PROCESARE PARALELĂ ȘI DISTRIBUITĂ

PROBLEMA RUCSACULUI

Student: Boșnegeanu Adrian-Gabriel

Cadru didactic îndrumător: Lect. dr. mat. Daniel CIUIU

1. A drawing of a bag and boxes

   Description automatically generatedIntroducere:

Problema rucsacului (Knapsack Problem) este o provocare clasică de optimizare combinatorică. Având o colecție de elemente, fiecare caracterizat de o greutate și o valoare, scopul este de a determina selecția optimă a acestora astfel încât greutatea totală să nu depășească o limită prestabilită, iar valoarea totală să fie maximă.

Această problemă își are originea la sfârșitul secolului al XIX-lea, primele lucrări datând din 1897. Denumirea "problema rucsacului" provine din analogia cu selecția optimă a articolelor de către un călător care trebuie să își împacheteze bagajul fără a depăși capacitatea acestuia. Matematicianul Tobias Dantzig este creditat pentru popularizarea termenului.

Cea mai comună variantă este problema rucsacului 0-1, care impune restricția ca fiecare obiect să fie inclus în rucsac cel mult o singură dată. Având o mulțime de n elemente numerotate de la 1 la n, fiecare definit printr-o greutate wi​ și o valoare vi​, împreună cu o capacitate maximă de greutate W, scopul este de a maximiza valoarea totală a obiectelor selectate fără a depăși limita de greutate.

Această problemă este fundamentală în teoria optimizării și are aplicații practice în diverse domenii, cum ar fi logistică, finanțe și informatică.

1. Algoritmul secvențial

A screenshot of a computer code

Description automatically generated

Acest cod implementează soluția pentru problema rucsacului (Knapsack Problem) utilizând programarea dinamică. Scopul este de a determina valoarea maximă care poate fi obținută dintr-un set de obiecte, având în vedere o capacitate limitată a rucsacului.

Explicație pas cu pas:

1. **Funcția rucsac()**:
   * Parametrii:
     + cost[]: un tablou care reprezintă greutățile obiectelor.
     + valoare[]: un tablou care reprezintă valorile obiectelor.
     + n: numărul de obiecte.
     + capacitate: capacitatea maximă a rucsacului.
   * Funcția utilizează o matrice de programare dinamică dp pentru a stoca soluțiile intermediare.
     + dp[i][w] reprezintă valoarea maximă pe care o putem obține cu primele i obiecte și cu o capacitate w a rucsacului.
     + Se initializează matricea cu zerouri, iar apoi se completează pas cu pas.
2. **Algoritmul principal**:
   * **Dublă buclă for**:
     + Prima buclă parcurge fiecare obiect (i de la 1 la n).
     + A doua buclă parcurge capacitățile rucsacului (w de la 1 la capacitate).
   * Dacă greutatea obiectului curent cost[i-1] este mai mică sau egală cu capacitatea curentă w, se poate alege dacă să includem obiectul în rucsac sau nu:
     + Dacă îl includem, valoarea maximă devine dp[i-1][w-cost[i-1]] + valoare[i-1].
     + Dacă nu îl includem, valoarea rămâne dp[i-1][w].
   * Dacă greutatea obiectului este mai mare decât capacitatea curentă w, nu putem adăuga obiectul în rucsac, iar valoarea rămâne aceeași ca în cazul anterior.
3. **Rezultatul**:
   * La final, valoarea maximă care poate fi obținută cu toate obiectele și capacitatea rucsacului este stocată în dp[n][capacitate].
4. **Funcția main()**:
   * Se definesc greutățile și valorile obiectelor, precum și capacitatea rucsacului.
   * Se calculează și se afișează valoarea maximă posibilă folosind funcția rucsac().

A screenshot of a computer program

Description automatically generatedIII. **Algoritmul paralel**

**A computer code with text

Description automatically generated**Acest cod implementează o variantă paralelă a problemei rucsacului (Knapsack Problem) folosind mai multe fire de execuție (multithreading). Algoritmul împarte obiectele în intervale și procesează fiecare interval într-un fir separat pentru a optimiza performanța. Iată explicația detaliată:

**1. Funcția procesare\_interval():**

Aceasta este funcția care procesează un interval de obiecte pentru a găsi valoarea maximă posibilă într-un subset de obiecte din acel interval.

* **Parametrii**:
  + cost: un vector care conține greutățile obiectelor.
  + valoare: un vector care conține valorile obiectelor.
  + start și end: indică intervalul de obiecte care vor fi procesate în acest fir de execuție.
  + capacitate: capacitatea maximă a rucsacului.
  + valoare\_max\_local: va conține valoarea maximă găsită pentru intervalul respectiv de obiecte.
* **Cum funcționează**:
  + Se generează toate subseturile posibile de obiecte pentru intervalul specificat.
  + Fiecare subset este reprezentat de un număr în binar (subsetul este determinat prin bitii care sunt setați la 1).
  + Pentru fiecare subset, se calculează suma greutăților și valorilor obiectelor incluse. Dacă greutatea totală este mai mică sau egală cu capacitatea rucsacului, valoarea totală este comparată cu valoarea maximă locală găsită până atunci.
  + valoare\_max\_local este actualizată cu valoarea maximă a subseturilor valabile (care nu depășesc capacitatea).

**2. Funcția main():**

Această funcție gestionează logica principală a programului, inclusiv crearea firelor de execuție și coordonarea acestora.

* **Inițializare**:
  + Se definesc greutățile obiectelor (cost) și valorile acestora (valoare).
  + Se află numărul de obiecte (n).
  + Se află numărul de fire de execuție disponibile pe mașina curentă, folosind thread::hardware\_concurrency(). Acesta va determina câte fire de execuție vor fi create.
* **Împărțirea obiectelor între fire**:
  + Se calculează dimensiunea fiecărui interval de obiecte care va fi procesat de fiecare fir de execuție. Dimensiunea este determinată astfel încât toate obiectele să fie distribuite echitabil între firele de execuție.
  + Fiecare fir va procesa un interval din obiecte (de la start la end).
* **Crearea și lansarea firelor**:
  + Pentru fiecare fir de execuție, se lansează un fir care va executa funcția procesare\_interval() pentru intervalul de obiecte corespunzător.
  + Utilizăm cref pentru a trece referințe constante pentru vectorii cost și valoare, iar ref pentru a putea modifica variabila valoare\_max\_local în fiecare fir.
* **Așteptarea finalizării firelor**:
  + După lansarea firelor, programul așteaptă ca fiecare fir să își termine execuția folosind join(). Acest lucru se face pentru fiecare fir activ.
* **Calcularea valorii maxime globale**:
  + După ce toate firele au terminat, valoarea maximă globală este calculată iterând prin valorile maxime locale returnate de fiecare fir. Se selectează valoarea maximă dintre ele.
* **Afisarea rezultatului**:
  + La final, valoarea maximă care poate fi obținută este afișată.

**3. Ce face acest algoritm?**

* **Paralelizarea**: Codul împarte obiectele în mai multe intervale și le procesează în paralel folosind fire de execuție. Aceasta este o metodă de a accelera procesul, deoarece fiecare fir de execuție poate calcula valoarea maximă pentru un subset de obiecte într-un interval.
* **Subseturi de obiecte**: Algoritmul explorează toate subseturile posibile de obiecte din fiecare interval, iar pentru fiecare subset calculează valoarea totală și greutatea. Dacă greutatea nu depășește capacitatea rucsacului, valoarea totală este comparată și actualizată.
  1. **Rezultate**

Algoritmul secvențial

* Complexitate:
* Timp de execuție(10 obiecte, 30 capacitatea): **520 microsecunde**

Algoritmul paralel

* Complexitate: **,** unde p este numărul de fire de execuție
* Timp de execuție(10 obiecte, 30 capacitatea, p=5): **220 microsecunde**