Procesare Paralelă

Capitolul 3: Algoritmi paraleli—generalităţi

Bogdan Dumitrescu

Facultatea de Automatică şi Calculatoare

Universitatea Politehnica Bucureşti

Cuprins

- Performanţele unui algoritm paralel
- Legea lui Amdahl
- Tehnici generale de paralelizare
- Grafuri de precedenţă
- Planificare

Timpul de execuţie

- Timpul de execuţie $T(n, p)$ deprinde (cel puţin) de
  - n, dimensiunea problemei, reprezentând mărimea datelor de intrare, de exemplu dimensiunea unor matrice sau vectori
  - p, numărul de procesoare
- Timp de execuţie este intervalul dintre momentul în care *primul* procesor începe lucrul şi cel în care *ultimul* procesor termină lucrul
- Măsurare corectă: algoritmul începe şi se termină cu câte o sincronizare, timpii iniţial şi final fiind măsuraţi imediat după acestea

Accelerarea

- Intuitiv, un algoritm paralel este bun dacă rulat pe p procesoare este mai rapid de p ori, sau de aproape p ori, decât un algoritm secvenţial care rezolvă aceeaşi problemă
- Accelerarea (speed-up, creştere de viteză) algoritmului paralel X este

$$S_X(n,p) = T_S(n)/T_X(n,p)$$

- $T_S(n)$ este timpul de execuţie al celui mai rapid algoritm secvenţial care rezolvă problema
- $T_X(n,p)$ este timpul de execuţie al algoritmului paralel
- In general, $1 \leq S(n,p) \leq p$
- Sunt posibile valori mai mici ca 1 ? (când algoritmul paralel e mai lent decât cel secvenţial)

Eficienţa

- Parametrul cel mai folosit în aprecierea unui algoritm paralel X este eficienţa

$$\varepsilon_X(n,p) = \frac{S_X(n,p)}{p} = \frac{T_S(n)}{pT_X(n,p)}$$

- Eficienţa are (aproape) întotdeauna o valoare subunitară
- Un algoritm este cu atât mai valoros cu cât eficienţa sa este mai aproape de 1
- Avantaj faţă de accelerare: nu trebuie cunoscut p pentru a aprecia daca algoritmul se comportă bine
- In general, eficienţa creşte o dată cu n şi scade o dată cu p
- Un algoritm paralel are eficienţa *asimptotic egală cu 1*, dacă $\lim_{n\to\infty} \varepsilon(n,p) = 1$, pentru orice p constant

Overhead

- Overhead (supraîncărcare): totalitatea factorilor care micşorează eficienţa de la valoarea ideală 1
- Tipuri de overhead:
  - datorat *comunicaţiei*, care nu există în cazul secvenţial
  - *algoritmic*: datorat părţilor intrinsec secvenţiale ale algoritmului—părţi care nu se pot paraleliza, sau datorat unor operaţii suplimentare efectuate în algoritmul paralel faţă de cel mai rapid algoritm secvenţial
  - datorat *încărcării inegale* a procesoarelor; ideal, calculele sunt repartizate uniform procesoarelor

Legea lui Amdahl

- Analizăm comportarea generală a unui algoritm în funcţie de numărul de procesoare p
- Ipoteză simplificatoare: algoritmul are o parte pur secvenţială, care necesită un timp W_1, şi o alta perfect paralelă, indiferent de numărul de procesoare, care necesită un timp (total) W_P
- Timpul de execuţie pe p procesoare este

$$T(p) = T_1 + T_P = W_1 + W_P/p$$

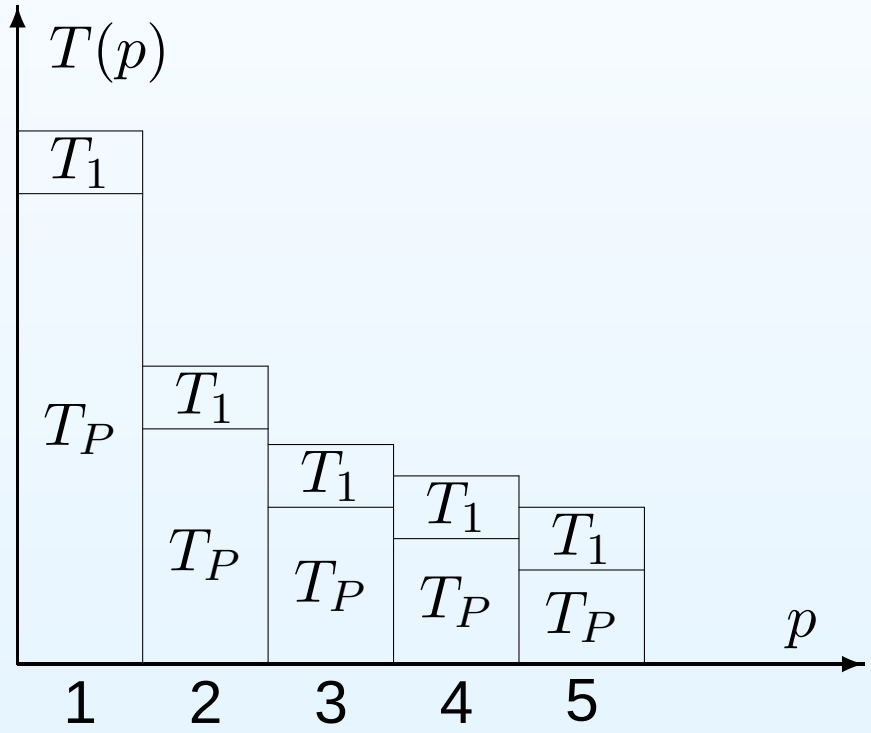
- "Legea lui Amdahl": accelerarea are expresia

$$S(p) = \frac{W_1 + W_P}{W_1 + W_P/p} < 1 + \frac{W_P}{W_1}$$

- Deci, oricât de mare ar fi p, accelerarea are o limită inerentă

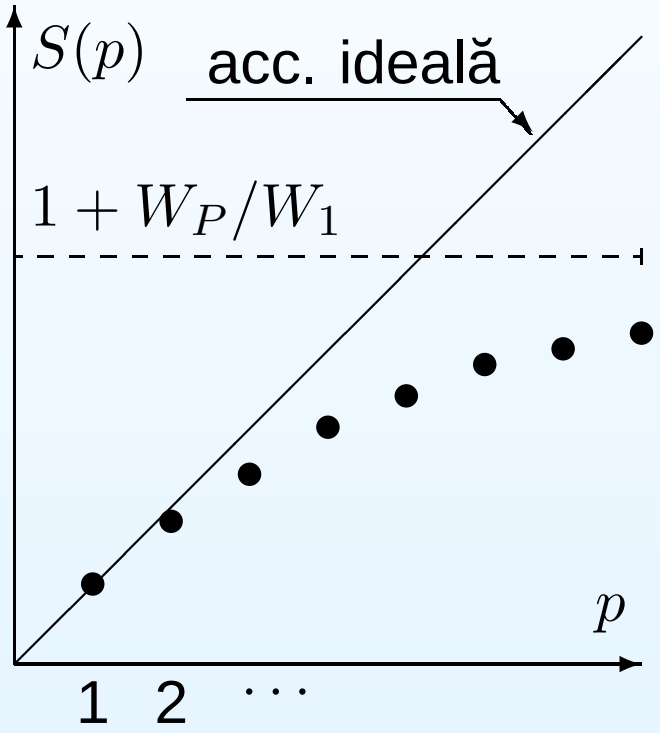
Ilustrarea legii lui Amdahl

- Timpul de execuţie evoluează astfel în funcţie de numărul de procesoare



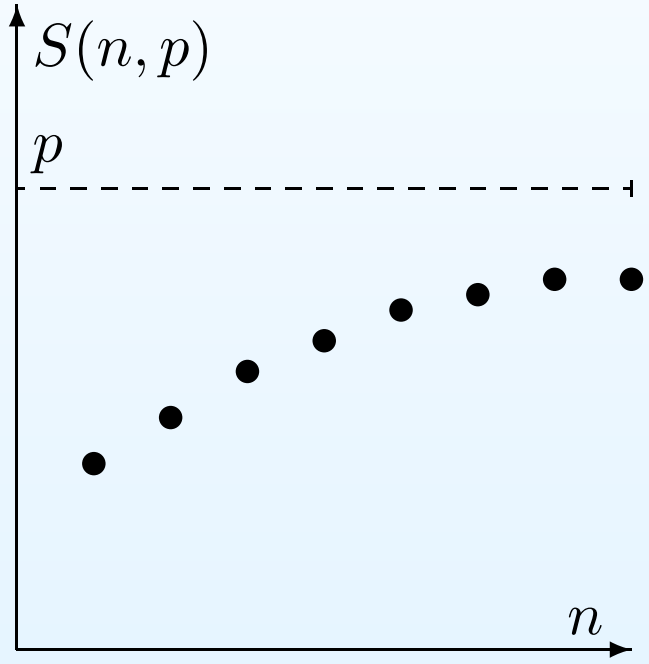
Ilustrarea legii lui Amdahl

- Evoluţia tipică a acceleărării în funcţie de numărul de procesoare



Efectul dimensiunii problemei

- De obicei, când dimensiunea problemei creşte, parte secvenţială a algoritmului scade faţă de partea paralelă
- Tipic, accelerarea evoluează astfel



Superliniaritate

- Superliniaritate: accelerare mai mare ca p sau eficienţă mai mare ca 1
- Cauza tipică: accesul la memorie
- Pe un calculator secvenţial, un program poate consuma multă memorie şi accesul la memoria principală poate fi frecvent
- Pe un calculator paralel, datele sunt distribuite, deci fiecare procesor are mai puţine; memoria rapidă poate fi utilizată mai eficient
- In general, superliniaritatea arată o slabă implementare sau execuţie secvenţială, nu neapărat o implementare paralelă foarte bună

Mod general de paralelizare

- In general, pentru a trece de la un algoritm secvenţial la unul paralel, parcurgem următoarele etape
  - gruparea în taskuri a operaţiilor ce trebuie executate
  - identificarea ordinii în care se pot executa taskurile, precum şi a taskurilor ce se pot executa în paralel; construcţie ajutătoare: graful de precedenţă a taskurilor
  - planificarea (scheduling) execuţiei taskurilor, adică stabilirea unui mod de a descrie procesorul pe care se execută fiecare task
- Aceste etape se parcurg mai mult sau mai puţin explicit, în funcţie de complexitatea problemei şi experienţa programatorului

Task-uri (thread-uri)

- Un task este o unitate indivizibilă de operaţii elementare, executate secvenţial
- Gruparea operaţiilor în task-uri se face în general euristic
- Părţile eminamente secvenţiale sunt grupate în acelaşi task
- Operaţii ce se pot executa în paralel se introduc în task-uri diferite
- Granularitatea unui algoritm
  - fină: task-urile au puţine operaţii (bine pe calculatoare SIMD sau cu comunicaţie rapidă)
  - mare: task-urile au multe operaţii (pe calculatoare cu comunicaţie lentă faţă de viteza de comunicaţie: majoritatea calculatoarelor MIMD)

Graf de precedenţă

- Dacă un task T_j are nevoie de datele produse de un alt task T_i pentru a se putea executa, spunem că T_i precede T_j şi notăm $T_i \prec T_j$
- Graf de precedenţă: nodurile sunt task-uri iar arcele relaţiile de precedenţă dintre ele
- Relaţia de precedenţă este tranzitivă
- Dacă $T_i \prec T_j \prec T_k$, punem în graf doar arcele pentru $T_i \prec T_j$ şi $T_j \prec T_k$
- Un graf de precedenţă este orientat şi aciclic (DAG: Directed Acyclic Graph)

DAG—proprietăţi

- Un DAG seamănă cu un arbore, dar:
  - are mai multe "rădăcini"
  - un nod poate avea mai mulţi taţi
- Pentru a simplifica definiţiile de mai jos, presupunem task-urile au durate de execuţie egale
- *Nivel*: nodurile terminale (care nu au nici un succesor) au nivel 0; nivelul $l(v)$ al unui nod v este $l(v) = \max_i l(v_i) + 1$, unde v_i sunt succesorii imediaţi (fiii) ai lui v
- *Înălţimea* (adâncimea): lungimea celei mai lungi căi
- *Lăţimea*: $\max |L_k|$, unde L_k este mulţimea nodurilor având nivelul k, iar $|L_k|$ este cardinalul acestei mulţimi

Complexitatea şi graful de precedenţă

- Cu ajutorul grafului de precedenţă asociat unei partiţionări în taskuri a operaţiilor dintr-un algoritm se poate determina paralelismul intrinsec al partiţionării
- Ignorăm timpul de comunicaţie între task-uri
- Presupunem că numărul de procesoare este suficient de mare
- Inălţimea grafului este timpul cel mai scurt timp în care se poate executa algoritmul
- Lăţimea grafului dă informaţii despre numărul necesar de procesoare pentru a atinge timpul cel mai scurt de execuţie
- Task-urile de pe acelaşi nivel se pot executa în paralel
- Numărul de procesoare necesar pentru obţinerea timpului minim de execuţie este egal cu lăţimea grafului

Planificare

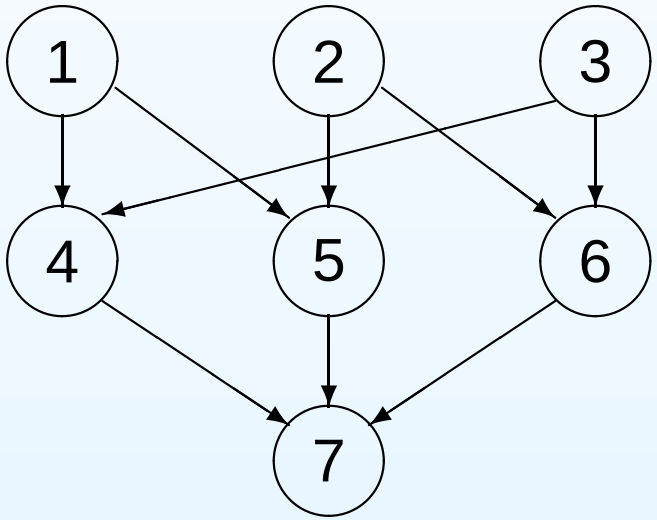
- Planificare (scheduling): stabilirea procesorului pe care se execută un task, adică o funcţie $s(T_i) = P_k$
- Planificarea poate fi
  - *statică*, în care funcţia s este cunoscută înaintea începerii execuţiei algoritmului
  - *dinamică*, în care funcţia s se construieşte pe măsură ce algoritmul se execută
- Planificarea dinamică este mai flexibilă, dar consuma timp suplimentar pentru calculul funcţiei s şi pentru informarea procesorului, eventual încărcarea task-ului
- Planificarea statică este mai simplă şi, în multe probleme de calcul ştiinţific, este mai eficientă

Planificare după listă

- Problema planificării se poate rezolva prin construirea unei liste cuprinzând toate task-urile; apoi se alocă taskurile, în ordinea din listă, procesoarelor libere, cu condiţia de a nu se planifica simultan decât taskuri independente
- Lista se poate crea folosind nivelele grafului de precedenţă
- Ultimele în listă sunt nodurile de pe nivelul 0, imediat înaintea lor cele de pe nivelul 1 etc.
- Pentru a ordona nodurile de pe acelaşi nivel se foloseşte, de exemplu, criteriul numărului de succesori direcţi: primul în listă este pus taskul cu cei mai mulţi succesori direcţi
- Acest algoritm este doar euristic, nu garantează planificarea optimă

Exemplu—graf de precedenţă

- Un graf cu 7 task-uri



Exemplu—planificări

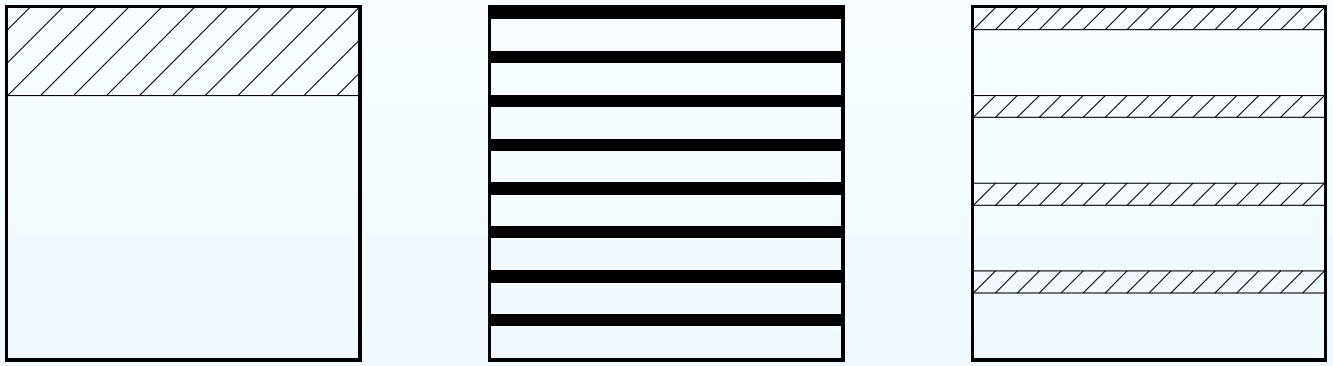
- Trei planificări pe 2 procesoare

a			**b**			**c**	
P_0	P_1		P_0	P_1		P_0	P_1
1	2		1	3		1	2
3			2	4		3	5
4	5		5	6		4	6
6			7			7	
7							

Efectul repartizării datelor

- De multe ori planificarea începe de la o repartizare a datelor
  - echilibrată: fiecare procesor primeşte cam aceeaşi cantitate din datele iniţiale
  - fără repetiţii: o anume dată este alocată iniţial unui singur procesor
- De exemplu, putem repartiza liniile unei matrice
  - *bloc linii*: primele m linii sunt locale procesorului P_0, următoarele m lui P_1, etc. Procesorul P_i posedă liniile cu numerele de la im la $(i+1)m-1$
  - *ciclic pe linii*: procesorului P_i îi aparţin liniile cu numerele $jp+i$, cu $j = 0, 1, \ldots, m-1$; sau, altfel spus, liniile l pentru care $l \bmod p = i$
  - *bloc ciclic pe linii*: combinaţia celor două moduri precedente

Ilustrarea repartizărilor



- Aceste moduri de repartizare sunt naturale pe inel
- Tot pe inel, repartizările se pot face pe coloane
- Generalizare pe tor: se repartizează similar atât pe linii cât şi pe coloane

Algoritmi simpli

- Algoritmii vor fi prezentaţi la curs:
  - suma
  - suma globală
  - suma prefixelor
  - produsul matrice-vector