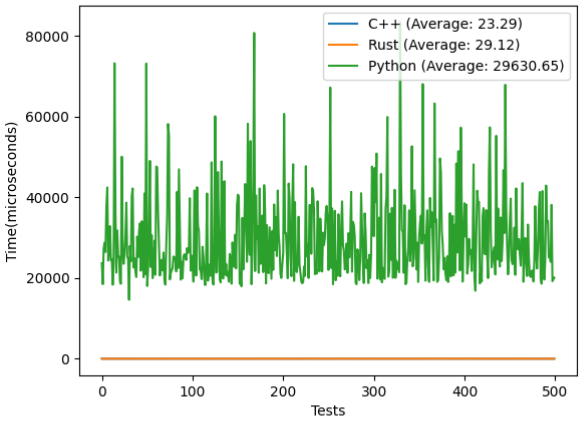

Figura 8: Thread Creation

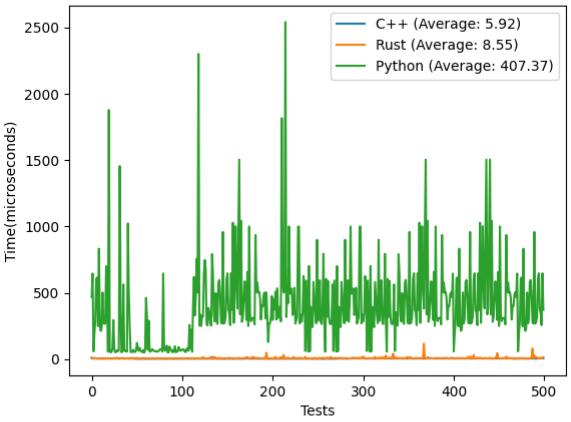

Figura 9: Migrarea thread-urilor

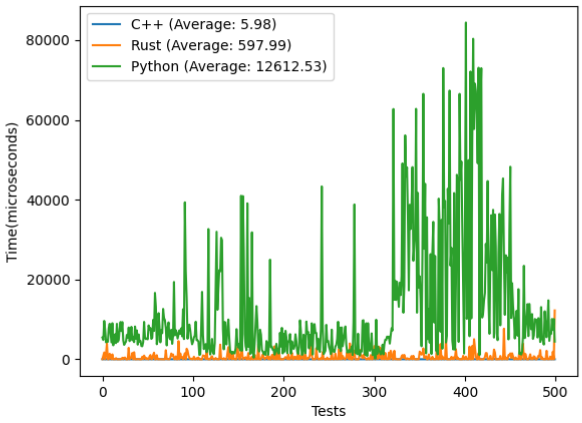

Figura 10: Context switch

7 Concluzie

Diferența de performanță între cele trei limbaje poate fi sumarizată în felul următor. C++ se distinge prin rapiditatea sa, dar acest avantaj vine la costul apariției erorilor în timpul dezvoltării software-ului, datorită necesității gestionării manuale a memoriei. Cu toate acestea, punctele sale forte includ apropierea strânsă de hardware. Rust, deși cu un cost de performanță ușor mai ridicat, oferă securitate și robustețe sporite în cod, compensând dezavantajul complexității limbajului. Pe de altă parte, Python este cel mai lent dintre cele trei, dar reușește să ofere cod extrem de lizibil și un timp de dezvoltare mai scăzut, contrabalansând într-o anumită măsură performanța redusă. Fiecare limbaj aduce propriile sale trade-off-uri, iar alegerea între ele depinde de prioritățile specifice ale proiectului și cerințele sale.

8 Dezvoltări ulteriore

Integrarea cu instrumente externe, cum ar fi Valgrind și ThreadSanitezer, ar fi o posibilă dezvoltare ulterioară. În plus, compatibilitatea multiplatformă este necesară pentru a asigura o experiență de utilizare uniformă indiferent de sistemul de operare sau arhitectura de procesor utilizată. De asemenea, extinderea suportul pentru alte limbaje de programare ar oferi utilizatorilor posibilitatea de a beneficia de funcționalități avansate în medii diferite și costul utilizării acestora.

Referințe

[1] Steve Klabnik și Carol Nichols, *The Rust Programming Language*, No Starch Press, 2018.

[2] Microsoft, *Processes and Threads*, Available online at Microsoft Learn.

[3] *Python Documentation*, Available online at Python.org.

[4] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 2013.