UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Măsurarea timpului de execuție a proceselor în diferite limbaje de programare

Structura Sistemelor de Calcul

Nistor Dalia
Grupa 30234

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

2023-2024

Cuprins

1.Introduction

1.1Context

Obiectivul acestui proiect este sa masoare executia in timp, a diferitelor operatii, in 3 limbaje de programare diferite si sa analizeze cat de bine se comporta acestea pentru diferite atirbutii. Astfel, putem sa aproximam cat de mare este diferenta la o scala mai inalta cum ar fi un proiect al unei companii cu mii de linii de cod si sa vedem care este mai performant.

Masuram timpul de executie al programelor cu timpul procesorului CPU (cantitatea de timp pentru care a fost folosita o unitate centrala de procesare pentru procesarea instructiunilor unui program de calculator) si se masoara in ceasuri de clock.

1.2 Motivatie

Masuratorile o sa fie facute in diferite IDE's. Am ales sa masor timpul de executie in JAVA, PYTHON si C. Pentru Java o sa folosesc IntelliJ IDEA, iar pentru Python si C Visual Studio Code.

1.3 Obiective

Designul si implementarea a 3 programe mici care masoara operatiile urmatoare: alocarea memoriei, accesul la memorie (static si dynamic), crearea unui thread si thread context switch. Programele vor fi identice iar singura diferenta va fi IDE-ul si limbajul de programare. O sa masuram aceleasi operatii de un numar finit de ori si vom calcula timpul mediu pentru fiecare ca mai apo isa comparam rezultatele obtinute si sa alegem care dintre cele 3 limbaje de programare este cel mai efficient in orice circumstanta.

2. Bibliographic study

În general, măsuram timpul de execuție a programului de la initiere la prezentarea unor intrări până la terminarea livrării ultimelor ieșiri. Câteva măsurători diferite ale performanței software-ului sunt de interes:

• Worst case execution time (WCET) - cel mai lung timp de execuție pentru orice combinație posibilă de intrări

• Best case execution time (BCET) - cel mai scurt timp de execuție pentru orice combinație posibilă de intrări

• Average case execution time (ACET) - pentru intrări tipice

Timpul mediu de executie (Average time) este folosit in scopuri mult mai diferite fata de timpul cel mai bun (Best time) sau timpul cel mai rau (Worst time)

Datele, de performanță medie, sunt deosebit de utile atunci când sunt înregistrate, nu doar pentru întregul program, ci și pentru bucati de program. Comportamentul mediu al cazurilor poate fi folosit pentru a identifica punctele slabe ale fiecarui program. Acestea pot fi datorate algoritmilor slabi, alegerii gresite a instructiunilor (din punct de vedere al performantei) sau alte cauze.

Pentru fiecare limbaj de programare aplicam același principiu de măsurare a timpului de execuție pentru o anumită parte a programului. Mai întâi stocăm timpul curent într-o variabilă (folosind metoda potrivită pentru limbajul nostru), apoi executam partea pe care vrem să o măsurăm, iar imediat după execuție stocăm oră curentă într-o altă variabilă. Diferența dintre ultima și prima variabilă domeniul nostru de execuție pentru cod.

Există două moduri de măsurare a timpului de execuție scurs în Java, fie utilizam System.currentTimeMillis() sau utilizam System.nanoTime() - ambele returnează timpul curent al programului în milisecunde și nanosecunde - aceste două metode pot fi folosite pentru a măsura timpul scurs sau de execuție între două apeluri de acțiune sau evenimente în Java.

În Python putem măsura the wall time (diferența dintre momentul la care un program și-a încheiat execuția și momentul la care a început programul. Include și timpul de așteptare pentru resurse) și timpul CPU (timpul CPU-ului a fost ocupat cu procesarea instrucțiunilor). programului.Timpul petrecut în așteptarea finalizării altei sarcini (cum ar fi operațiunile I/O) nu este inclus în timpul CPU.Nu include timpul de așteptare pentru resurse). Funcțiile time.time() (pentru wall time) și time.process_time() (pentru timpul CPU) returnează ora curentă în secunde.

Există mai multe metode pentru a găsi timpul de execuție în limbajul C, dar cea mai des folosită este funcția clock() (returnează numărul total de tick-uri de ceas). Pentru a converti timpul în secunde, împărțim ticurile de ceas în macrocomandă CLOCKS_PER_SEC.

3.Analysis

3.1 Alocarea memoriei

Alocarea memoriei este un proces prin care programele și serviciile de calculator sunt asignate cu spațiu de memorie fizic sau virtual. Alocarea memoriei se face fie înainte, fie în timpul executării programului.

JAVA - Alocarea statică a memoriei în Java: În alocarea statică a memoriei, trebuie să declarăm variabilele înainte de a executa programul. Memoria statică este alocată în timpul compilării.

-Alocarea dinamică a memoriei în Java: Alocarea dinamică a memoriei în Java înseamnă că memoria este alocată obiectelor Java în timpul rulării sau executării programului. Este în contradicție cu alocarea statică a memoriei. Alocarea dinamică a memoriei are loc în spațiul heap. Spațiul heap este locul unde sunt create întotdeauna obiectele noi și referințele lor sunt stocate în memoria stivei.

C - Memoria statică este alocată pentru variabilele declarate de către compilator. Alocarea memoriei are loc în timpul compilării și se realizează înainte de executarea programului. În alocarea statică a memoriei, variabilele sunt alocate permanent, până când programul se execută sau apelul unei funcții se încheie, și utilizează stiva.

-În alocarea dinamică a memoriei, variabilele sunt alocate doar dacă unitatea programului devine activă. Alocarea dinamică a memoriei se face în timpul execuției programului și utilizează heap-ul.

PYTHON - În Python, memoria este gestionată de managerul de memorie al Pythonului, care determină unde să plaseze datele aplicației în memorie. Prin urmare, trebuie să avem cunoștințe despre managerul de memorie Python pentru a scrie cod eficient și ușor de întreținut.

- Structura de date, Stiva, este folosită pentru a stoca memoria statică. Este necesară doar în cadrul anumitor funcții sau apeluri de metode. Funcția este adăugată în stiva de apeluri a programului ori de câte ori o apelăm.

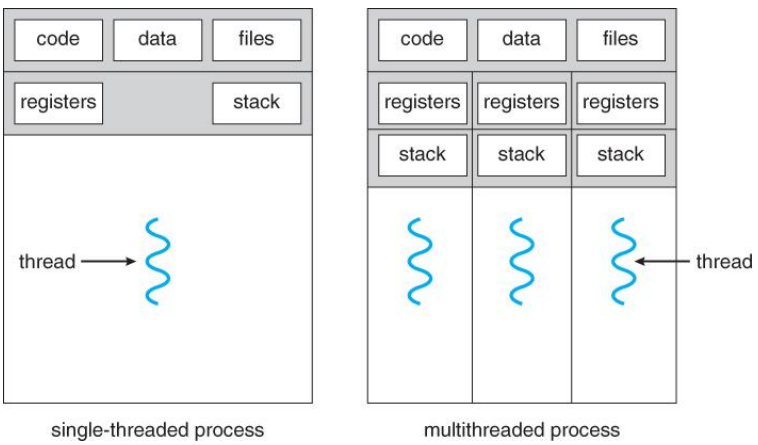
-Știind că totul în Python este un obiect, înseamnă că alocarea dinamică a memoriei inspiră gestionarea memoriei în Python. Managerul de memorie Python dispare automat atunci când obiectul nu mai este în uz. Să presupunem că există două sau mai multe variabile care conțin aceeași valoare, astfel că mașina virtuală Python nu creează un alt obiect cu aceeași valoare în heap-ul privat. De fapt, face ca a doua variabilă să indice către valoarea deja existentă în heap-ul privat. Acest lucru este extrem de benefic pentru a economisi memoria, care poate fi utilizată de o altă variabilă.

3.2 Accesul memoriei

În cazul memoriei de calculator, timpul de acces reprezintă cantitatea de timp necesară procesorului computerului pentru a citi date din memorie. Putem accesa memoria statică (alocată pe stivă) sau memoria dinamică (alocată pe heap) a unui program. Pentru a măsura accesul la memorie, am decis să suprascriu fiecare element al unei matrice de numere întregi. Pentru a accesa memoria statică, am declarat o matrice în interiorul unei funcții/metode, iar pentru memoria dinamică am utilizat obiecte (în Java, instanțiate cu "new") și am creat matrice alocate dinamic în limbajele C și Python.

3.3 Crearea thread-urilor

Toți programatorii sunt familiarizați cu scrierea programelor secvențiale (programe care execută sarcini una câte una până la final). Adică, fiecare are un început, o secvență de execuție și un sfârșit. În orice moment pe durata execuției programului, există un singur punct de execuție. În informatică, un fir de execuție este cea mai mică secvență de instrucțiuni programate care poate fi gestionată independent de un program scheduler, care este în mod obișnuit o parte a sistemului de operare. Implementarea firelor de execuție și a proceselor diferă între sistemele de operare, dar în majoritatea cazurilor un fir de execuție este o componentă a unui proces. Mai multe fire de execuție ale unui proces dat pot fi executate concurent (prin capacitatea de multithreading), partajând resurse precum memoria, în timp ce procesele diferite nu partajează aceste resurse.



single-threaded process multithreaded process

JAVA - În Java există două modalități de a crea un fir de execuție: prin extinderea clasei Thread sau prin implementarea interfeței Runnable.

-Clasa Thread furnizează constructori și metode pentru a crea și efectua operații pe un fir de execuție. Clasa Thread extinde clasa Object și implementează interfața Runnable.

-Interfața Runnable ar trebui să fie implementată de orice clasă a cărei instanțe sunt destinate să fie executate de un fir de execuție. Interfața Runnable are doar o metodă numită run().

-Indiferent de modul dorit de creare a unui fir de execuție, în ambele cazuri trebuie să suprascriem metoda "run()", deoarece aceasta descrie comportamentul firului nostru de execuție. Pentru a reține: într-un program cu mai multe fire de execuție ar trebui întotdeauna să pornim firele de execuție cu metoda start(), pentru că dacă apelăm direct metoda run(), programul va fi executat secvențial.

C - Pentru a utiliza firele de execuție în C, este suficient să includem antetul specific și să folosim funcțiile preimplementate de acolo. Fișierul antet "pthread.h" conține declarații de funcții și asignări pentru interfețele de thread-uri și definește o serie de constante folosite de acele funcții. La crearea unui fir de execuție (folosind pthread_create), specificăm, de asemenea, funcția pe care trebuie să o execute și argumentele pe care dorim să le transferăm funcției sale. Spre deosebire de Java, nu trebuie să pornim firul de execuție cu o comandă suplimentară.

PYTHON - Crearea firelor de execuție în Python este foarte similară cu crearea firelor de execuție în C. Trebuie să importăm clasa "Thread" din modulul "threading" și apoi să o creăm folosind: our_thread = Thread(target, args ...). La creare, firul de execuție primește ținta sa (funcția de executat) și unele argumente (opționale). Biblioteca "threading" are, de asemenea, mai multe funcții preimplementate pe care le putem utiliza în programul nostru pentru a controla firele de execuție.

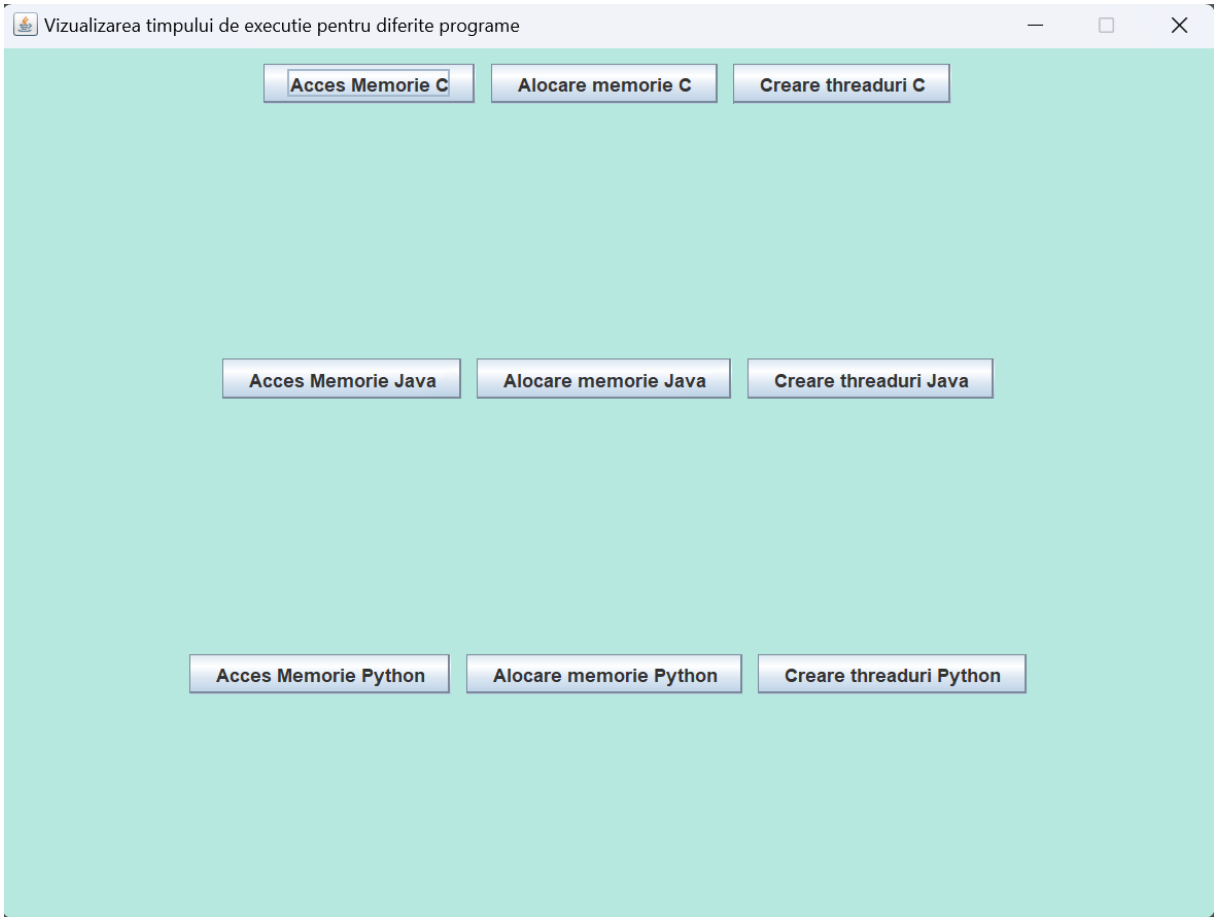
4.Design

Aplicația are o interfață grafică simplă construită cu Java Swing. Fereastra principală conține trei seturi de butoane pentru limbajele de programare C, Java și Python. Butoanele sunt grupate în panouri, fiecare set având câte trei butoane pentru diferite acțiuni legate de accesul la memorie, alocarea de memorie și crearea de fire de execuție.

Design-ul aplicației este centrat în jurul unei organizări simple și intuitive, facilitând utilizatorului accesul la informațiile relevante pentru diferitele operațiuni ale limbajelor de programare.

Aplicatia permite utilizatorului să acceseze informații despre timpul de execuție pentru diferite operațiuni scrise în limbajele de programare C, Java și Python. Apăsarea butoanelor asociate cu fiecare limbaj deschide fișiere text care conțin măsurători specifice pentru accesul la memorie, alocarea de memorie și crearea de fire de execuție în respectivele limbaje. Prin deschiderea acestor fișiere, utilizatorul poate examina detaliile performanței fiecărui limbaj într-un mod simplu și accesibil.

Această funcționalitate oferă o modalitate convenabilă de analiză a rezultatelor măsurătorilor pentru fiecare limbaj, facilitând înțelegerea și compararea performanțelor în funcție de tipul de operațiuni efectuate.



5.Implementation

Am decis să scriu fiecare cod sursă într-un mod cât mai generic posibil, astfel încât, dacă dorim să măsurăm o anumită operație pentru diferite valori (de exemplu, poate dorim să alocăm memorie pentru mai multe elemente), trebuie să modificăm doar o valoare dintr-un fișier text în care sunt stocate datele despre măsurători. Prin urmare, în fiecare fișier sursă am utilizat o funcție care deschide fișierul și citește parametrul sau parametrii pentru măsurători. Pentru fiecare operație, am efectuat exact 10 măsurători și am calculat timpul mediu la final. Pentru măsurarea timpului de execuție, am utilizat:

(Java): System.nanoTime() și apoi am convertit nanosecundele în secunde.
(C): clock() și apoi am împărțit rezultatul final la constanta CLOCKS_PER_SEC.
(Python): time.time().

5.1 Alocarea memoriei

Am utilizat un tablou cu 100.000 de elemente și am măsurat timpul. Timpul de început a fost obținut imediat înainte de bucla în care am introdus valorile în tablou, iar timpul de sfârșit a fost setat imediat după aceea. Toate celelalte operații nu au fost măsurate în timp.

JAVA - Metoda readConstantsFromFile: Această metodă este responsabilă pentru citirea parametrilor dintr-un fișier text (Masuratori.txt). Se utilizează un obiect BufferedReader

pentru a citi liniile din fișier. Parametrul NUMBER_OF_INTEGERS este setat pe baza valorii citite din fișier.

-Metoda main: În main, se apelează readConstantsFromFile pentru a inițializa constanta NUMBER_OF_INTEGERS din fișierul de configurare. Se inițializează un obiect FileWriter pentru a scrie rezultatele măsurătorilor într-un fișier (AlocareaMemoriei_Java.txt). Se adaugă un antet în fișier cu informații despre limbajul de programare și tipul de operație măsurată. Se inițializează un obiect de tip ArrayList<Integer> pentru a stoca elementele alocate dinamic. Se măsoară timpul necesar pentru alocarea memoriei folosind un ciclu while, unde se adaugă elemente în listă și se măsoară intervalul de timp utilizând System.currentTimeMillis(). Media timpilor este calculată pentru cele 10 iterații și este scrisă în fișierul de rezultate. Optimizări și

-Eliberarea Memoriei: După fiecare iterație, lista este curățată (v.clear()) pentru a elibera memoria ocupată și se forțează apelul colectorului de gunoi (System.gc()). Se verifică dacă intervalul de timp măsurat este diferit de zero pentru a evita considerarea cazurilor în care operația poate fi instantanee.

C - Am alocat un tablou dinamic cu malloc și am setat valorile pentru fiecare element, apoi am eliberat memoria pentru a începe din nou. Principala diferență în limbajul C a fost că a trebuit să repet acest proces de mai multe ori pentru a obține un timp de execuție mai mare de 0,0 secunde. Astfel, pentru fiecare măsurătoare, am alocat memorie pentru tablou și am setat valorile de 100 de ori, iar timpul final a fost împărțit la 100.

PYTHON - Inițial, am declarat un tablou gol. Pentru fiecare măsurătoare, am adăugat elemente în tablou, am calculat timpul, am curățat tabloul și am apelat colectorul de gunoi folosind gc.collect() pentru a ne asigura că totul funcționează corect.

5.2 Accesul la memorie

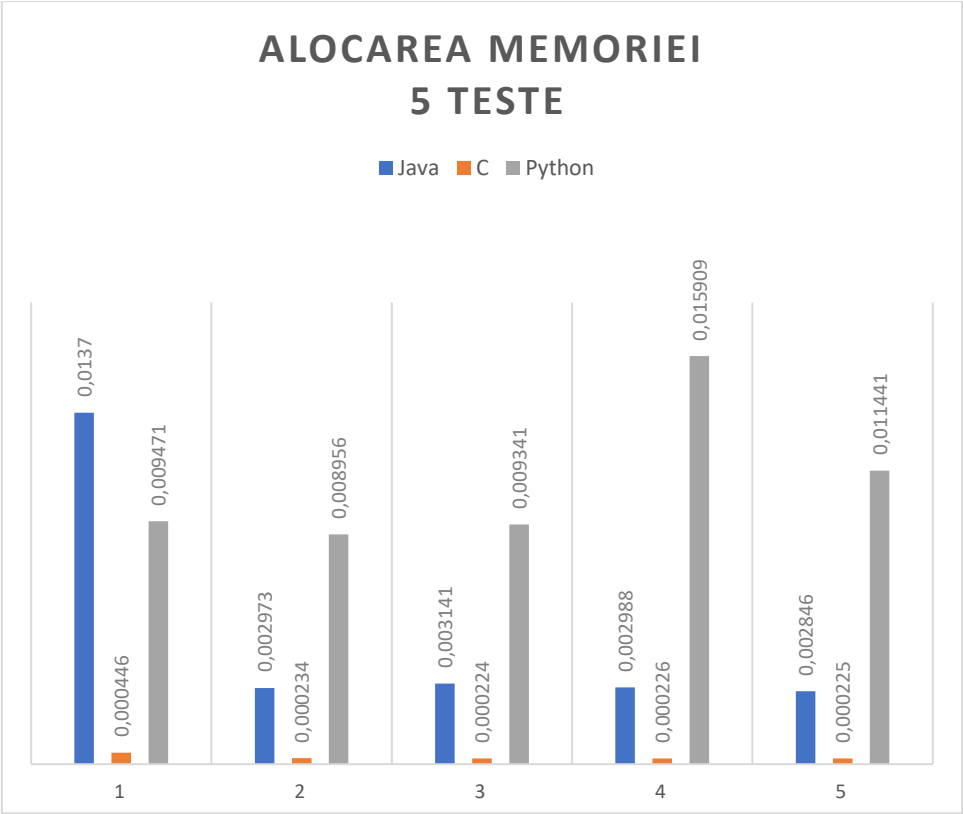
Pentru aceasta, am utilizat, de asemenea, tablouri cu 100.000 de elemente, inițializând fiecare element cu valoarea întregului 0. Măsurătorile constau în a seta fiecare element să fie egal cu indexul său și au fost realizate atât pe tablouri alocate static, cât și pe cele alocate dinamic. Pentru tablourile statice, strategia mea a fost să creez o funcție/metodă (pentru a utiliza memoria stivei, unde valorile statice sunt stocate), unde am declarat un tablou și apoi am setat fiecare element al său, măsurând timpul în interiorul acelui bloc de funcție. Pentru tablourile dinamice, am alocat un tablou la fel ca mai sus, în secțiunea de alocare a memoriei, și am efectuat aceleași modificări.

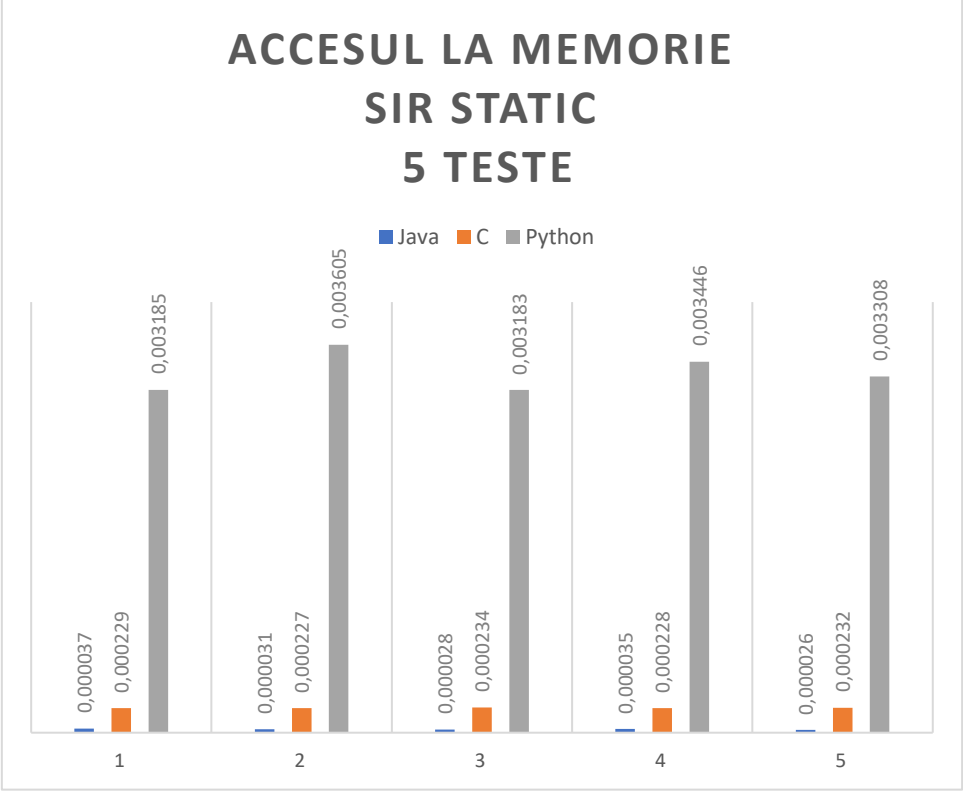
5.3 Crearea thread-urilor

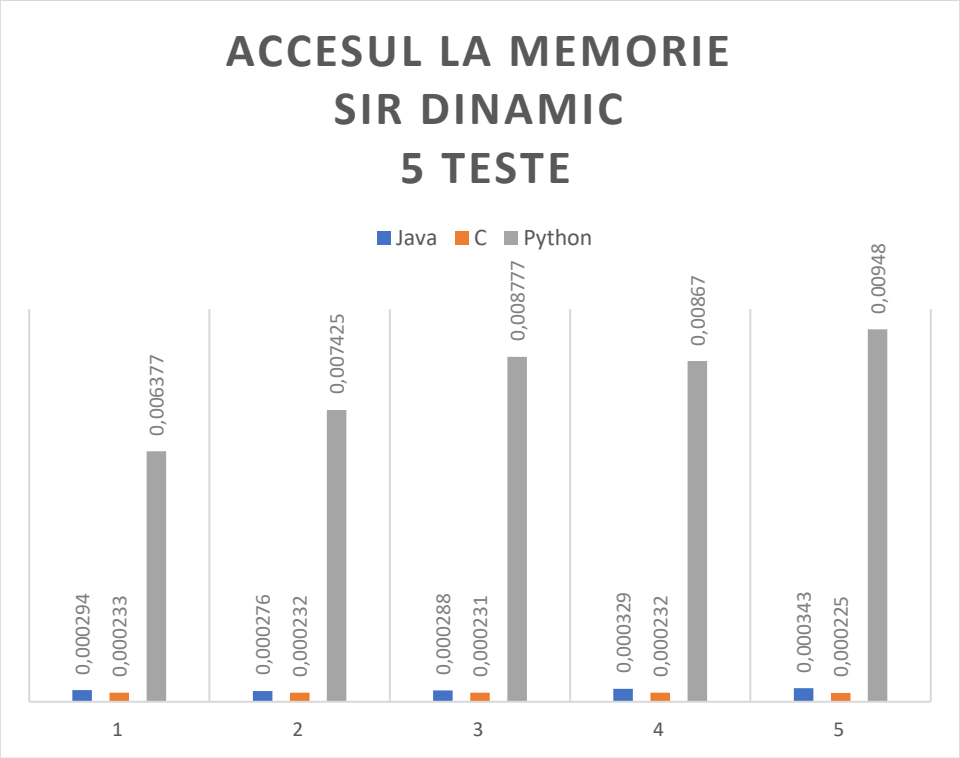
Pentru acest scop, am decis să creez doar un tablou cu 5000 de fire de execuție. Firele de execuție nu au nicio sarcină specifică, nu primesc nimic de făcut; dorim doar să măsurăm timpul necesar pentru a le crea.

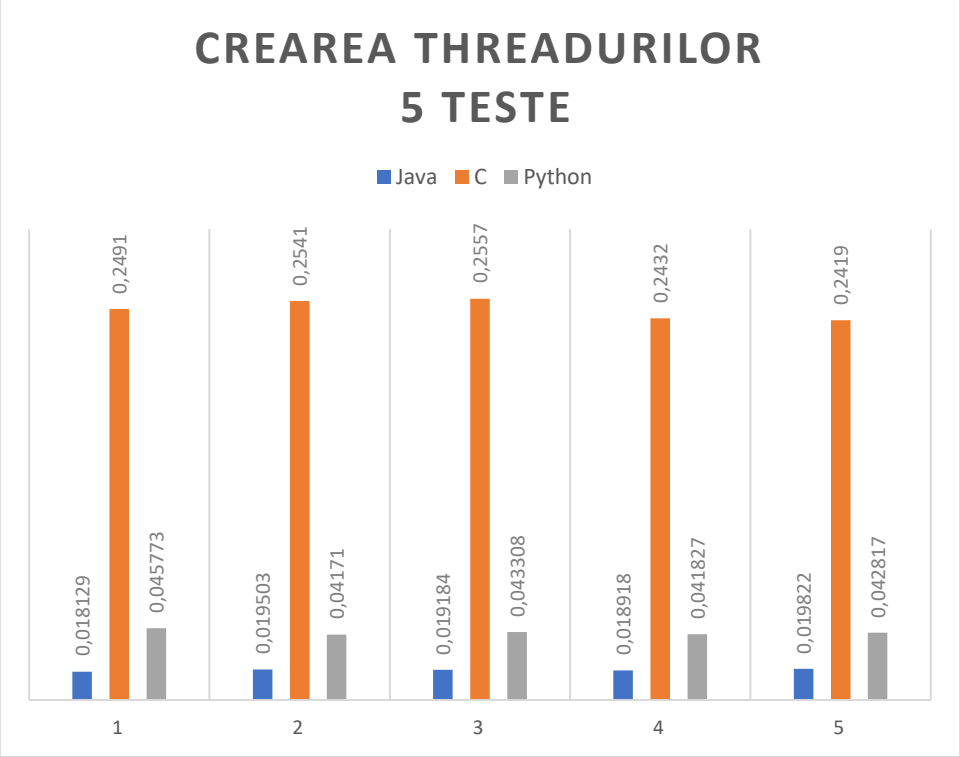
6.Testing and validation

Am executat de cate 5 ori fiecare program si am pus informatia intr-o diagrama pentru a se vedea mai bine diferentele.






ACCESUL LA MEMORIE
SIR DINAMIC
5 TESTE
Java
C
Python
0,000294
0,000233
0,006377
0,000276
0,000232
0,007425
0,000288
0,000231
0,008777
0,000329
0,000232
0,00867
0,000343
0,000225
0,00948
1
2
3
4
5


CREAREA THREADURILOR
5 TESTE
Java
C
Python
0,018129
0,2491
0,045773
0,019503
0,2541
0,04171
0,019184
0,2557
0,043308
0,018918
0,2432
0,041827
0,019822
0,2419
0,042817
1
2
3
4
5

7. Conclusions

Concluzia mea legată de măsurători este că limbajul de programare C a fost cel mai constant în timp în execuții (cel mai puțin fluctuant în timp), în timp ce Java și Python au fost mai volatile (diferențe mai mari în timp de la o măsurare la alta). Pentru alocarea și accesarea memoriei, cea mai bună opțiune este limbajul C, iar cea mai slabă este Python. Dar când vine vorba de multiprocesare (crearea și lucrul cu mai multe fire de execuție), Java și Python sunt mult mai optimizate pentru acest tip de lucru, în timp ce C necesită un timp semnificativ mai lung pentru a realiza aceste operațiuni.

8.Bibliography

1. https://www.w3schools.com/java/java_threads.asp
2. https://www.geeksforgeeks.org/multithreading-in-c/
3. https://realpython.com/intro-to-python-threading/
4. https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
5. https://javachallengers.com/memory-allocation-with-java/
6. https://www.geeksforgeeks.org/memory-management-in-python/
7. https://www.javatpoint.com/java-swing