UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Facultatea de Automatică și Calculatoare-Calculatoare

STRUCTURA SISTEMELOR DE CALCUL

Măsurarea timpului de execuție a proceselor în diferite limbaje de programare (C, C++, Java)

Student: Niță Alexandru-Gabriel

Grupa 30233

Profesor coordonator: Neagu Mădălin

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Cuprins

1.Introducere

1.1 Context

Performanța și eficiența sunt componente esențiale ale dezvoltării de software în domeniul programării. Timpul de execuție al unui program sau timpul total necesar pentru a efectua o sarcină, este unul dintre elementele cheie de performanță.

Pentru programatori, măsurarea timpului de execuție joacă un rol important, deoarece oferă o imagine de ansamblu a eficienței unui program. Acest lucru devine deosebit de important atunci când vorbim de dezvoltarea de software, deoarece optimizarea performanței are un impact mare atât asupra experienței utilizatorilor, cât și asupra cheltuielilor de operare in ceea ce privește produsul final.

Pe lângă sintaxă și funcționalitate, programatorii trebuie să se gândească și la modul în care limbajul de programare pe care îl aleg va afecta timpul în care rulează codul lor.

1.2 Motivație

Înțelegerea modului în care diverse limbaje de programare gestionează în detaliu probleme precum alocarea memoriei, accesul la memorie, crearea și gestionarea thread-urilor(firelor de execuție) este foarte importantă pentru a scrie un software eficient. Intenția acestui proiect este de a oferi o analiză comparativă a timpului de execuție al proceselor în limbajele de programare C, C++ și Java.

1.3 Obiective

Obiectivul principal al acestui proiect este de a oferi o analiză comparativă a timpului de execuție al proceselor în limbajele de programare C, C++, și Java. De asemenea, un scop suplimentar îl poate reprezenta atenția asupra avantajelor și dezavantajelor fiecărui limbaj în acest context.

Prin măsurarea și compararea timpilor de execuție în cele trei limbaje, vom evidenția anumite caracteristici ale fiecărui limbaj. În timp ce C și C++ sunt renumite pentru controlul lor complex asupra resurselor hardware, Java asigură gestionarea automată a memoriei și portabilitatea.

Etape importante în îndeplinirea obiectivelor acestui proiect:

- **Măsurarea alocării memoriei**: Examinarea efectelor asupra performanței metodelor de alocare statică și dinamică a memoriei utilizate de fiecare limbaj.

- **Măsurarea accesului la memorie**: Examinarea timpului necesar pentru a accesa memoria în scenarii de citire și scriere și observarea diferențelor specifice fiecărui limbaj.

- **Crearea și gestionarea firelor de execuție**: Calcularea timpului necesar pentru a porni un nou fir de execuție, pentru a muta firele de execuție existente între procesoare și pentru a schimba contextul între ele.

- **Compararea performanțelor**: Compararea măsurătorilor în limbajele de programare C, C++ și Java, remarcând diferențele majore între cele trei limbaje.

2.Studiu bibliografic

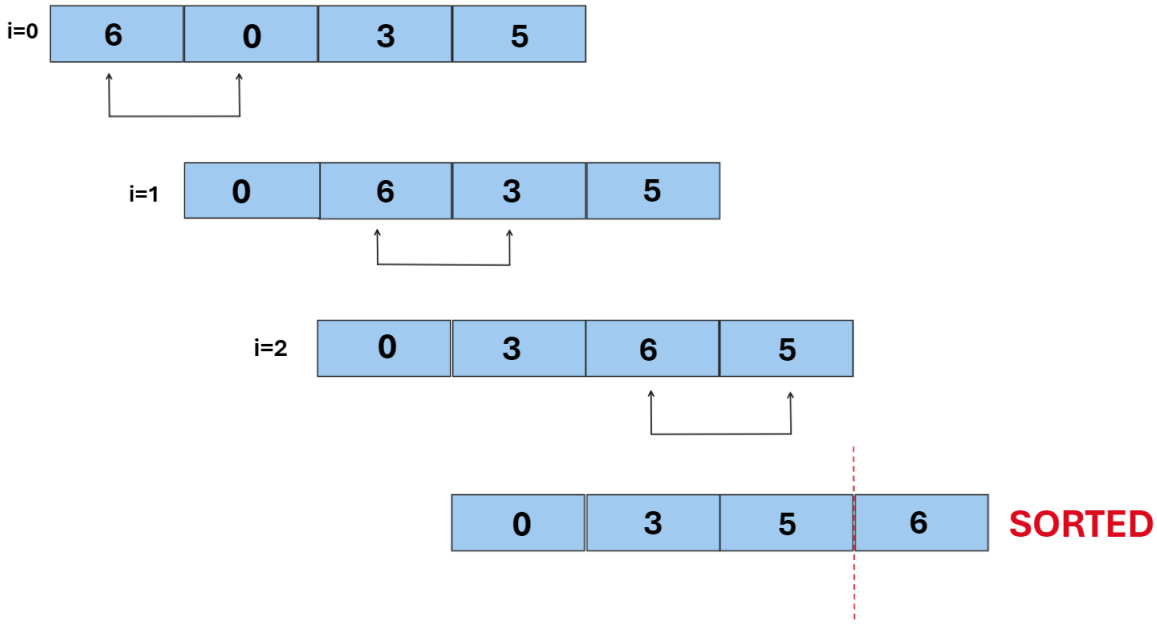
Materiale

- **Extension Pack for Java:**https://code.visualstudio.com/docs/languages/java
- **C/C++ extension :**https://code.visualstudio.com/docs/languages/cpp
- **BubbleSort** https://www.geeksforgeeks.org/bubble-sort/
- **How to measure time taken:** https://www.geeksforgeeks.org/how-to-measure-time-taken-by-a-program-in-c/
- https://www.geeksforgeeks.org/garbage-collection-java/

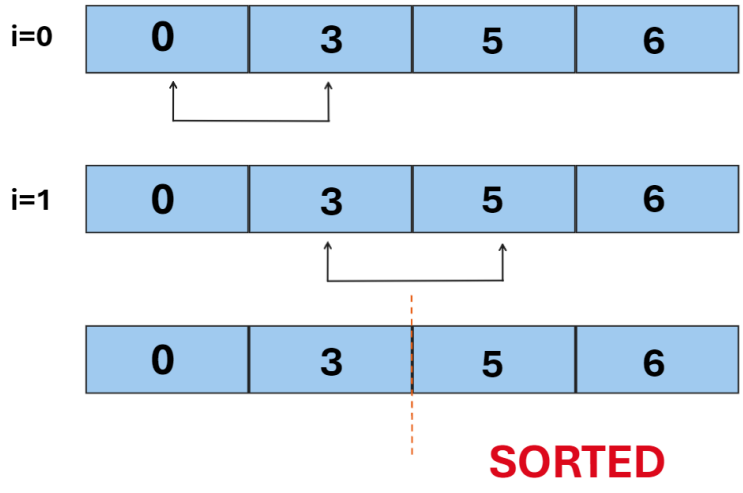
Algoritmul Bubble Sort

În acest algoritm,

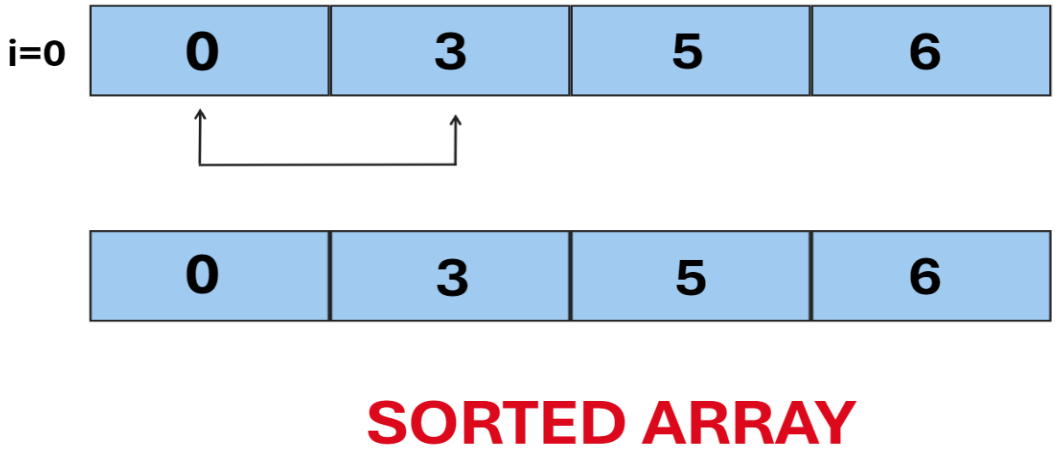
- **traversați din stânga și comparați elementele adiacente, iar cel mai mare este plasat în partea dreaptă.**



- **în acest fel, cel mai mare element este deplasat la capătul din dreapta la fiecare parcurgere**



- **acest proces este apoi continuat pentru a găsi al doilea element ca mărime și va fi plasat la dreapta și așa mai departe până când datele sunt sortate**



- **Numărparcurgeri : n-1** **Număr comparații: n*(n-1)/2**

3.Design si analiza

3.1 Algoritm Bubble Sort

Algoritmul Bubble Sort este un algoritm de sortare simplu, care funcționează prin parcurgerea repetată a listei și **schimbarea pozițiilor adiacente** dacă acestea nu sunt în ordine corectă. Am ales acest algoritm pentru studiul nostru deoarece este ușor de implementat și oferă o perspectivă clară asupra modului în care limbajele de programare alese (C,C++,Java) gestionează operațiile de bază, precum schimbul de elemente într-un vector.

Complexitatea O(n^2) în cel mai rău caz și impactul său evident asupra timpului de execuție fac ca acest algoritm să fie relevant pentru evaluarea performanței limbajelor.

3.2 Alocarea și gestionarea memoriei

Limbajul C

În limbajul C, am folosit alocarea statică și dinamică a memoriei(alocarea statică pentru vectorul static și alocarea dinamică pentru vectorul dinamic)

Astfel, alocarea statică se realizează la compilare, iar memoria este rezervată în timpul definirii variabilelor. De exemplu, în codul nostru, avem un vector static definit în funcția main(), in timp ce alocarea dinamică se realizează cu ajutorul funcțiilor malloc() și free(). Acestea permit programatorului să aloce și să elibereze memorie în timpul execuției, oferind o flexibilitate mai mare, dar necesită gestionarea manuală a resurselor.

```
int staticArray[10000];  // static

//dinamic
int* dynamicArray = (int*)malloc(arraySize * sizeof(int));
```

```
free(dynamicArray);
```

Limbajul C++

În limbajul C++, alocarea statică este similară cu cea din C, dar cu adăugarea de caracteristici precum constructori și destructori pentru obiecte statice. Alocarea dinamică este gestionată prin operatorii new și delete. În exemplul nostru, am folosit new pentru a aloca un vector dinamic.

```
const int n = 10000;
int static_array[n];  // static

int* dynamic_array = new int[n]; // dinamic
```

În aceeași manieră ca în limbajul C, gestionarea manuală a memoriei în C++ poate aduce un control mai mare, dar trebuie realizată cu atenție pentru a evita problemele legate de alocarea și dealocarea incorectă a memoriei.

```
delete[] dynamic_array;
```

Limbajul Java

În Java, gestionarea memoriei este realizată automat prin garbage collector(colectorul de gunoi), iar vectorii sunt tratati ca obiecte. Obiectele sunt alocate în stiva (stack) sau pe heap, iar colectorul de gunoi se ocupă de eliberarea automată a memoriei alocată, eliminând necesitatea gestionării manuale a resurselor ca-n cele 2 limbaje precedente.

Alocarea dinamică este implicită pentru obiectele create cu new în Java. Alocarea și dezalocarea automată a memoriei în Java pot contribui la eficiența programului, dar pot afecta timpului de execuție.

```
int n=10000;
int[] staticArray = new int[10000]; // alocare statică

int[] dynamicArray = new int[n]; // alocare dinamică
```

3.3 Accesul la memorie

Limbajul C

În limbajul C, avem acces direct la adresele de memorie ale elementelor din vector. Bubble Sort efectuează citiri și scrieri directe la adrese. Schimbul de elemente implică manipularea valorilor stocate la adresele specifice. Această abordare directă duce la performanțe mai bune în comparație cu Java,

```
C ---- Time taken for static array: 0.393000 seconds
C ---- Time taken for dynamic array: 0.387000 seconds
```

Limbajul C++

La fel ca în C, C++ permite accesul direct la memorie. Cu toate acestea, în implementarea Bubble Sort cu C++, există utilizarea operatorilor new și delete pentru a aloca și dezaloca dinamic memoria. Accesul la memorie este, în esență, similar cu C, cu avantajul adăugat al funcționalităților OOP. Astfel, implică manipularea directă a valorilor stocate în adresele de memorie, oferind un control detaliat.

```
C++ ---- Time taken for static array: 0.409000 seconds
C++ ---- Time taken for dynamic array: 0.401000 seconds
```

Limbajul Java

În Java vectorii sunt obiecte dinamice. Accesul la memorie este gestionat implicit prin intermediul pointerilor de referință la obiecte. În Bubble Sort,operațiile de citire și scriere se realizează prin intermediul referințelor la obiect, iar schimbul de elemente presupune manipularea acestor referințe. Aceasta aduce o penalizare în ceea ce privește performanța, deoarece accesul la memorie este indirect și gestionat de garbage collector(colectorul de gunoi)**.**

```
Java ---- Time taken for static array: 1.269659 seconds
Java ---- Time taken for dynamic array: 1.276577 seconds
```

3.4 Crearea și gestionarea firelor de execuție

Limbajul C

În limbajul C, crearea și gestionarea firelor de execuție(threadurilor) se realizează prin intermediul bibliotecii pthread. În crearea unui nou fir de execuție(thread) se folosește funcția pthread_create,având ca parametri locali atât funcția pe care ne-o dorim să fie executată în noul fir de execuție(thread) cât și orice alți parametri necesari.

Gestionarea firelor de execuție(thread-urilor) se referă la sincronizarea și comunicarea între firele de executie (thread-urilor) și se realizeaza cu ajutorul variabilelor condiționale (pthread_cond_t) și mutex-urilor (pthread_mutex_t). Această metodă oferă un control detaliat,dar o responsabilitate mare a programatorului de gestionare a thread-urilor.

```
C ---- THREAD : Time taken for static array: 0.053000 seconds
C ---- THREAD : Time taken for dynamic array: 0.056000 seconds
```

Limbajul Java

În limbajul Java, crearea și gestionarea firelor de execuție se fac prin intermediul clasei Thread. Java oferă un mod de gestionare mai ușor de utilizat comparativ cu limbajul C sau C++.

Pentru a crea un fir de execuție(thread), se creează o clasă care extinde clasa Thread sau se furnizează o implementare a interfeței Runnable. Firul de execuție (Thread-ul) este pornit cu metoda start(). Java oferă diferite clase și interfețe precum ExecutorService, ThreadPoolExecutor, și Future pentru a facilita gestionarea execuției concurente și comunicarea între firele de execuție

```
Java ---- THREAD : Time taken for static array: 0.274239 seconds
Java ---- THREAD : Time taken for dynamic array: 0.277067 seconds
```

Limbajul C++

În limbajul C++, crearea și gestionarea firelor de execuție (thread-urilor) se realizează prin intermediul bibliotecii <thread>. Pentru a crea un nou fir de execuție (thread), se utilizează clasa std::thread, având ca parametri o funcție pe care dorim să o executăm în noul fir de execuție și orice alți parametri necesari.

Gestionarea firelor de execuție (thread-urilor) în C++ se bazează pe concepte precum mutex-uri și variabile condiționale la fel ca in C. Mutex-urile (clasa std::mutex) sunt utilizate pentru a asigura accesul exclusiv la resurse partajate între firele de execuție, evitând astfel conflictele de sincronizare.

```
C++ ---- THREAD : Time taken for static array: 0.121000 seconds
C++ ---- THREAD : Time taken for dynamic array: 0.097000 seconds
```

3.5 Compararea performanțelor

Alocarea și gestionarea memoriei: Limbajele C și C++ oferă programatorului control detaliat asupra alocării și dezalocării memoriei, atât static cât și dinamic, oferind flexibilitate, dar cu responsabilitatea gestionării manuale a resurselor. În schimb, Java gestionează automat memoria prin garbage collector, eliminând nevoia de gestionare manuală, dar potențial afectând timpul de execuție.

Accesul la memorie: Limbajul C oferă acces direct la adresele de memorie, având astfel performanțe mai bune în operațiile precum cele din Bubble Sort. C++ păstrează accesul direct, dar adaugă funcționalități OOP. Java, cu gestionarea referințelor la obiecte, suferă o penalizare în performanță în comparație cu C și C++.

Crearea și gestionarea firelor de execuție: C oferă un control detaliat și responsabilitate mare asupra firelor de execuție prin biblioteca pthread, în timp ce C++ utilizează clasa std::thread cu concepte similare. Java oferă o gestionare mai ușoară a firelor de execuție prin clase precum Thread și interfețe precum ExecutorService.

4 Concluzii

Prin intermediul acestui proiect, am analizat măsurarea timpului de execuție a proceselor în limbajele de programare C, C++, și Java. Am regăsit diferite aspecte ale performanței, inclusiv alocarea și gestionarea memoriei, accesul la memorie, crearea și gestionarea firelor de execuție.

În ceea ce privește alocarea și gestionarea memoriei, am observat că limbajele C și C++ oferă programatorului control detaliat asupra resurselor, permițând alocarea și dezalocarea atât statică, cât și dinamică a memoriei. În schimb, Java oferă gestionare automată a memoriei prin garbage collector, eliminând necesitatea intervenției manuale, dar aducând cu sine potențiale impacturi asupra timpului de execuție.

Accesul la memorie a evidențiat diferențe semnificative între limbaje. C și C++ oferă acces direct la adresele de memorie, contribuind la performanță, în timp ce Java, prin gestionarea referințelor la obiecte, suferă o penalizare în ceea ce privește eficiența în operațiile de citire și scriere.

În ceea ce privește crearea și gestionarea firelor de execuție, am constatat că C oferă un control detaliat și responsabilitate mare prin biblioteca pthread, în timp ce C++ și Java furnizează soluții mai ușor de utilizat, cu clase precum std::thread și Thread, respectiv.

Compararea performanțelor ne-a permis să evidențiem avantajele și dezavantajele fiecărui limbaj în contextul măsurării timpului de execuție. C și C++ se remarcă pentru controlul lor asupra resurselor hardware, în timp ce Java aduce beneficii în gestionarea automată a memoriei și portabilitate.

În concluzie, acest proiect a oferit o perspectivă detaliată asupra modului în care limbajele de programare C, C++, și Java gestionează timpul de execuție al proceselor. Analiza a contribuit la înțelegerea aspectelor practice și teoretice care pot influența performanța unui program în funcție de limbajul de programare ales.