

Cuprins

Stadiul actual

Despre curs

De ce paralelism? De ce distribuție?

Bibliografie

Algoritmi paraleli și distribuți

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași



○○○

Cuprins

Stadiul actual

Despre curs

Ce se dorește?

Sistemul de evaluare

Conținut

De ce paralelism? De ce distribuție?

Bibliografie

Prezentul este distribuit!

- Internetul
- Web-ul
- Intranetul
- Rețelele ATM ale băncilor
- Rețelele de telefonie mobilă
- Internet of things
-

Stadiul actual

Viitorul este paralel!

- Multicore - o nouă eră
 - Oportunitate de a face paralelismul mai usor pentru oricine.
 - Mediul academic este pregătit.
 - Practicienii trebuie să facă un efort de adaptare.
- Sistemele multiprocesor - în revenire
 - și în acest sens mediul academic este pregătit.
 - Mediul de afaceri trebuie să le folosească mai mult.

Ce se doreste?

Obiectivele cursului

- Formarea unei viziuni de ansamblu asupra sistemelor de calcul paralel/distribuit ;
- Inițiere în algoritmica și programarea paralelă și distribuită.

Rezultatele învățării

- Cunoașterea problemelor generice care pot fi rezolvate prin algoritmi paraleli și distribuți;
- Abilitatea de a aplica algoritmul paralel și/sau distribuit adecvat unei probleme;
- Capacitatea de a proiecta, implementa și testa algoritmi paraleli și distribuți;
- Abilități de programare paralelă și distribuită.

Sistemul de evaluare

Examen: nota minimă 5, Ponderea în nota finală 60%

- Evaluarea finală (nota minimă 5)
 - Ponderea în nota examen: 80%
 - Test de cunoștințe
 - Criterii de evaluare: calitatea, corectitudinea, acuratețea cunoștințelor teoretice și practice acumulate
- Teme de casă
 - Ponderea în nota examen: 20%
 - Criterii de evaluare: rezolvarea temelor propuse și calitatea soluțiilor.

Laborator: nota minimă 5), ponderea în nota finală 40%

- Criterii de evaluare: rezolvarea temelor propuse și calitatea soluțiilor.

Prelegeri

- Arhitecturi de calcul paralel/distribuit
- Modele de calcul paralel/distribuit
- Calcul de înaltă performanță
- Comunicarea în sistemele de calcul paralel/distribuit
- Calcul paralel
 - Algoritmi paraleli fundamentali
 - Sortare paralelă
 - Algoritmi paraleli pentru calculul matricial
 - Algoritmi paraleli pentru sisteme de ecuații liniare
 - Transformata Fourier
- Calcul distribuit
 - Alegerea liderului
 - Excluderea mutuală

Lucrări de laborator

- Introducere în OpenMP
- Introducere în MPI
- Programe MPI de simulare a scenarilor de comunicare unu la toți și toți la toți pe hipercub
- Programe MPI de simulare a scenarilor de comunicare unu la toți și toți la unu pe un arbore oarecare
- Programe OpenMP și MPI pentru pentru comprimare (reducere) și calcul prefixe (scan)
- Temă de casă: MapReduce (program MPI, cu predare în săptămâna 14)
- Sortare paralelă (Muller-Preparata - program OpenMP, Par-impar – programe OpenMP, MPI)
- Sortare paralelă (sortare bitonică pe hipercub, program MPI)
- Calcul matriceal (transpusa unei matrice și înmulțirea a două matrice pătratice cu algoritmul Cannon, program MPI + demo OpenCL/CUDA)
- Sisteme de ecuații liniare (program OpenMP + demo OpenCL/CUDA)
- Alegerea liderului pe un inel (program MPI pentru algoritmul LCR - LeLann, Chang și Roberts)
- Alegerea liderului pe o topologie de graf oarecate: program MPI pentru algoritmul FloodMax (2 ore)

Profesori

Curs

- Mitică Craus

Laborator

- Adrian Alexandrescu
- Alexandru Archip
- Silviu Pavăl

De ce paralelism? De ce distribuție?

- Aplicațiile o cer:

- predicția vremii, cutremurelor, tornadelor, uraganelor;
- simularea pe calculator a proceselor fizice, chimice, biologice;
- industria aeronațională (dinamica fluidelor) și a automobilelor (simularea coliziunilor);
- ingineria materialelor;
- nano tehnologiile;
- modelarea organelor;
- descoperirea de medicamente noi;
- grafica, video;
- robotica
- bazele de date, data mining;
- inteligența artificială.

- Tendințele tehnologice

- integrarea masivă;
- interconectarea.

- Tendințele arhitecturale

- arhitecturi multicore și multiprocesor;
- clusterele, *grid*-urile, norii de calcul (*cloud*).

Bibliografie

- V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003
- H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, John Wiley & Sons, Inc., 2004
- K. Berman, J. Paul, Algorithms: Sequential, Parallel, and Distributed, Thomson Learning, Inc., 2005
- T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Addison-Wesley, 2005

- 80

- 300

- 60

Algoritmi paraleli și distribuiți

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

- ○

Cuprins

Introducere

Sisteme abstracte de calcul paralel/distribuit

Arhitecturi paralele/distribuite

Modele de comunicare

Taxonomie

Modelul de comunicare bazat pe memorie partajată

Modelul de comunicare bazat pe retele de interconectare

Topologii de comunicare

Clasificare

Topologii regulate

Graful complet

Lanțul, inelul și steaua

Topologii arborescente

Plasele

Hipercubul

Evaluarea topologiilor regulate

Toplogii neregulate

Graful oarecare

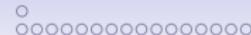
Arborele oarecare

Bibliografie



Sisteme abstracte de calcul paralel/distribuit

- Un sistem abstract de calcul paralel/distribuit (SACP/D) este un ansamblu de unități de procesare a datelor, care comunică între ele.
 - Unitatea de procesare este un element atomic al sistemului, cu o structură internă invizibilă.
 - Se remarcă în această definiție două componente de bază ale unui SACP/D:
 - componenta computațională;
 - componenta communicatională.



Arhitecturi paralele/distribuite

- O arhitectură paralelă/distribuită este o implementare a unui SACP/D. Aceasta înseamnă:
 - maparea subsistemului computațional pe o mulțime de mașini cu o funcționalitate bine definită și
 - asignarea relațiilor de comunicare la un model de comunicare.
- Modelul de comunicare se referă la mediul de comunicare și modalitatea în care unitățile de procesare comunică între ele.
- O arhitectură paralelă/distribuită execută algoritmi paraleli/distribuiți.
- Definiția unui algoritm paralel/distribuit poate fi derivată din definiția unui algoritm secvențial, prin înlocuirea noțiunii de pas secvențial cu cea de pas paralel.
- Pasul paralel poate fi un pas de sincronizare (pas paralel impropriu) sau un ansamblu de pași secvențiali, execuți în paralel în unitățile de procesare (pas paralel propriu).
- *Observație:* Paralelismul implică distribuție dar distribuția nu implică paralelism. Aceasta ar putea fi punctul de plecare în diferențierea algoritmilor paraleli de cei distribuiți.



Modele de comunicare Taxonomie

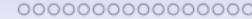
- Determinant în definirea unui model de comunicare este mediul de comunicare.
- Modelele de comunicare universal acceptate sunt cele bazate pe memoria partajată și rețelele de interconectare.

Modelul de comunicare bazat pe memorie partajată

- Are la bază principiul comunicării prin variabile de memorie.
- Topologia de comunicare are potențialul unui graf complet, deși acesta nu este decât o stea specială în care memoria comună este nodul central.
- Oricare două unități de procesare pot comunica doar prin intermediul memoriei comune (centrul stelei). Aceasta reprezintă o vulnerabilitate.

Modelul de comunicare bazat pe rețele de interconectare

- Oricare două unități de procesare comunică între ele prin canalul de comunicații care le leagă sau prin intermediul unei secvențe de canale care are la capete cele două unități de procesare.
- Un astfel de model poate fi definit ca fiind format din:
 - un graf $G = (V, E)$, care reprezintă topologia de comunicare și ultimele de capacitate de comunicare.



Modelul de comunicare bazat pe memorie partajată

- a. memorie locală privată și memorie externă partajată;
- b. memorie locală privată+partajată și memorie externă partajată;
- c. memorie locală privată+partajată.

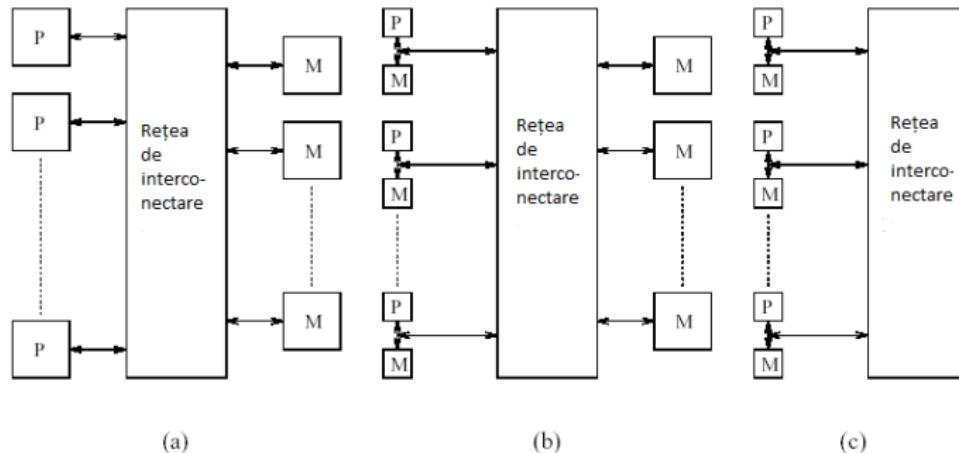


Figura 1: Modelul de comunicare bazat pe memorie partajată

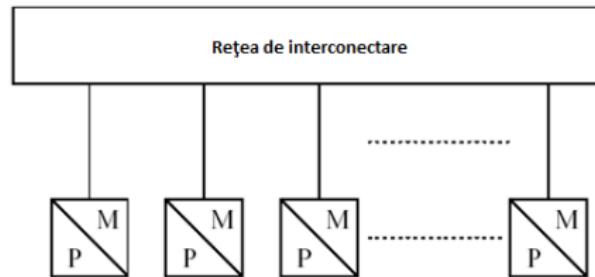


Figura 2: Modelul de comunicare bazat pe rețele de interconectare



Topologii de comunicare

Clasificare

- topologii regulate:
 - graful complet;
 - lanțul;
 - inelul;
 - steaua;
 - arborele binar complet;
 - arborele gras;
 - fluturele;
 - banyan
 - plasa;
 - plasa de arbori;
 - hipercubul;
 - amestecul prefect.
- topologii neregulate:
 - graful oarecare;
 - arborele.

- 88

-

- 10

Graful complet

- Rețelele peer-to-peer.

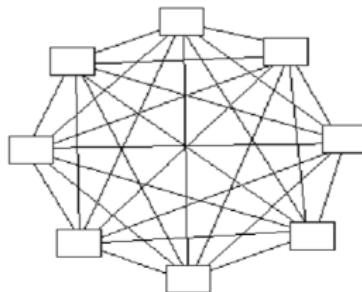


Figura 3: Graful complet



Lanțul

- Mulțimea nodurilor V este organizată sub forma unui tablou unidimensional de dimensiune n , prin intermediul unei funcții de indexare $I : V \rightarrow L = \{0, 1, \dots, n - 1\}$.
- Dacă se notează cu $V_i = I^{-1}(i)$, nodul V_i este conectat cu nodurile V_{i-1} și V_{i+1} . Fac excepție nodurile $V_0 = I^{-1}(0)$ și $V_{n-1} = I^{-1}(n-1)$. Nodul V_0 este conectat doar cu nodul V_1 . Nodul V_{n-1} este conectat doar cu nodul V_{n-2} .



Figura 4: Lanțul



Inelul

- Mulțimea nodurilor V este indexată prin intermediul unei funcții $I : V \rightarrow L = \{0, 1, \dots, n - 1\}$.
- Nodul $V_i = I^{-1}(i)$ are vecini nodurile $V_{i \ominus 1}$ și $V_{i \oplus 1}$, unde \ominus este scăderea modulo n iar \oplus este adunarea modulo n .

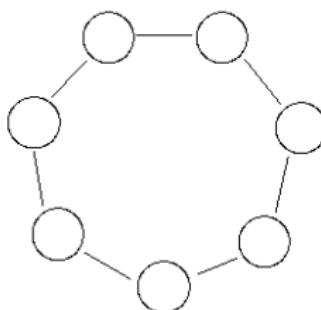


Figura 5: Lanțul



Steaua

- Mulțimea nodurilor, V , este indexată prin intermediul unei funcții $I : V \rightarrow L = \{0, 1, \dots, n - 1\}$.
- Există un nodul special, V_0 , la care sunt conectate toate nodurile.
- Comunicarea între noduri se realizează prin intermediul nodului V_0 .
- Steaua este topologia specifică modelului de comunicare prin intermediul memoriei comune.

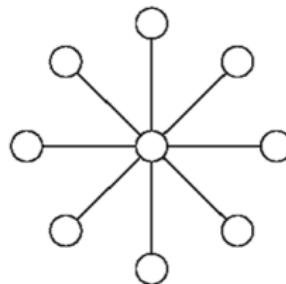


Figura 6: Steaua



Arborele binar complet

- Mulțimea nodurilor V este indexată prin intermediul unei funcții $I : V \rightarrow L = \{0, 1, \dots, n - 1\}$.
- Există un nodul special, V_0 , numit radăcină. Radăcina nu are părinte și are doi fii, V_1 și V_2 .
- Fiecare nod interior are un părinte și doi fii. Părintele unui nod interior $V_i = I^{-1}(i)$ este $V_{\lfloor (i-1)/2 \rfloor}$. Fiiii sunt V_{2i+1} și V_{2i+2} .
- Frunzele au un părinte, nu au fii și sunt situate pe același nivel. Părintele unei frunze $V_i = I^{-1}(i)$ este $V_{\lfloor (i-1)/2 \rfloor}$.
- Între oricare două noduri există un singur drum. Numărul conexiunilor este $n - 1$.

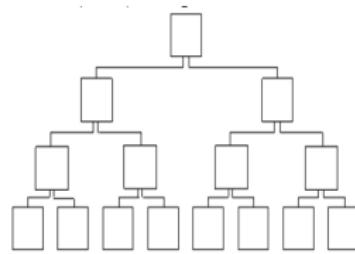


Figura 7: Arborele binar complet

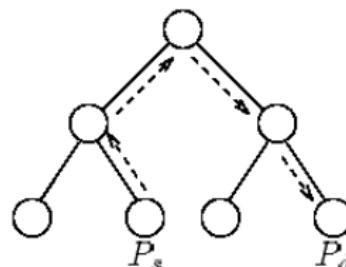


Figura 8: Comunicarea

Arborele gras

- Arbore binar complet.
- Fiecare pereche de noduri dispune de un canal de comunicare propriu.
- Numărul de muchii care leagă un nod interior de fii este egal cu $2^{h-nivel-1}$, unde h este adâncimea arborelui iar nivel(rădăcină)=1, nivel(frunză)=h.

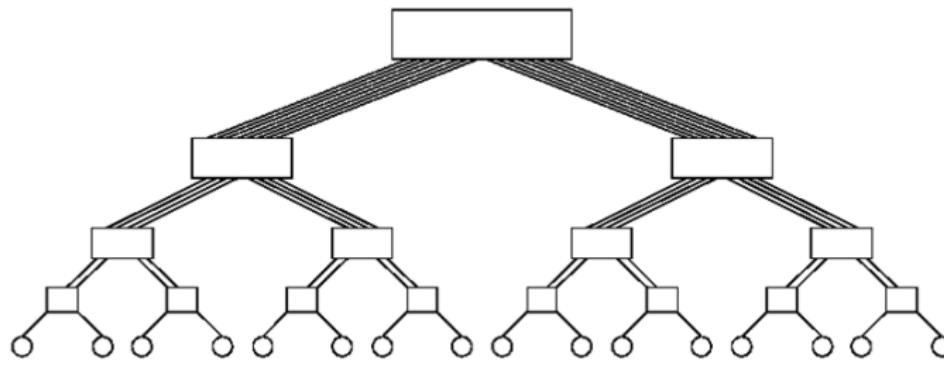


Figura 9: Arborele gras



Fluturele

- Arbore cu mai multe rădăcini!
- Exact un drum între oricare două noduri de pe frontieră.
- Multimea nodurilor V este organizată ca un tablou bidimensional de dimensiune $n \times m$, $n = 2^{m-1}$, prin intermediul unei funcții de indexare $I : V \rightarrow L = \{(i, j) / 0 \leq i < n, 0 \leq j < m\}$.
- Notăm cu $V_{i,j}$ nodul dat de $I^{-1}((i, j))$. Nodul $V_{i,j}$, situat pe coloana j este conectat la coloana $j+1$ prin nodurile $V_{i,j+1}$ și $V_{i \oplus 2^{m-j-2} + \lfloor i/2^{m-j-1} \rfloor \times 2^{m-j-1}, j+1}$, unde \oplus este adunarea modulo 2^{m-j-1} .

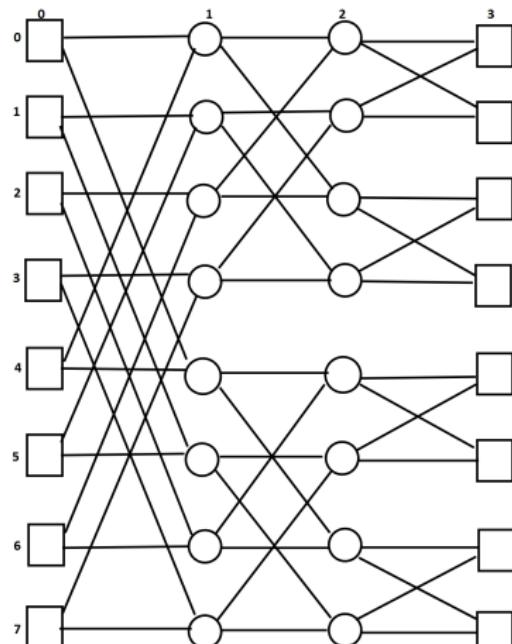


Figura 10: Fluturele

Topologia Banyan

- Tot arbore cu mai multe rădăcini!
- Mulțimea nodurilor V este organizată ca un tablou bidimensional de dimensiune $n \times m$, $n = 2^{m-1}$, prin intermediul unei funcții de indexare $I : V \rightarrow L = \{(i, j) / 0 \leq i < n, 0 \leq j < m\}$.
- Notăm cu $V_{i,j}$ nodul dat de $I^{-1}((i, j))$. Nodul $V_{i,j}$, situat pe coloana j este conectat la coloana $j+1$ prin nodurile $V_{\lfloor i/2 \rfloor \oplus 0 + \lfloor i/2^{m-j-1} \rfloor \times 2^{m-j-1}, j+1}$ și $V_{\lfloor i/2 \rfloor \oplus 2^{m-j-2} + \lfloor i/2^{m-j-1} \rfloor \times 2^{m-j-1}, j+1}$, unde \oplus este adunarea modulo 2^{m-j-1} .

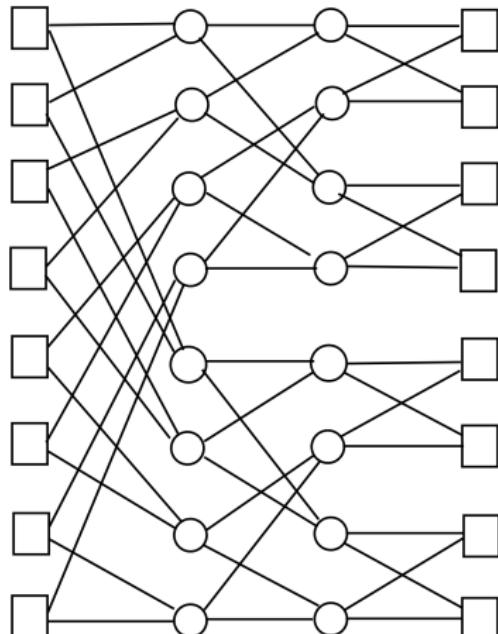


Figura 11: Topologia Banyan



Plasa

- Este o generalizare a topologiei de tip lanț.
- Mulțimea nodurilor V este organizată ca un tablou k -dimensional prin intermediul unei funcții de indexare
 $I : V \rightarrow L = \{(i_{k-1}, i_{k-2}, \dots, i_0) / 0 \leq i_j < n_j, j = k-1, k-2, \dots, 0\}$, unde n_j este mărimea celei de-a j -a dimensiuni, iar $n = n_{k-1} * n_{k-2} * \dots * n_0$.
- Notăm cu $V_{i_{k-1}, i_{k-2}, \dots, i_0}$ nodul dat de $I^{-1}(i_{k-1}, i_{k-2}, \dots, i_0)$.
- Nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j, \dots, i_0}$ este adiacent cu nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j + 1, \dots, i_0}$, dacă $i_j + 1 \leq n_j$.
- Nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j, \dots, i_0}$ este adiacent cu nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j - 1, \dots, i_0}$, dacă $i_j - 1 \geq 0$.

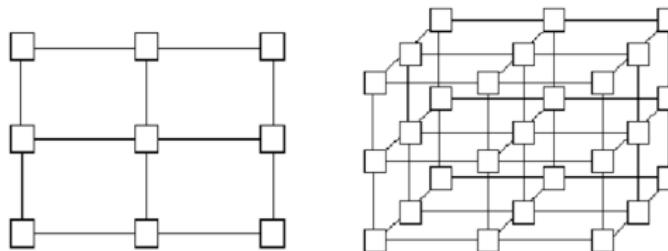


Figura 12: Plase



Plasa circulară

- Este o generalizare a topologiei de tip inel.
- Nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j, \dots, i_0}$ este adiacent cu nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j \oplus 1, \dots, i_0}$, unde \oplus este adunarea modulo n_j .
- Nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j, \dots, i_0}$ este adiacent cu nodul $V_{i_{k-1}, i_{k-2}, \dots, i_j \ominus 1, \dots, i_0}$, unde \ominus este scăderea modulo n_j .

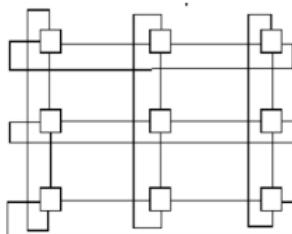


Figura 13: Plasa circulară



Plasa de arbori

- Mulțimea nodurilor, V este partită în două submulțimi M și T
- Nodurile din M sunt organizate sub forma unei matrice.
- Conectarea nodurilor din M se realizează prin intermediul unor arbori binari formați din noduri din mulțimea T , câte un arbore pentru fiecare linie și fiecare coloană.
- Nodurile din M aparținând unei linii/coloane constituie frunzele arborelui asociat liniei/coloanei respective.

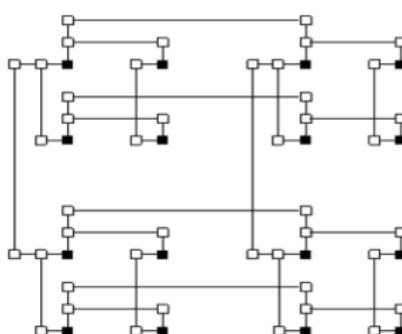


Figura 14: Plasa de arbori



Hipercubul (Cubul cu k dimensiuni)

- Numărul n al nodurilor grafului ce reprezintă o astfel de topologie este 2^k , unde k este dimensiunea cubului.
- Fie $i_{k-1}i_{k-2}\dots i_0$ reprezentarea binară a lui $i \in \{0, \dots, n-1\}$ și fie $i^{(j)}$ numărul a cărui reprezentare binară este $i_{k-1}i_{k-2}\dots i_{j+1}\tilde{i}_j i_{j-1}\dots 0$, unde $\tilde{i}_j = 1 - i_j, 0 \leq j < k$.
- Nodul V_i este adiacent cu nodurile din mulțimea $\{V_{i^{(j)}}; 0 \leq j < k\}$.

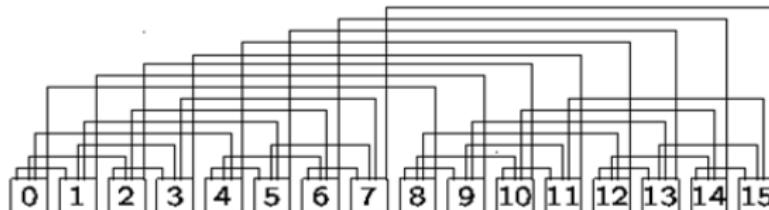


Figura 15: Hipercub liniarizat



Hipercuburi cu 0, 1, 2 , 3 și 4 dimensiuni

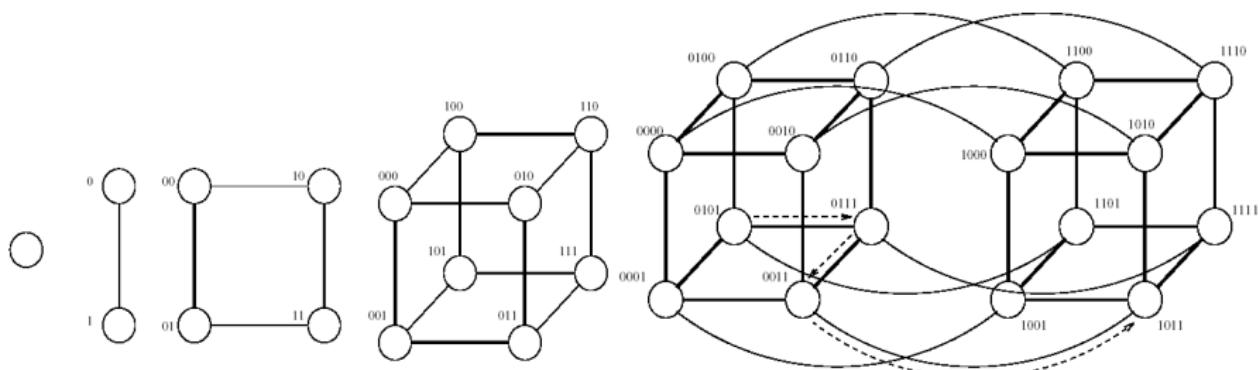


Figura 16: Hipercuburi cu 0, 1, 2 , 3 și 4 dimensiuni



Amestecul perfect

- Fie $i_{k-1}i_{k-2}\dots i_0$ reprezentarea binară a lui $i \in \{0, \dots, n-1\}$.
- Nodul V_i este conectat la nodurile $V_{i(0)}$, $V_{i_{k-2}i_{k-3}\dots i_0i_{k-1}}$ și $V_{i_0i_{k-1}i_{k-2}\dots i_1}$, $0 \leq i < n$,
- n, q, i și $i^{(0)}$ sunt definite ca în cazul hipercubului..

000 001 010 011 100 101 110 111

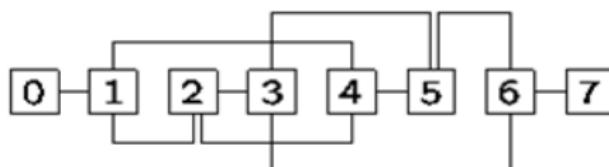
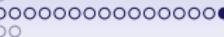


Figura 17: Amestec perfect

Criterii de evaluare a topologiilor regulate

- Distanța = cel mai scurt drum între două unități de procesare.
- Diametrul = distanța maximă între două unități de procesare.
- Lățimea bisecției = numărul minim de linii de comunicații care pot fi întrerupte pentru a sparge rețeaua în două rețele de dimensiuni aproximativ egale.
- Costul = numărul de linii de comunicații.



Evaluarea a 8 topologii regulate, cu p unități de procesare

Topologia	Diametrul	Lățimea bisecției	Costul
Graful complet	1	$\lfloor p^2/4 \rfloor$	$p(p-1)/2$
Steaua	2		$p-1$
Arborele binar complet	$2\lfloor \log(p+1)/2 \rfloor$	1	$p-1$
Lanțul	$p-1$	1	$p-1$
Inelul	$\lfloor p/2 \rfloor$	2	p
Plasa 2D	$2(\sqrt{p}-1)$	\sqrt{p}	$2(p-\sqrt{p})$
Plasa 2D ciclică	$2\lfloor (\sqrt{p}/2) \rfloor$	4	$2p$
Hipercubul	$\log p$	$p/2$	$p \log p/2$



Graful oarecare

- Internet-ul.
- Web-ul.

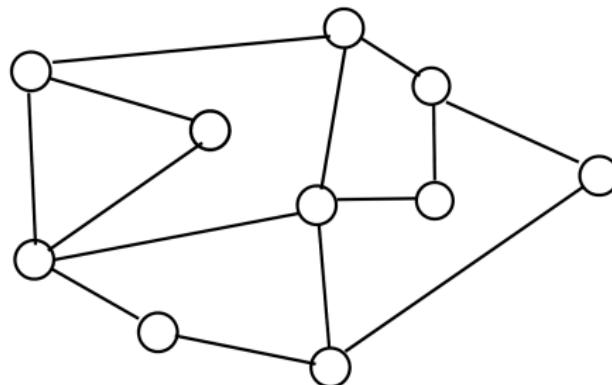


Figura 18: Graful oarecare

- 6

- 8

- 8

Arborele oarecare

- rețelele locale.

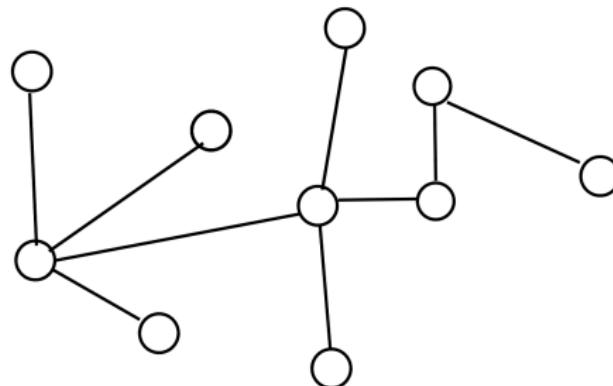
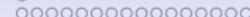


Figura 19: Arborele oarecare



Bibliografie

- V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003
 - H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, John Wiley & Sons, Inc., 2004
 - K. Berman, J. Paul, Algorithms: Sequential, Parallel, and Distributed, Thomson Learning, Inc., 2005
 - T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Addison-Weslwy, 2005

Algoritmi paraleli și distribuiți

Modele de calcul

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Modele de calcul paralel/distribuit

Taxonomii

Modelul Shared-Memory

Modelul PRAM

Modelul Message-Passing

Modelul sistolic

Parametri de performanță a unui algoritm paralel/distribuit

Bibliografie



Modele de calcul paralel/distribuit

Taxonomia lui Flynn

- Formulată de Michael J. Flynn, în 1966. Criteriul de clasificare este concurența instrucțiunilor și a fluxurilor de date.
- Un calculator monoprocesor poate executa la un moment dat o singură instrucțiune, în timp ce un sistem multiprocesor poate executa simultan mai multe instrucțiuni asupra mai multor fluxuri de date.
- Taxonomia:
 - **SISD** Single Instruction / Single Data stream (o singură instrucțiune / un singur flux de date). Exemplu: calculatoarele monoprocesor;
 - **SIMD** Single Instruction / Multiple Data streams (o singură instrucțiune / mai multe fluxuri de date). Exemplu: calculatoarele vectoriale, sistemele GPU;
 - **MISD** Multiple Instructions / Single Data stream (mai multe instrucțiuni / un singur flux de date). Exemplu: sistemele *pipeline*;
 - **MIMD** Multiple Instructions / Multiple Data streams (mai multe instrucțiuni / mai multe fluxuri de date). Exemplu: calculatoarele multiprocesor, clusterele, *grid*-urile, norii de calcul (*cloud*).
- Tipuri de sisteme **MIMD**
 - **SPMD** Single Program / Multiple Data streams (un singur program / mai multe fluxuri de date);
 - **MPMD** Multiple Program / Multiple Data streams (mai multe programe / mai multe fluxuri de date).



Modele arhitecturale

- Modele bazate pe comunicarea prin memoria comună (Shared-Memory)
- Modele bazate pe comunicarea prin mesaje (Message-Passing)



Modelul Shared-Memory

- Un sistem Shared-Memory este compus din n unități de procesare p_0, p_1, \dots, p_{n-1} și m registri de memorie, r_0, r_1, \dots, r_{m-1} .
- Fiecare registru este caracterizat prin:
 - valoarea care poate fi memorată în registrul;
 - operațiile care pot fi efectuate asupra conținutului registrului;
 - valoarea returnată de fiecare operație ;
 - valoarea aflată în registrul după fiecare operație.
- Exemplu - registrul *integer-valued read/write*:
 - Valorile v memorate de registrul r sunt numere întregi.
 - Operațiile suportate sunt `read(r,v)` and `write(r,v)`.
 - Operația `read(r,v)` întoarce valoarea v și lasă conținutul registrului neschimbă.
 - Operația `write(r,v)` nu returnează nimic, dar schimbă conținutul registrului r , înscriind acolo valoarea v .
- Evenimente:
 - Evenimentele sunt *pași computaționali*, executați de unitățile de procesare .
 - Un pas computațional, executat de o unitate de procesare p_i , este executat astfel:
 - În funcție de starea sa, p_i selectează un registru r_j și una din operațiile asociate acestui registru.
 - Execută operația asupra conținutului lui r_j .
 - Își schimbă starea, în funcție de starea curentă și valoarea returnată de operația executată.



Modelul PRAM

- Este derivat din modelul Shared-Memory.
- Unitățile de procesare sunt procesoare simple.
- Cele $p(n)$ procesoare $P_1, \dots, P_{p(n)}$ comunică între ele prin intermediul regiștrilor de memorie $R_1, \dots, R_{m(n)}$.
- Procesoarele execută în mod sincron aceeași instrucțiune, sub control central.
- Deși instrucțiunea este aceeași, regiștrii de memorie asupra căror acționează procesoarele pot difera de la un procesor la altul.
- Conform taxonomiei Flynn, mașina PRAM este un sistem SIMD.

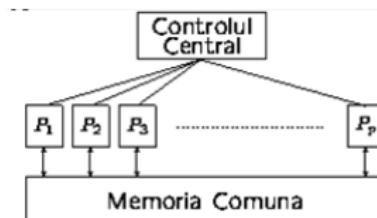


Figura 1: Mașina PRAM



Tipuri de mașini PRAM

- La un moment dat t , un procesor poate accesa un registru de memorie pentru citire sau scriere.
- În funcție de accesul concurrent permis, mașinile PRAM pot fi de tipul
 - EREW (Exclusive-Read,Exclusive-Write),
 - CREW (Concurrent-Read,Exclusive-Write),
 - CRCW (Concurrent-Read,Concurrent-Write).
- O mașină EREW-PRAM nu permite acces simultan la un registru de memorie. Aceasta înseamnă că, la un moment dat, cel mult un procesor poate accesa un registru de memorie.
- Ultimele două tipuri (CREW,CRCW) permit citiri simultane din același registru de memorie. Diferențierea derivă din accesul la scriere.
- Tipul CREW-PRAM definește o mașină în care un singur procesor poate accesa la un moment dat un registru de memorie pentru scriere.
- O mașină CRCW-PRAM este caracterizată prin faptul că sunt permise accesări concurente pentru scriere.



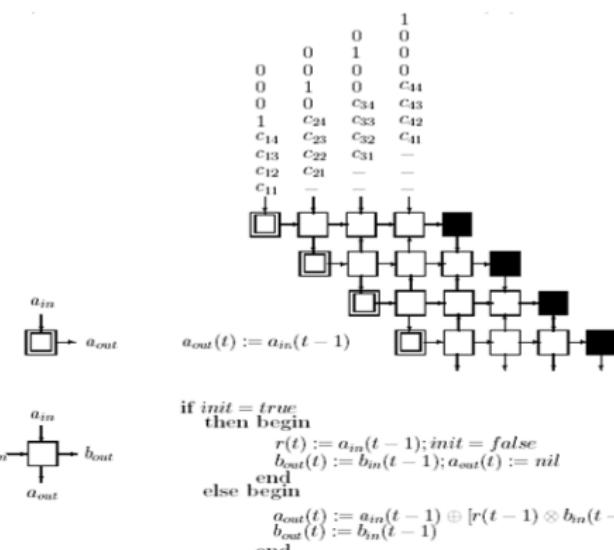
Modelul Message-Passing

- Un sistem Message-Passing este compus din n unități de procesare p_0, p_1, \dots, p_{n-1} care comunică între ele prin mesaje transmise pe canalele de comunicare care le conectează.
- Topologia de comunicare este reprezentată printr-un graf $G = (v, E)$, care asignează fiecărui vârf v_i o unitate de procesare p_i . Fiecare muchie $e_k \in E$ reprezintă un canal de comunicare directă între două unități de procesare p_i și p_j .
- Fiecare unitate de procesare p_i poate fi într-o stare din mulțimea de stări Q_i . Fiecare stare include două mulțimi de regiștri: $(inbuf_i^k, k = 1, \dots, d)$ și $(outbuf_i^k, k = 1, \dots, d)$, unde d este gradul vârfului v_i asignat lui p_i .
- Un registru $inbuf_i^k$ conține mesajele pe care unitatea de procesare p_i le-a primit de la unitatea de procesare vecină p_j , prin canalul de comunicare directă k . Un registru $outbuf_i^k$ conține mesajele pe care unitatea de procesare p_i le-a transmis spre unitatea de procesare vecină p_j , prin canalul de comunicare directă k .
- Mulțimea de stări Q_i conține o submulțime de stări speciale, în care regiștrii $inbuf_i$ nu memorează nici un mesaj.
- Evenimentele sunt pași *computaționali* și pași de *comunicare*.
- În timpul unui pas computațional, o unitate de procesare p_i citește și apoi resetează regiștrii $inbuf_i^k, k = 1, \dots, d$ și produce valori noi pentru $outbuf_i^k, k = 1, \dots, d$, schimbându-și în același timp starea internă.
- Evenimentul de comunicare înseamnă o operație de transmisie, i.e. un mesaj memorat în $outbuf$ -ul unei unități de procesare p_i este transmis, prin canalul de comunicare directă, la destinația p_j , unde va fi memorat în $inbuf$.



Modelul sistolic

- Este derivat din modelul Message-Passing.
- Arhitecturile utilizate de modelul sistolic au două proprietăți foarte importante: regularitatea și interconexiunile locale.
- O arhitectură sistolică este
 - compusă din celule computaționale de același tip sau dintr-o mulțime restrânsă de tipuri,
 - interconectate regulat prin conexiuni locale.



Y. Robert si D. Trystram
Problema algebrică a drumurilor

Figura 2: Exemplu de arhitectură sistolică

Parametri de performanță a unui algoritm paralel/distribuit

- Parametri de performanță a unui algoritm paralel:
 - timpul de execuție paralelă: $T_p(n)$, p = numărul unităților de procesare, n = dimensiunea problemei;
 - timpul de execuție sequențială: $T_s(n)$;
 - factorul de încarcare: $L = n/p$;
 - accelerarea: $S(n) = T_s(n)/T_p(n)$, $T_s(n)$ = timpul de execuție al celui mai rapid algoritm secvențial;
 - eficiență: $E(n) = S(n)/p = T_s(n)/(pT_p(n))$;
 - costul: $C(n) = Costul = pT_p(n)$
- Parametri de performanță a unui algoritm distribuit:
 - numărul de mesaje transmise prin rețeaua de interconectare: $M(n)$, p = numărul unităților de procesare.

Bibliografie

- V. Kumar, A. Grama A. Gupta & G Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Addison Wesley, 2003
- H. Attiya, J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, John Wiley & Sons, Inc., 2004
- K. Berman, J. Paul, *Algorithms: Sequential, Parallel, and Distributed*, Thomson Learning, Inc., 2005
- T. Mattson, B. Sanders, B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005

Algoritmi paraleli și distribuiți Performanță, Comunicare

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași



Parametri de performanță a unui algoritm paralel/distribuit

-

Comunicarea în sistemele de calcul paralel/distribuit



Cuprins

Parametri de performanță a unui algoritm paralel/distribuit
Comunicarea în sistemele de calcul paralel/distribuit

Tipuri de comunicare

Comunicare prin rețele de interconectare cu topologie regulată

Difuzia *unu la toți*

Difuzia *toți la toți*

Comunicarea personalizată *unu la toți*

Comunicarea personalizată *toți la toți*

Comunicare prin rețele de interconectare cu topologie arbore oarecare

Comunicare *grup la grup*



Parametri de performanță a unui algoritm paralel/distribuit

- Parametri de performanță a unui algoritm paralel:
 - timpul de execuție paralelă: $T_p(n)$, $p =$ numărul unităților de procesare, $n =$ dimensiunea problemei;
 - timpul de execuție secvențială: $T_s(n)$;
 - factorul de încarcare: $L = n/p$;
 - accelerarea: $S(n) = T_s(n)/T_p(n)$, $T_s(n) =$ timpul de execuție al celui mai rapid algoritm secvențial;
 - eficiență: $E(n) = S(n)/p = T_s(n)/(pT_p(n))$;
 - costul: $C(n) = Costul = pT_p(n)$
- Parametri de performanță a unui algoritm distribuit:
 - numărul de mesaje transmise prin rețeaua de interconectare: $M(n)$, $p =$ numărul unităților de procesare.



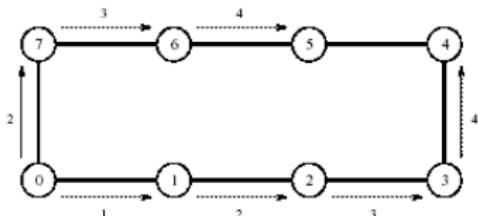
Tipuri de comunicare

- *unu la unu.*
- *comunicări colective:*
 - *unu la un grup, unu-la-toți: broadcast, scatter;*
 - *un grup la un grup, un grup la toți, toți la un grup, toți la toți;*
 - *un grup la unu, toți la unu: reduction, gather.*

Difuzia *unu la toti*

Definiție: O unitatea de procesare trimite același mesaj M la toate celelalte.

Difuzia *unu la toti* pe un inel

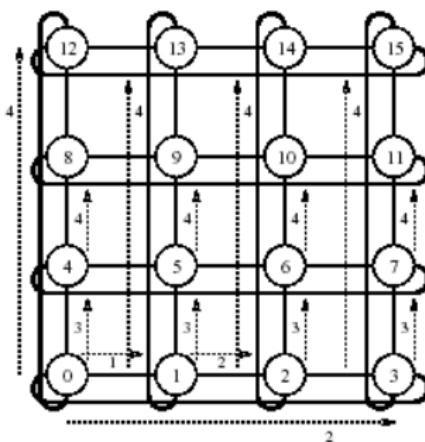


Copyright © V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

Figura 1: Exemplu de difuzie *unu la toti* pe un inel

- Timpul de execuție: $T = (T_s + T_c m) \lceil p/2 \rceil$, unde
 - T_s = timpul de inițializare a transmisiei;
 - T_c = timpul de transmisie a unui cuvânt;
 - m = numărul de cuvinte care compun mesajul M ; prin cuvânt se înțelege unitatea lexicală care compune mesajul (exemplu: octet, 2 octeți, etc.);
 - p = numărul de unități de procesare.

Difuzia *unu la toți* pe o plasă circulară

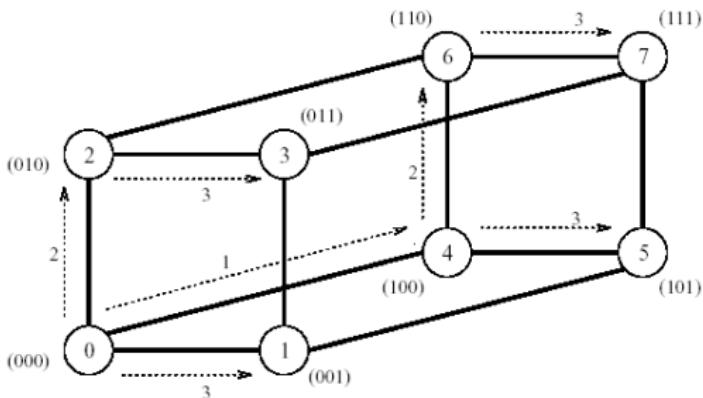


Copyright © V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

Figura 2: Exemplu de difuzie *unu la toți* pe o plasă circulară

Timpul de execuție: $T = (T_s + T_c m) \lceil \sqrt{p}/2 \rceil$.

Difuzia *unu la toti* pe un hipercub



Copyright © V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

Figura 3: Exemplu de difuzie *unu la toti* pe un hipercub

Timpul de execuție: $T = (T_s + T_c m) \log p$



Algoritmul de difuzie unu la toți pe un hipercub

- *Notății:*

- H este un hipercub cu d dimensiuni.
- M este un mesaj care urmează a fi transmis tuturor unităților de procesare.
- i este identificatorul unității de procesare care executa algoritmul de difuzie unu la toți pe hipercub.

- *Premise:*

- Inițial, mesajul M se află în nodul 0 al hipercubului H .

DIFUZIE_UNU_LA_TOTI_PE_HIPERCUB(H, d, i, M)

```

1 masca  $\leftarrow 2^d - 1$  /* inițializeaza cu 1 toti bitii mastii */
2 for  $k \leftarrow d - 1$  downto 0
3 do masca  $\leftarrow masca \text{ XOR } 2^k$ ; /* seteaza pe 0 al k-lea bit al mastii */
4 if ( $i \text{ AND } masca$ ) = 0
5 then /* daca ultimii k biti sunt 0 */
6 if ( $i \text{ AND } 2^k$ ) = 0
7 then destinatie  $\leftarrow i \text{ XOR } 2^k$ ;
8         trimite  $M$  la destinatie;
9 else sursa  $\leftarrow i \text{ XOR } 2^k$ ;
10        primeste  $M$  de la sursa.

```



Algoritmul generalizat de difuzie *unu la toti* pe un hipercub

- *Notări:*

- H este un hipercub cu d dimensiuni.
- M este un mesaj care urmează a fi transmis tuturor unităților de procesare.
- i este identificatorul unității de procesare care executa algoritmul de difuzie *unu la toti* pe hipercub.

- *Premise:*

- Inițial, mesajul M se află în nodul nodul s al hipercubului H .

DIFUZIE_UNU_LA_TOTI_PE_HIPERCUB(H, d, i, s, M)

```

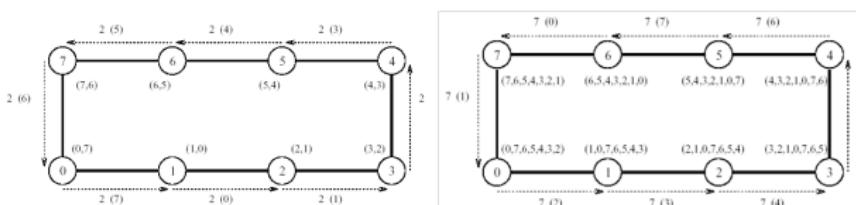
1  id_virtual  $\leftarrow i \text{ XOR } s;
2  masca  $\leftarrow 2^d - 1$  /* initializeaza cu 1 toti bitii mastii */
3  for  $k \leftarrow d - 1$  downto 0
4  do masca  $\leftarrow \text{masca XOR } 2^k$ ; /* seteaza pe 0 al k-lea bit al mastii */
5    if (id_virtual AND masca) = 0
6      then /* daca ultimii k biti sunt 0 */
7        if (id_virtual AND  $2^k$ ) = 0
8          then destinatie_virtuala  $\leftarrow id\_virtual \text{ XOR } 2^k$ ;
9            trimite M la destinatie_virtuala XOR s;
10           else sursa_virtuala  $\leftarrow id\_virtual \text{ XOR } 2^k$ ;
11             primește M de la sursa_virtuala XOR s.$ 
```



Difuzia *toți la toți*

Definiție: Fiecare unitatea de procesare i trimite un mesaj M_i la toate celelalte.

Difuzia *toți la toți* pe un inel



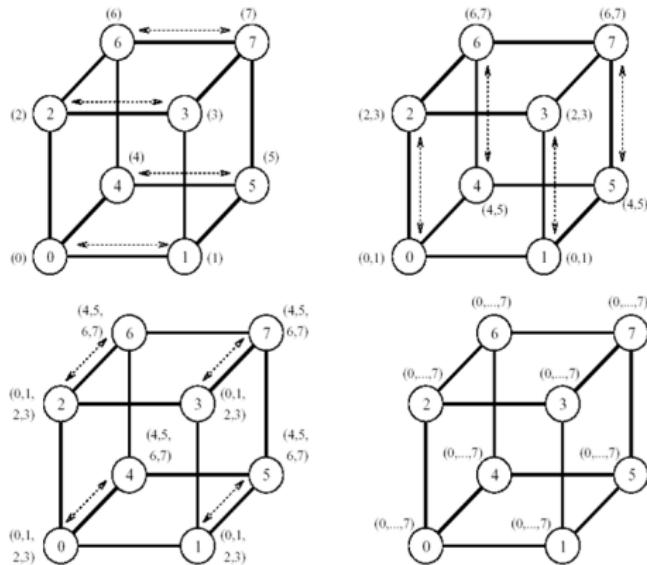
Copyright © V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

Figura 4: Exemplu de difuzie *toți la toți* pe un inel - pașii 2 și 7

$$\text{Timpul de execuție: } T = (T_s + T_c m)(p - 1)$$



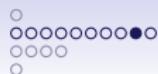
Difuzia *toți la toți* pe un hipercub



Copyright © V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

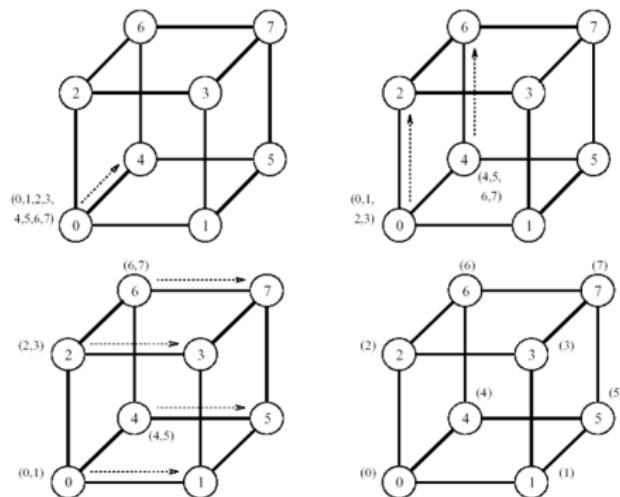
Figura 5: Exemplu de difuzie *toți la toți* pe un hipercub

$$\text{Timul de executie: } T = T_s \log p + T_c m(p-1)$$



Comunicarea personalizată *unu la toti*

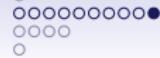
Definiție: O unitate de procesare i trimite un trimite câte un mesaj M_{ij} la fiecare unitate de procesare j .



Copyright © V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

Figura 6: Exemplu de comunicarea personalizată *unu la toti* pe un hipercub

$$\text{Timpul de execuție: } T = T_s \log p + T_c m(p-1)$$



Comunicarea personalizată *toți la toti*

Definiție: Fiecare unitate de procesare i trimite căte un mesaj M_{ij} la fiecare unitate de procesare j .

Difuzia pe un arbore

Premise : Inițial, unitatea de procesare p_r (rădăcina) deține mesajul M , care urmează a fi difuzat în arbore.

cod p_r

DIFUZIE_UNU_LA_TOTI_PE_ARBORE(T, p_r, M)

- 1 *trimite M catre fii;*
- 2 *termina execuția.*

cod $p_i \neq p_r$

DIFUZIE_UNU_LA_TOTI_PE_ARBORE(T, p_i, M)

- 1 *asteapta primirea mesajului M de la parinte;*
- 2 *dupa primirea lui M trimite mesajul la fii;*
- 3 *termina execuția.*



Comunicarea de tip *toți la unu* pe un arbore

Premise : Inițial, fiecare unitate de procesare i deține mesajul M_i , care urmează a fi transmis spre rădăcină.

cod $p_i =$ frunza

DIFUZIE_TOTI_LA_UNU_PE_ARBOR(E) (T, p_i, M_i)

- 1 trimit M_i catre parinte;
- 2 termina executia.

cod $p_i =$ nod interior

DIFUZIE_TOTI_LA_UNU_PE_ARBOR(E) (T, p_i, M_i)

- 1 asteapta primirea valorilor $M_{i_0}, M_{i_1}, \dots, M_{i_{k-1}}$ de la fii;
- 2 dupa primire, calculeaza $M_i \leftarrow f(M_i, M_{i_0}, M_{i_1}, \dots, M_{i_{k-1}})$;
- 3 trimit M_i catre parinte;
- 4 termina executia.

cod p_r

DIFUZIE_TOTI_LA_UNU_PE_ARBOR(E) (T, p_r, M_r)

- 1 asteapta primirea valorilor $M_{r_0}, M_{r_1}, \dots, M_{r_{k-1}}$ de la fii;
- 2 dupa primire, calculeaza $M_r \leftarrow f(M_r, M_{r_0}, M_{r_1}, \dots, M_{r_{k-1}})$;
- 3 termina executia.

Complexitatea

Teorema (3)

Numărul de pași, după care toate unitățile de procesare termină execuția algoritmului = h (adâncimea arborelui T).

Demonstrație.

Inducție. □

Teorema (4)

Numărul de mesaje, care sunt transmise de toate unitățile de procesare, în timpul execuției algoritmului = $n - 1$.

Demonstrație.

Pe fiecare muchie este transmis câte un mesaj, de la fii către părinte. □



Comunicare *grup la grup*

- Comunicarea între grupuri este similară cu comunicare între toate unitățile de procesare ale sistemului.
- Deosebirile sunt următoarele:
 - Pe parcursul execuției algoritmului de comunicare doar unitățile de procesare membre ale grupurilor vor iniția un transfer sau vor memora un mesaj.
 - Celelalte unități de procesare au rol de retransmisie a mesajelor (releu).

Algoritmi paraleli și distribuiți Algoritmi fundamentali

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași



-
-
-
-
-
-
-

-
-
-
-
-

-
-
-
-
-

Cuprins

Comprimarea (Reducerea)

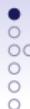
- Algoritm recursiv de comprimare
- Algoritm paralel de comprimare
- Corectitudinea și complexitatea
- Reducerea numărului unităților de procesare
- Comentarii

Calculul Prefixelor (Scan)

- Algoritmul de calcul al prefixelor în cazul în care operatia \oplus admite element simetric
- Algoritmul de calcul al prefixelor în cazul în care operatia \oplus nu admite element simetric
- Corectitudinea și complexitatea
- Comentarii

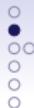
Scurtcircuitarea (dublarea)

- Algoritm paralel de scurtcircuitare
- Complexitatea
- Aplicații ale scurtcircuitării: Calcul ranguri
- Comentarii



Comprimarea (Reducerea)

- Fie M o mulțime de $n = 2^m$ elemente, $M = \{a_i / i = 0, 1, \dots, n-1\} \subseteq M_r$ = mulțime de referință.
- Mulțimea M urmează a fi procesată pentru a calcula valoarea $a_1 \oplus \dots \oplus a_n$, unde \oplus este o lege de compozitie asociativă definită pe M_r .
- Mulțimea de referință M_r poate fi \mathbb{R} , iar \oplus poate fi $+$, \min , \max , etc.



Algoritm recursiv de comprimare

- *Notății:*

- $A[0..n-1]$ este un tablou de dimensiune $n = 2^m$.
- Prin $(A, k, k+l-1)$ se notează segmentul din tabloul A care începe pe poziția k și se termină pe poziția $k+l-1$

- *Premise:*

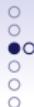
- Datele de intrare sunt inițial memorate în tabloul $A[0..n-1]$.
- Inițial, $k = 0$ și $l = n$.

COMPRIM_RECURSIV($A, k, k+l-1$)

```

1  if  $l = 1$ 
2    then return  $A[k]$ 
3    else return COMPRIM_RECURSIV( $A, k, k+l/2-1$ ) ⊕ COMPRIM_RECURSIV( $A, k+l/2, k+l-1$ ).

```



Algoritm paralel de comprimare

- *Notății:*
 - $A[0..2n-1]$ este un tablou de dimensiune $2n = 2^{m+1}$.
- *Premise:*
 - Datele de intrare sunt memorate în tabloul $A[0..2n-1]$, în locațiile $A[n], A[n+1], \dots, A[2n-1]$.

```

COMPRIM_PARALEL( $A, \oplus$ )
1  for  $k \leftarrow m-1$  down to 0
2  do for all  $j : 2^k \leq j \leq 2^{k+1} - 1$ 
3    do in parallel
4       $A[j] \leftarrow A[2j] \oplus A[2j + 1];$ 

```



Exemplu de execuție a algoritmului de comprimare

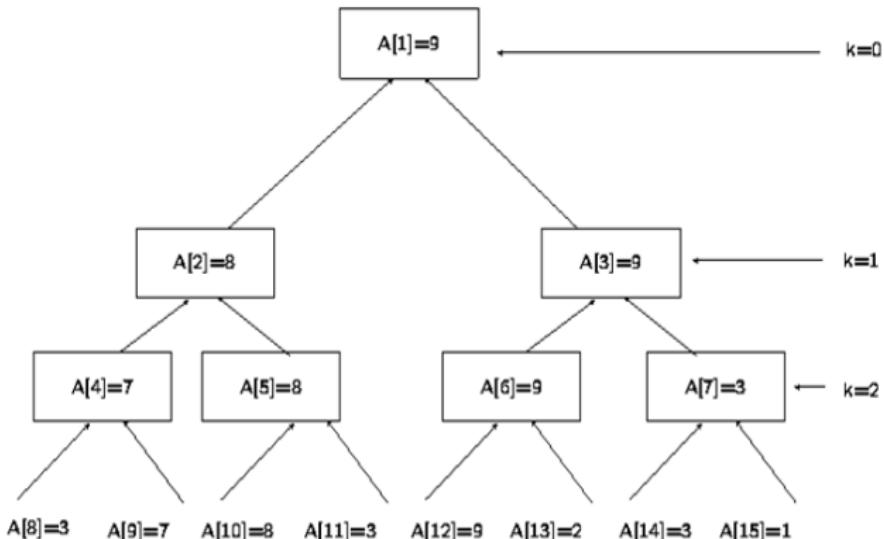


Figura 1: Exemplu de execuție a algoritmului de comprimare



Corectitudinea și complexitatea

Teorema (1)

La terminarea execuției algoritmului $A[1] = a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$.

Teorema (2)

Complexitatea timp a algoritmului de comprimare (implementat pe o masina CREW-PRAM sau pe o arhitectura VLSI de tip arbore binar) este $O(\log n)$.

Demonstrație.

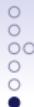
Adâncimea arborelui de calcul este $\log n$. □

Teorema (3)

Numarul unităților de procesare este $p \geq \lceil \frac{n}{2} \rceil$

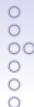
Demonstrație.

In prima iterație sunt active $\lceil \frac{n}{2} \rceil$ unități de procesare. Apoi numărul lor scade pâna la 1. □



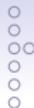
Comentarii

- Comprimarea nu se reduce la operarea cu elementele unei mulțimi de numere.
- $a_0 \oplus a_1 \oplus \dots a_{n-1}$ pot fi n fișiere, care urmează a fi concatenate.
- Mai pot fi n instanțe ale unei tabele ale unei baze de date distribuite, care trebuie unificate.
- În aceste situații algoritmul este distribuit.



Calculul Prefixelor (Scan)

- Tehnica calculării prefixelor dezvoltă tehnica comprimării, în care parcurgerea arborelui asociat execuției algoritmului se face dinspre frunze spre rădăcină.
- Algoritmii de calcul a prefixelor parcurg arborele de execuție în ambele sensuri.
- Calculul prefixelor constă în determinarea valorilor $a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$, unde \oplus este o operație binară asociativă.



Algoritmul de calcul al prefixelor în cazul în care operatia \oplus admite element simetric

- *Notății:*

- $A[0..2n-1]$ este un tablou de dimensiune $2n = 2^{m+1}$.

- *Premise:*

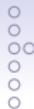
- Datele de intrare sunt memorate în tabloul $A[0..n-1]$, în locațiile $A[n], A[n+1], \dots, A[2n-1]$.

CALCUL_PREFIXE_ \oplus _ADMITE_ELEMENT_SIMETRIC(A, \oplus)

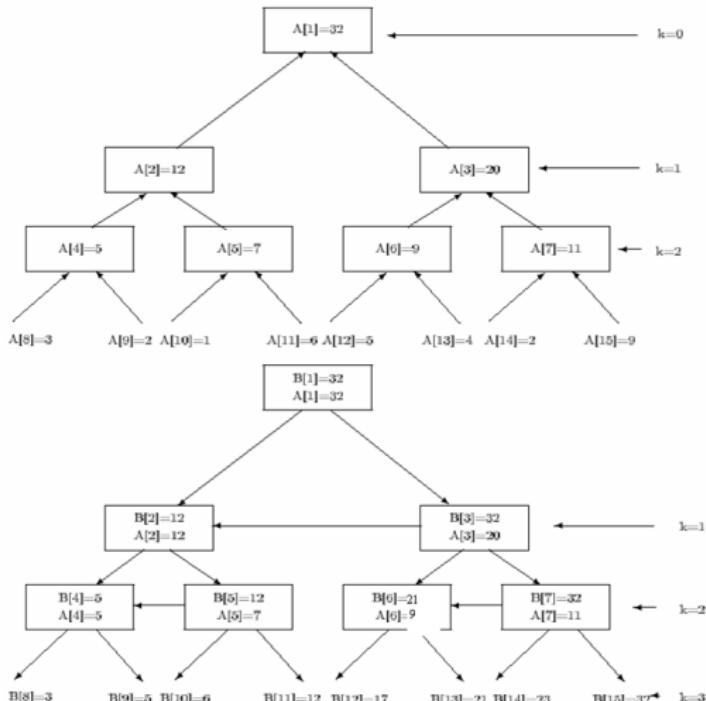
```

1  /* Comprimare */
2  for k ← m − 1 down to 0
3  do for all j :  $2^k \leq j \leq 2^{k+1} - 1$ 
4      do in parallel  $A[j] \leftarrow A[2j] \oplus A[2j + 1];$ 
5  /* Calcul prefixe */
6   $B[1] \leftarrow A[1];$ 
7  for k ← 1 to m
8  do for all j :  $2^k \leq j \leq 2^{k+1} - 1$ 
9      do in parallel;
10         if bit0(j) = 1
11             then  $B[j] \leftarrow B[(j - 1)/2];$ 
12             else  $B[j] \leftarrow B[j/2] \oplus (-A[j + 1]);$ 

```



Exemplu de execuție a algoritmului de calcul al prefixelor în cazul în care operatia \oplus admite element simetric





Algoritmul de calcul al prefixelor în cazul în care operatia \oplus nu admite element simetric

- Notății:* $A[0..2n-1]$ și $B[0..2n-1]$ sunt tablouri de dimensiune $2n = 2^{m+1}$.
- Premise:* Datele de intrare sunt memorate în tabloul $A[0..2n-1]$, în locațiile $A[n], A[n+1], \dots, A[2n-1]$. La final,
 $B[n] = a_0$, $B[n+1] = a_0 \oplus a_1, \dots, B[2n-1] = a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$.

CALCUL_PREFIXE_⊕_NU ADMITE_ELEMENT_SIMETRIC(A, \oplus)

```

1  /* Comprimare */
2  for k ← m − 1 down to 0
3  do for all j :  $2^k \leq j \leq 2^{k+1} - 1$ 
4      do in parallel  $A[j] \leftarrow A[2j] \oplus A[2j + 1]$ ;
5  /* Calcul prefixe */
6  for k ← 0 to m
7  do for all j :  $2^k \leq j \leq 2^{k+1} - 1$ 
8      do in parallel
9          if  $j = 2^k$ 
10             then  $B[j] \leftarrow A[j]$ ;
11             else if  $bit_0(j) = 1$ 
12                 then  $B[j] \leftarrow B[(j - 1)/2]$ ;
13                 else  $B[j] \leftarrow B[(j - 2)/2] \oplus (A[j])$ ;

```



Corectitudinea și complexitatea

Teorema (4)

La terminarea execuției algoritmului,

$$B[n] = a_0, \quad B[n+1] = a_0 \oplus a_1, \dots, \quad B[2n-1] = a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}.$$

Teorema (5)

Complexitatea timp a algoritmului paralel de calcul al prefixelor (implementat pe o masina CREW-PRAM sau pe o arhitectura VLSI de tip arbore binar) este $O(\log n)$.

Demonstrație.

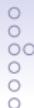
Adâncimea arborelui de calcul este $\log n$. □

Teorema (6)

Numarul unităților de procesare este $\lceil \frac{n}{\log n} \rceil$.

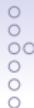
Demonstrație.

Necesarul de unități de procesare este același cu cel de la comprimare . □



Comentarii

- Ca și în cazul comprimării, calculul prefixelor nu se reduce la operarea cu elementele unei multimi de numere.
- Compilatoarele folosesc masiv calculul prefixelor.
- Descoperirea patternurilor frecvente în colecții de date de dimensiuni foarte mari, conține ca task de bază calculul de prefixe.
- Regăsirea informației este un alt domeniu de aplicare a calculului prefixelor.
- Algoritmii paraleli pot aduce preformanță sporită.
- Recurențele sunt cel mai clar exemplu de aplicare a calculului prefixelor.
 - Fie recurența $x_k = x_{k-1} \oplus a_k, k > 0, x_0 = a_0$.
 - Dacă \oplus este un operator binar asociativ, atunci $x_k = a_0 \oplus a_1 \oplus \dots \oplus a_k$.



Scurtcircuitarea (dublarea)

- Fie L o listă simplu înlățuită formată din n elemente.
- Fiecare element $k \in L$ are asociat un număr $valoare[k]$ și un pointer $pointer[k]$.
- Se presupune că fiecărui element $k \in L$ îi este asignată o unitate de procesare p_k .
- Scurtcircuitarea se referă la indirectarea prin intermediul pointerilor, pentru a accesa locații noi.



Algoritm paralel de scurtcircuitare

SCURTCIRCUITARE_PARALELA($L, valoare, \oplus$)

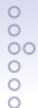
```
1  for  $i \leftarrow 1$  to  $\log n$ 
2  do for all  $k : k \in L$ 
3    do in parallel
4      if  $pointer[k] \neq pointer[pointer[k]]$ 
5        then  $valoare[k] \leftarrow valoare[k] \oplus valoare[pointer[k]];$ 
6               $pointer[k] \leftarrow pointer[pointer[k]];$ 
```



Complexitatea

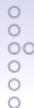
Teorema (7)

Complexitatea timp a algoritmului paralel de scurtcircuitare, implementat pe o masina CREW-PRAM, este $O(\log n)$.



Aplicații ale scurtcircuitării: Calcul ranguri

- Un exemplu tipic de utilizare a tehnicii scurtcircuitării îl constituie calcularea numerelor de ordine ale elementelor unei liste, față de sfârșitul acesteia.
- Pentru un element $k \in L$, $rang[k]$ va conține în final numărul i al elementor din lista L care se află în listă după elementul k , în ordinea dată de pointeri.



Algoritm paralel de calcul ranguri

CALCUL_RANGURI_PARALEL($L, rang, +$)

```
1 /* Initializare vectori pointer și rang */ for all k : k ∈ L
2 do in parallel
3     pointer[k] ← succesor(k) /* succesor(k) = elementul care urmează lui k în lista L */
4     if pointer[k] ≠ k
5         then rang[k] ← 1;
6         else rang[k] ← 0;
7 /* Aplicarea tehnicii scurtcircuitării */
8 SCURTCIRCUITARE_PARALELA( $L, rang, +$ )
```



Exemplu de execuție a algoritmului de calcul ranguri

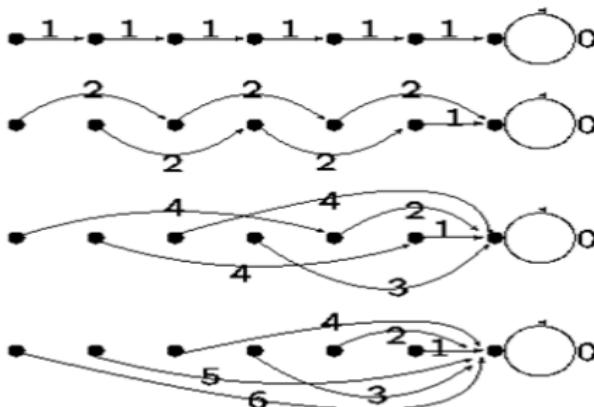
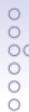


Figura 3: Exemplu de execuție a algoritmului de calcul ranguri



Comentarii

- Scurtcircuitarea nu se reduce la operarea cu elementele unei liste.
- Documentele web pot fi obiectul unor operații de scurtcircuitare paralelă, atunci când se dorește clusterizarea acestora.
- Rutarea poate folosi tehnica scurtcircuitării paralele.

Algoritmi paraleli și distribuiți

Sortare paralelă

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

Introducere	Algoritmul Muller-Preparata	Algoritmul Impar-Par (Odd-Even Sort)	Sortare bitonică	Sortare rapidă pe hipercub	Comentarii bibliografice
○	○		○○	○	
○	○		○○	○	
○	○		○	○○○	
○	○		○○○○	○	
○	○		○○○○○		
○	○		○○○○○○		
			○		

Cuprins

Introducere

Algoritmul Muller-Preparata

- Descriere
- Pseudocod
- Exemplu de execuție
- Corectitudinea
- Complexitatea
- Comentarii

Algoritmul Impar-Par (Odd-Even Sort)

- Descriere
- Pseudocod pentru algoritmul secvențial
- Exemplu de execuție
- Pseudocod pentru un lanț de unități de procesare
- Complexitatea
- Comentarii

Sortare bitonică

- Descriere
- Pseudocod
- Exemplu de sortare a unei secvențe bitone
- Corectitudinea
- Implementare
- Complexitatea
- Comentarii

Sortare rapidă pe hipercub

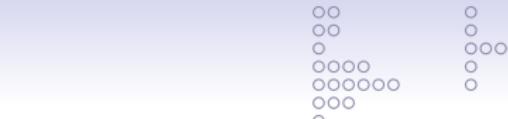
- Descriere
- Pseudocod
- Exemplu de execuție
- Complexitatea
- Comentarii

Comentarii bibliografice

Introducere	Algoritm Muller-Preparata	Algoritm Impar-Par (Odd-Even Sort)	Sortare bitonică	Sortare rapidă pe hipercub	Comentarii bibliografice
○	○		○○	○	
○	○		○○	○	
○	○		○	○○○	
○	○		○○○○	○	
○	○		○○○○○		
○	○		○○○○○○		
			○		

Introducere

- Există o vastă literatură de specialitate având ca subiect sortarea.
- Aceasta se explică prin faptul că sortarea apare ca subtask în soluțiile algoritmice ale multor probleme.
- Problema poate fi enunțată astfel:
Date fiind n elemente a_0, a_1, \dots, a_{n-1} , dintr-o mulțime U peste care este definită o relație de ordine totală " $<$ ", se dorește renumerotarea lor astfel încât $a_i < a_j$, $i, j \in \{0, 1, \dots, n-1\}$, $i < j$.
- Se presupune, pentru simplitate, că $a_i \neq a_j$, dacă $i \neq j$.



Algoritmul Muller-Preparata - descriere

- Autori: David E. Muller și Franco P. Preparata. Anul publicării: 1975
- Algoritmul este compus din trei faze:
 1. Determinarea pozițiilor relative pentru fiecare pereche $\{a_i, a_j\}, i, j = 0, 1, \dots, n - 1$:
 - Notații: $\text{pozitie_relativa}_{a_j}(a_i)$ și $\text{pozitie_relativa}_{a_i}(a_j)$ desemnează poziția lui a_i , respectiv a_j în secvența (a_i, a_j) sortată crescător.
 - Dacă $a_i < a_j$, atunci $\text{pozitie_relativa}_{a_j}(a_i) = 0$ și $\text{pozitie_relativa}_{a_i}(a_j) = 1$.
 - Dacă $a_i > a_j$, atunci $\text{pozitie_relativa}_{a_j}(a_i) = 1$ și $\text{pozitie_relativa}_{a_i}(a_j) = 0$.
 2. Calcularea pozitiilor finale ale elementelor $a_i, i = 0, 1, \dots, n - 1$.
 3. Plasarea elementelor a_j pe pozitiile finale.



Algoritmul Muller-Preparata - pseudocod

- **Notări:**

- $A[0..n-1]$ și $P[0..n-1]$ sunt două tablouri, fiecare de dimensiune n .
- $R[0..2n-2, 0..n-1]$ este un tablou bidimensional de mărime $(2n-1) \times n$; $R[j]$ desemnează coloana j .

- **Premise:**

- Datele de intrare sunt memorate în tabloul $A[0..n-1]$.
- Pozițiile relative vor fi memorate în tabloul R , în liniile $n-1, n, \dots, 2n-2$.
- Pozițiile finale vor fi reținute în tabloul P .

SORTARE_PARALELA_MULLER_PREPARATA(A, R, n)

```

1  for all  $i, j : 0 \leq i, j \leq n-1$ 
2  do in parallel /* calcularea pozițiilor relative */
3    if  $A[i] < A[j]$ 
4      then  $R[i+n-1, j] \leftarrow 1$ 
5      else  $R[i+n-1, j] \leftarrow 0$ 
6  for all  $j : 0 \leq j \leq n-1$ 
7  do in parallel /* calcularea pozițiilor finale */
8    /* Se calculează numărul elementelor care se află în fața elementului  $a_j$  */
9    COMPRIM_ITERATIV( $R[j], +$ )
10    $P[j] = R[0, j]$ 
11  for all  $j : 0 \leq j \leq n-1$ 
12  do in parallel /* plasarea pe pozițiile finale */
13    $A[P[j]] = A[j]$ 

```



Exemplu de execuție a algoritmului Muller-Preparata

Tabloul R

a_j	2	6	3	8
i	0	1	2	3
a_i	2	0	1	2
0	0	2	1	3
1	0	1	1	2
2	0	1	0	1
3	3	0	1	1
6	4	0	0	1
3	5	0	1	0
8	6	0	0	0

Figura 1 : Exemplu de execuție a algoritmului de sortare paralelă Muller-Preparata pentru secvența 2,6,3,8



Corectitudinea

Lema (1)

Pozițiile finale ale elementelor secvenței a_0, a_1, \dots, a_{n-1} respectă relația de ordine " $<$ ".

Demonstrație.

În urma calculării numărului elementelor care se află în fața unui element a_i (COMPRIM_ITERATIV ($R[j], +$)), se obține poziția finală a acestuia, în concordanță cu relația de ordine " $<$ ". □

Teorema (1)

Algoritmul Muller-Preparata sortează corect o secvență de elemente a_0, a_1, \dots, a_{n-1} dintr-o mulțime U peste care este definită o relație de ordine totală " $<$ ".

Demonstrație.

Consecință imediată a lemei 1. □



Complexitatea

Teorema (2)

Complexitatea timp a algoritmului de sortare paralelă Muller-Preparata, implementat pe o masina CREW-PRAM cu $O(\frac{n^2}{\log n})$ unități de procesare, este $O(\log n)$.

Demonstrație.

1. Determinarea pozițiilor relative pentru fiecare pereche $\{a_i, a_j\}, i, j = 0, 1, \dots, n-1$: dacă mașina CREW-PRAM este compusă din n^2 unități de procesare, timpul paralel este $O(1)$; dacă numărul unităților de procesare este $\lceil \frac{n^2}{\log n} \rceil$, timpul paralel este $O(\log n)$ (tehnica este aceeași cu cea de la comprimare).
2. Calcularea pozitiei finale ale elementelor $a_i, i = 0, 1, \dots, n-1$: pentru fiecare i , sunt necesare cel puțin $\lceil \frac{n}{\log n} \rceil$ unități de procesare, pentru a calcula poziția finală a elementului a_i în timpul paralel $O(\log n)$ (vezi complexitatea algoritmului paralel de comprimare). Rezultă un necesar de $n \lceil \frac{n}{\log n} \rceil$ unități de procesare pentru a calcula toate pozițiile finale în timpul paralel $O(\log n)$.
3. Plasarea elementelor a_j pe pozitiile corecte: cu n unități de procesare pentru se obține timpul paralel $O(1)$.



Comentarii

- Relativ la algoritmul secvențial, bazat pe metoda enumerării, care necesită $O(n^2)$ timp, eficiența algoritmului este $E = \frac{O(n^2)}{O(\frac{n^2}{\log n}) \log n} = O(1)$.
- Totusi, algoritmul nu este optimal, deoarece cel mai rapid algoritm secvențial are timpul de executie de $O(n \log n)$.



Algoritmul Impar-Par (Odd-Even Sort)

Sortare bitonică



Sortare rapidă pe hipercub

Comentarii bibliografice

Algoritmul Impar-Par (Odd-Even Sort) - descriere

- Este o versiune a algoritmului Bubble.
- Se desfășoară în faze.
 - În fazele impare sunt sortate perechile $\{a_i, a_{i+1}\}$ cu i par.
 - În fazele pare sunt sortate perechile $\{a_i, a_{i+1}\}$ cu i impar.
- Este paralelizabil.



Algoritmul Impar-Par secvențial - pseudocod

- Notării:* $A[0..n-1]$ este un tablouri de dimensiune n .
- Premise:* Datele de intrare sunt memorate în tabloul $A[0..n-1]$.

SORTARE_SECVENTIALA_IMPAR_PAR(A, n)

```

1  for fază ← 1 to n
2  do if fază este impara
3      then for i ← 0 to  $2\lfloor \frac{n}{2} \rfloor - 2$  step 2
4          do COMPARA_SI_INTERSCHIMBA( $i, i+1$ )
5  if fază este para
6      then for i ← 1 to  $2\lfloor \frac{n-1}{2} \rfloor - 1$  step 2
7          do COMPARA_SI_INTERSCHIMBA( $i, i+1$ )

```

COMPARA_SI_INTERSCHIMBA(i, j)

```

1  if  $a_i > a_j$ 
2  then temp ←  $a_i$ 
3       $a_i \leftarrow a_j$ 
4       $a_j \leftarrow temp$ 
5

```



Exemplu de execuție a algoritmului Impar-Par

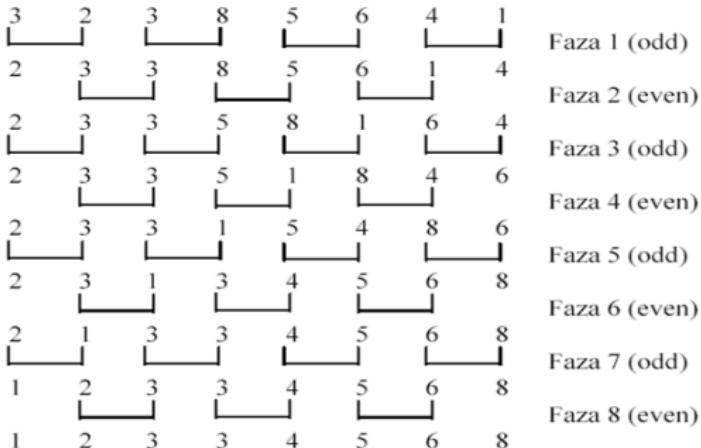
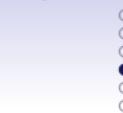


Figura 2 : Exemplu de execuție a algoritmului de sortare Impar-Par pentru $n = 8$



Algoritmul Impar-Par paralel - pseudocod pentru un lanț de unități de procesare

- Premise: Inițial, o unitate de procesare p_i memorează elementul a_i în registrul r .

SORTARE_PARALELA_IMPAR_PAR(p_i, r, n)

```

1  for faza  $\leftarrow 1$  to  $n$ 
2  do if faza este impara si  $0 \leq i \leq 2\lfloor \frac{n}{2} \rfloor - 1$ 
3      then if i este par
4          then trimit lui  $p_{i+1}$  valoarea memorata in registrul r
5              primeste de la  $p_{i+1}$  o valoare v
6               $r \leftarrow \min(r, v)$ 
7          else trimit lui  $p_{i-1}$  valoarea memorata in registrul r
8              primeste de la  $p_{i-1}$  o valoare v
9               $r \leftarrow \max(r, v)$ 
10     if faza este para si  $1 \leq i \leq 2\lfloor \frac{n-1}{2} \rfloor$ 
11        then if i este impar
12            then trimit lui  $p_{i+1}$  valoarea memorata in registrul r
13                primeste de la  $p_{i+1}$  o valoare v
14                 $r \leftarrow \min(r, v)$ 
15            else trimit lui  $p_{i-1}$  valoarea memorata in registrul r
16                primeste de la  $p_{i-1}$  o valoare v
17                 $r \leftarrow \max(r, v)$ 

```



Complexitatea

Teorema (3)

Complexitatea timp a algoritmului de sortare paralelă Impar-Par, implementat pe un lanț de n unități de procesare, este $O(n)$.

Demonstrație.

Timpul paralel pentru fiecare fază este $O(1)$. După n faze algoritmul se termină. □



Comentarii

- Algoritmul de sortare paralelă *Impar-Par* este optimal pentru arhitectură: Fiecare unitate de procesare este solicitată $O(n)$ timp
- Costul nu este optimal: (Numarul de unități de procesare) \times (timpul paralel) = $n \times n = O(n^2)$. Timpul pentru cel mai rapid algoritm secvențial este $O(n \log n)$.
- Algoritmul poate fi implementat și pe o mașină CREW-PRAM.

Exercițiu: Scrieți un pseudocod pentru sortarea Impar-Par pe o astfel de mașină.



Algoritmul lui Batcher de sortare bitonică - descriere

- Autor: Batcher; Anul publicării: 1968.
- Operația de bază este sortarea unei secvențe bitone.
- Esența problemei sortării unei secvențe bitone este transformarea sortării unei secvențe de bitone lungime n în sortarea a două secvențe bitone de dimensiune $\frac{n}{2}$.
- Pentru a sorta o secvență de n elemente, prin tehnica sortării unei secvențe bitone, trebuie să dispunem de o secvență bitonă formată din n elemente.
- Observații:
 - Două elemente formează o secvență bitonă.
 - Orice secvență nesortată este o concatenare de secvențe bitone de lungime 2.
- Ideea transformării unei secvențe oarecare în una bitonă: combinarea a două secvențe bitone de lungime $\frac{n}{2}$ pentru a obține o secvență bitonă de lungime n .
- Algoritmul este paralelizabil.



Secvențe bitone

- Secvența bitonă este o secvență de elemente $[a_0, a_1, \dots, a_{n-1}]$ pentru care
 - există i astfel încât $[a_0, a_1, \dots, a_i]$ este monoton crescătoare și $[a_{i+1}, \dots, a_{n-1}]$ este monoton descrescătoare sau
 - există o permutare circulară astfel încât să fie satisfacută condiția anterioară.
- Exemple:
 - $[1, 2, 4, 7, 6, 0]$; întâi crește și apoi descrește; $i = 3$.
 - $[8, 9, 2, 1, 0, 4]$; după o permutare circulară la stânga cu 4 poziții rezultă $[0, 4, 8, 9, 2, 1]$; $i = 3$.
- Fie $S = [a_0, a_1, \dots, a_{n-1}]$ o secvență bitonă,
 - $S_1 = [\min\{a_0, a_{\frac{n}{2}}\}, \min\{a_1, a_{\frac{n}{2}+1}\}, \dots, \min\{a_{\frac{n}{2}-1}, a_{n-1}\}]$ și
 - $S_2 = [\max\{a_0, a_{\frac{n}{2}}\}, \max\{a_1, a_{\frac{n}{2}+1}\}, \dots, \max\{a_{\frac{n}{2}-1}, a_{n-1}\}]$
- Secvențele S_1 și S_2 au proprietățile următoare:
 - Sunt bitone.
 - Fiecare element din S_1 este mai mic decit fiecare element din S_2 .



Algoritmul lui Batcher de sortare bitonică - pseudocod

- **Notății:**
 - $A[0..n-1]$ este un tablou unidimensional de dimensiune n .
 - (A, i, d) definește segmentul $A[i..i+d-1] = A[i], \dots, A[i+d-1]$.
 - s este un parametru binar care specifică ordinea crescătoare ($s = 0$) sau descrescătoare ($s = 1$) a cheilor de sortare.
 - **COMPARA_SI_SCHIMBA(x, y, s)** desemnează sortarea a două elemente x și y ordinea indicată de parametrul s .
- **Premise:** Inițial, $A[0..n-1]$ conține secvența de sortat.
- **Apel:** **SORTARE_BATCHER($A, 0, n, 0$)** sau **SORTARE_BATCHER($A, 0, n, 1$)**.

SORTARE_BATCHER(A, i, d, s)

```

1  if  $d = 2$ 
2    then  $(A[i], A[i+1]) \leftarrow \text{COMPARA_SI_SCHIMBA}(A[i], A[i+1], s)$ 
3    else /* sortare crescătoare a unei secvențe  $S$  de lungime  $\frac{d}{2}$  */
4      SORTARE_BATCHER( $A, i, \frac{d}{2}, 0$ )
5      /* sortare descrescătoare a secvenței  $S'$ , care urmează lui  $S$ , de lungime  $\frac{d}{2}$  */
6      SORTARE_BATCHER( $A, i + \frac{d}{2}, \frac{d}{2}, 1$ )
7      /* sortarea secvenței bitone  $SS'$ , de lungime  $d$  */
8      SORTARE_SECVENTA_BITONA( $A, i, d, s$ )

```



Sortarea unei secvențe bitone - pseudocod

- Premise: Inițial, segmentul $A[i..i + d - 1]$ conține o secvență bitonă de lungime d ;

SORTARE_SECVENTA_BITONA(A, i, d, s)

```

1  if  $d = 2$ 
2    then  $(A[i], A[i + 1]) \leftarrow \text{COMPARA\_SI\_SCHIMBA}(A[i], A[i + 1], s)$ 
3    else /* Construirea secvențelor bitone  $S_1$  și  $S_2$ , de lungime  $\frac{d}{2}$  */
4      for all  $j : 0 \leq j < \frac{d}{2}$ 
5      do in parallel
6           $(A[i + j], A[i + j + \frac{d}{2}]) \leftarrow \text{COMPARA\_SI\_SCHIMBA}(A[i + j], A[i + j + \frac{d}{2}], s)$ 
7          /* sortarea secvenței bitone  $S_1$ , de lungime  $\frac{d}{2}$  */
8          SORTARE_SECVENTA_BITONA( $A, i, \frac{d}{2}, s$ )
9          /* sortarea secvenței bitone  $S_2$ , de lungime  $\frac{d}{2}$  */
10         SORTARE_SECVENTA_BITONA( $A, i + \frac{d}{2}, \frac{d}{2}, s$ )

```



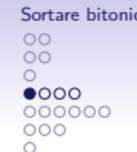
Exemplu de sortare a unei secvențe bitone

Original

sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 3 : Exemplu de sortare a unei secvențe bitone



Corectitudinea

Lema (2)

Dacă algoritmul lui Batcher sortează orice secvență de chei de sortare binare, atunci sortează orice secvență de chei de sortare numere reale oarecare.

Demonstrație.

Fie $f : R \rightarrow R$ o funcție monotonă. Astfel, $f(a_i) \leq f(a_j)$ dacă și numai dacă $a_i \leq a_j$. Evident, dacă algoritmul lui Batcher transformă secvența $[a_1, a_2, \dots, a_n]$ în secvența $[b_1, b_2, \dots, b_n]$, atunci va transforma secvența $[f(a_1), f(a_2), \dots, f(a_n)]$ în secvența $[f(b_1), f(b_2), \dots, f(b_n)]$. Astfel, dacă în secvența $[b_1, b_2, \dots, b_n]$ există un indice i pentru care $b_i > b_{i+1}$, atunci în secvența $[f(b_1), f(b_2), \dots, f(b_n)]$ vom avea $f(b_i) > f(b_{i+1})$.

Fie acum f o funcție monotonă definită astfel: $f(b_j) = \begin{cases} 0 & , \text{dacă } b_j < b_i \\ 1 & , \text{dacă } b_j \geq b_i \end{cases}$

În aceste condiții, secvența $[f(b_1), f(b_2), \dots, f(b_n)]$ va fi o secvență binară nesortată deoarece $f(b_i) = 1$ și $f(b_{i+1}) = 0$. Rezultă că algoritmul lui Batcher eșuează în sortarea secvenței binare $[f(a_1), f(a_2), \dots, f(a_n)]$. Deci, dacă algoritmul lui Batcher eșuează în sortarea unei secvențe de chei de sortare numere reale oarecare, atunci există o secvență binară care nu va fi sortată în urma aplicării algoritmului lui Batcher.





Corectitudinea -continuare

Teorema (4)

Algoritmul lui Batcher sorteaza cele n elemente ale secvenței memorate în tabloul A[0..n – 1], în ordinea crescătoare (s = 0) respectiv descrescătoare (s = 1) a cheilor de sortare.

Demonstrație.

Este suficient să demonstrăm corectitudinea procedurii **SORTARE_SECVENTA_BITONA** pentru cazul binar (Lema 2). Procedăm prin inducție după lungimea d a secvențelor procesate.

Dacă $d = 2$, evident procedura **SORTARE_SECVENTA_BITONA** transformă secvența inițială $S_{init} = A[i..i + d - 1]$ într-o secvență sortată S_{fin} .

Vom demonstra că procedura **SORTARE_SECVENTA_BITONA** transformă o secvență binară S_{init} de tipul $0^r1^t0^v$ sau $1^r0^t1^v$ ($r + t + v = d$) într-o secvență S_{fin} sortată în ordinea indicată de valoarea lui s , ($\forall d \geq 2$)

Pasul paralel 6 transformă secvența S_{init} într-o secvență S_{temp} conform figurilor 2 și 3.

Se observă că în toate cazurile, secvența rezultată S_{temp} , este formată din două sub-secvențe bitone S_{temp}^1 și S_{temp}^2 , fiecare de lungime $\frac{d}{2}$. O secvență este de tipul S_{init} iar cealaltă secvență conține numai cifre 0 sau numai cifre 1. Dacă $s = 0$, cheia maximă din S_{temp}^1 este mai mică sau egală cu cheia minimă din S_{temp}^2 . Dacă $s = 1$, cheia minimă din S_{temp}^1 este mai mare sau egală cu cheia maximă din S_{temp}^2 .

Conform ipotezei de inducție, procedura **SORTARE_SECVENTA_BITONA** transformă secvențele S_{temp}^1 și S_{temp}^2 în două secvențe S_{fin}^1 și S_{fin}^2 , sortate crescător ($d = 0$) sau descrescător ($s = 1$). Dacă $s = 0$, cheia maximă din S_{fin}^1 este mai mică sau egală cu cheia minimă din S_{fin}^2 deci secvența $S_{fin}^1 S_{fin}^2$ este crescătoare. Dacă $s = 1$, cheia minimă din S_{fin}^1 este mai mare sau egală cu cheia maximă din S_{fin}^2 deci secvența $S_{fin}^1 S_{fin}^2$ este descrescătoare.





Corectitudinea -continuare

s	$p + q (p, q \leq \frac{d}{2})$	Secvența inițială	Secvența rezultată
0	$\leq \frac{d}{2}$	$0^{\frac{d}{2}-p} 1^p 1^q 0^{\frac{d}{2}-q}$	$0^{\frac{d}{2}} 1^q 0^{\frac{d}{2}-(p+q)} 1^p$
0	$\leq \frac{d}{2}$	$0^{\frac{d}{2}} 0^k 1^p + q 0^m$	$0^{\frac{d}{2}} 0^k 1^{p+q} 0^m$
0	$\leq \frac{d}{2}$	$0^k 1^{p+q} 0^m 0^{\frac{d}{2}}$	$0^{\frac{d}{2}} 0^k 1^{p+q} 0^m$
0	$\leq \frac{d}{2}$	$1^p 0^{\frac{d}{2}-p} 0^{\frac{d}{2}-q} 1^q$	$0^{\frac{d}{2}} 1^p 0^{\frac{d}{2}-(p+q)} 1^q$
0	$\leq \frac{d}{2}$	$1^{\frac{d}{2}} 1^p 0^{\frac{d}{2}-(p+q)} 1^q$	$1^p 0^{\frac{d}{2}-(p+q)} 1^q 1^{\frac{d}{2}}$
0	$\leq \frac{d}{2}$	$1^p 0^{\frac{d}{2}-(p+q)} 1^q 1^{\frac{d}{2}}$	$1^p 0^{\frac{d}{2}-(p+q)} 1^q 1^{\frac{d}{2}}$
0	$> \frac{d}{2}$	$0^{\frac{d}{2}-p} 1^p 1^q 0^{\frac{d}{2}-q}$	$0^{\frac{d}{2}} - p_1(p+q) - \frac{d}{2} 0^{\frac{d}{2}-q} 1^{\frac{d}{2}}$
0	$> \frac{d}{2}$	$1^p 0^{\frac{d}{2}-p} 0^{\frac{d}{2}-q} 1^q$	$0^{\frac{d}{2}} - q_1(p+q) - \frac{d}{2} 0^{\frac{d}{2}-p} 1^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$0^{\frac{d}{2}} - p_1(p+q) 0^{\frac{d}{2}-q}$	$1^q 0^{\frac{d}{2}-(p+q)} 1^p 0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$0^{\frac{d}{2}} 0^k 1^{p+q} 0^m$	$0^k 1^{p+q} 0^m 0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$0^k 1^{p+q} 0^m 0^{\frac{d}{2}}$	$0^k 1^{p+q} 0^m 0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$1^p 0^{\frac{d}{2}-p} 0^{\frac{d}{2}-q} 1^q$	$1^p 0^{\frac{d}{2}-(p+q)} 1^q 0^{\frac{d}{2}}$
1	$\leq \frac{d}{2}$	$1^{\frac{d}{2}} 1^p 0^{\frac{d}{2}-(p+q)} 1^q$	$1^{\frac{d}{2}} 1^p 0^{\frac{d}{2}-(p+q)} 1^q$
1	$\leq \frac{d}{2}$	$1^p 0^{\frac{d}{2}-(p+q)} 1^q 1^{\frac{d}{2}}$	$1^{\frac{d}{2}} 1^p 0^{\frac{d}{2}-(p+q)} 1^q$
1	$> \frac{d}{2}$	$0^{\frac{d}{2}-p} 1^p 1^q 0^{\frac{d}{2}-q}$	$1^{\frac{d}{2}} 0^{\frac{d}{2}-p} 1(p+q) - \frac{d}{2} 0^{\frac{d}{2}-q}$
1	$> \frac{d}{2}$	$1^p 0^{\frac{d}{2}-p} 0^{\frac{d}{2}-q} 1^q$	$1^{\frac{d}{2}} 0^{\frac{d}{2}-q} 1(p+q) - \frac{d}{2} 0^{\frac{d}{2}-p}$

Figura 4 : Corectitudinea



Corectitudinea -continuare

s=0

0000111111	1110000000	→	0000000000	111011111111
0000000000	0011111000	→	0000000000	001111100000
0011111000	0000000000	→	0000000000	001111100000
11111110000	0000000011	→	0000000000	111111100111
1111111111	1100000111	→	1100000111	111111111111
1100000111	1111111111	→	1100000111	111111111111
0011111111	1111000000	→	0011000000	111111111111
11111111000	0000001111	→	0000001000	111111111111

S-1

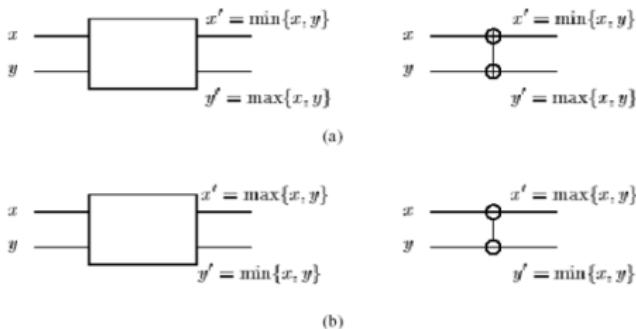
0000111111	1110000000	→	1110111111	0000000000
0000000000	0011111100	→	0011111100	0000000000
0011111100	0000000000	→	0011111100	0000000000
1111110000	00000000111	→	1111110111	0000000000
1111111111	1111000111	→	1111111111	1111000111
1111000111	1111111111	→	1111111111	1111000111
0111111111	1110000000	→	1111111111	0110000000
1111100000	0011111111	→	1111111111	0011100000

Figura 5 : Corectitudinea



Implementarea algoritmului lui Batcher pe rețele de sortare

- Dacă se derursivează algoritmul lui Batcher, se constată că sortarea unei secvențe de $n = 2^m$ elemente constă în m faze de sortare a unei secvențe bitone: $SSB_0, SSB_1, \dots, SSB_{m-1}$
- În faza SSB_k , $k \in \{0, 1, \dots, m-1\}$, se realizează sortarea secvențelor bitone $S_i^{2^d}$ formate din perechile de secvențe consecutive $S_i^d S_i'^d$ de lungime $d = 2^k$, S_i^d fiind sortată crescător și $S_i'^d$ descrescător.
- Secvențele $S_i^{2^d}$ cu numărul de ordine i par sunt sortate crescător. Cele cu i impar sunt sortate descrescător.



Copyright ©1994 Benjamin/Cummings Publishing Co.

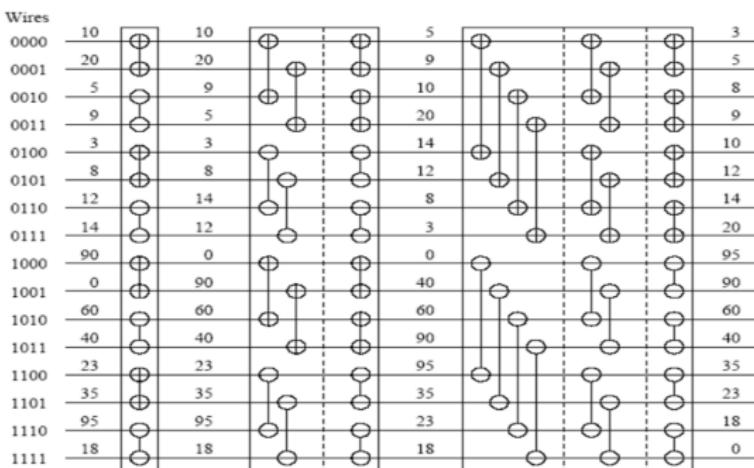
Figura 6 : Comparatori pentru sortarea a două elemente

○
○
○
○
○
○
○○
○
○
○
○
○
○○○
○○○
○○○○
○●○○○○
○○○○
○○○
○○
○
○○○
○
○

Rețea de comparatori care transformă o secvență oarecare în una bitonă

Fazele $0, 1, \dots, m-2$

- Intrare: o secvență oarecare; ieșire: o secvență bitonă.



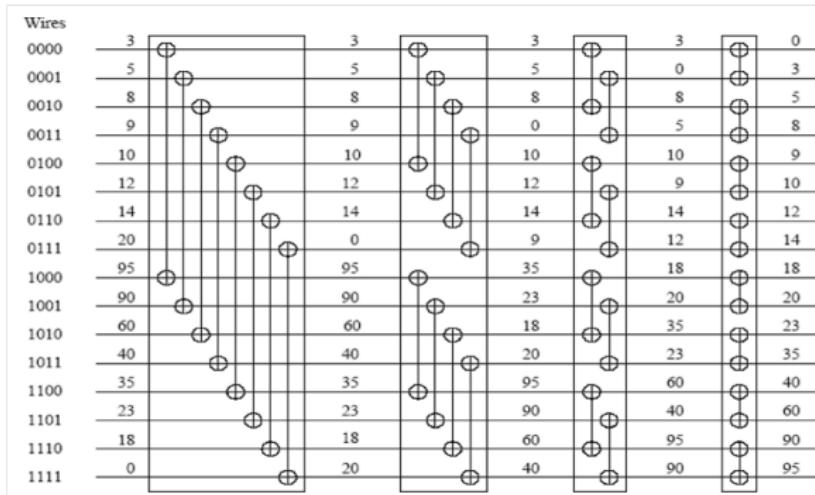
Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 7 : Rețea de comparatori care transformă o secvență oarecare în una bitonă (R_1); $n = 16$



Rețea de sortare a unei secvențe bitone - Faza $m-1$

- Intrare: o secvență bitonă; ieșire: o secvență sortată.



Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 8 : Rețea de sortare a unei secvențe bitone (R_2); $n = 16$

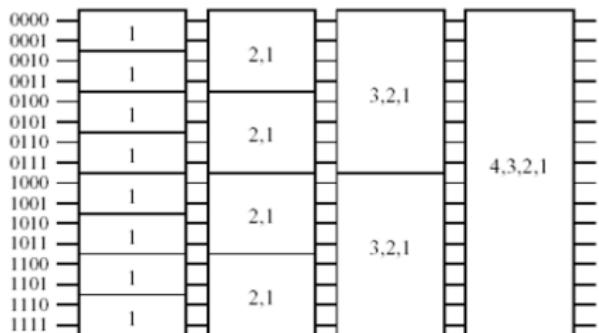


Implementarea algoritmului lui Batcher pe hipercub

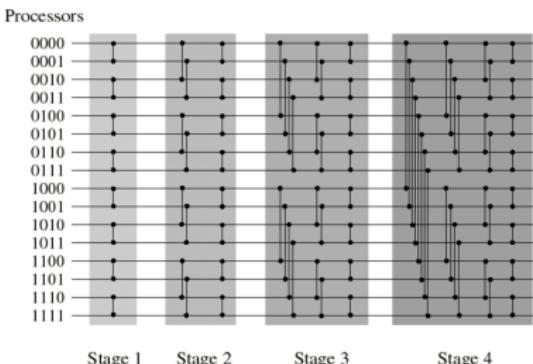
- Cubul binar multidimensional este o arhitectură ideală pentru implementarea algoritmului lui Batcher.
- Reamintim:
 - Dacă se derecursivează algoritmul lui Batcher, se constată că sortarea unei secvențe de $n = 2^m$ elemente constă în m faze de sortare a unei secvențe bitone:
 $SSB_0, SSB_1, \dots, SSB_{m-1}$
 - În faza SSB_k , $k \in \{0, 1, \dots, m-1\}$, se realizează sortarea secvențelor bitone $S_i^{2^d}$ formate din perechile de secvențe consecutive $S_i^d S_i'^d$ de lungime $d = 2^k$, S_i^d fiind sortată crescător și $S_i'^d$ descrescător.
 - Secvențele $S_i^{2^d}$ cu numărul de ordine i par sunt sortate crescător. Cele cu i impar sunt sortate descrescător.
- Execuția fazei SSB_k pe hipercub necesită utilizarea succesivă a dimensiunilor $D_k, D_{k-1}, \dots, D_1, D_0$.
- Planificarea utilizării dimensiunilor pentru o sortare completă poate fi reprezentată astfel:
 - $SSB_0 : D_0$
 - $SSB_1 : D_1, D_0$
 - $SSB_2 : D_2, D_1, D_0$
 - ...
 - $SSB_{m-1} : D_{m-1}, D_{m-2}, \dots, D_0$

○
○
○
○
○
○○
○
○
○
○
○○○
○○○
○○○○
○○○○●○
○○○○
○○○
○

Planificarea dimensiunilor = Fazele sortării prin rețele de sortare Batcher



(a)



(b)

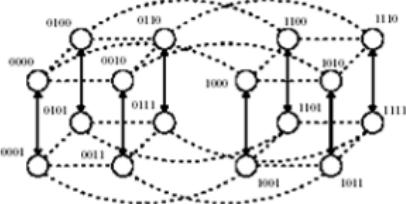
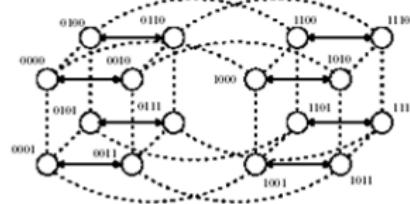
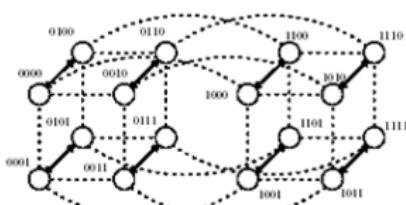
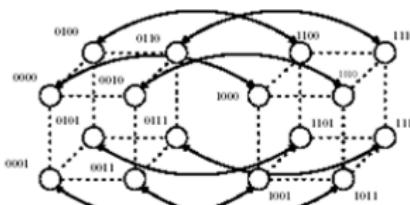
Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 9 : (a) Planificarea dimensiunilor (b)Fazele sortării prin rețele de sortare Batcher



Comunicarea pe hipercub în timpul ultimei faze a algoritmului lui Batcher

Fiecare linie continuă reprezintă o operație de interschimbare.



Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 10 : Comunicarea în timpul ultimei faze a algoritmului lui Batcher



Complexitatea implementării pe o mașină CREW-PRAM

Teorema (4)

Complexitatea timp a algoritmului lui Batcher de sortare paralelă, implementat pe mașină CREW-PRAM cu $O(n)$ unități de procesare este $O(\log^2 n)$. Eficiența algoritmului este $O(\frac{1}{\log n})$.

Demonstrație.

Timpul paralel pentru sortarea unei secvențe bitone de lungime $d < n$ este $O(\log d)$. După $\log n$ faze algoritmul se termină. Eficiența este $\frac{O(n \log n)}{n O(\log^2 n)} = O(\frac{1}{\log n})$. □



Complexitatea implementării pe rețele de sortare

Teorema (5)

Complexitatea timp a algoritmului lui Batcher de sortare paralelă, implementat pe o retea de sortare cu $O(n)$ intrări și ieșiri, este $O(\log^2 n)$.

Demonstratie.

Rețeaua (R_1, R_2) implementează algoritmul lui Batcher. Numărul de faze este $\log n$. Fazele sunt compuse din $\log d < \log n$ pași □



Complexitatea implementării pe hipercub

Teorema (6)

Complexitatea timp a algoritmului lui Batcher de sortare paralelă, implementat pe un hipercub cu $O(n)$ unități de procesare este $O(\log^2 n)$.

Demonstrație.

Sortarea Batcher pe hipercub este congruentă cu sortarea Batcher pe retele de sortare

○

○

○

○

○

○

○

○

○

○

○

○

○○

○○

○○

○○○○

○○○○○○

○○○○

●

○

○

○○○

○

○

Comentarii

- Algoritmul lui Batcher de sortare paralelă este din clasa timp $O(\log^2 n)$. Numărul de unități de procesare este din clasa $O(n)$. Comparativ cu algoritmii Muller-Preparata și Impar-Par are un cost mai bun.
- Totuși, costul nu este optimal: (Numarul de unități de procesare) x (timpul paralel) = $n \log^2 n = O(n \log^2 n)$. Timpul pentru cel mai rapid algoritm secvențial este $O(n \log n)$.

○

○

○

○

○

○

○

○

○

○

○○

○○

○

○○○○

○○○○○○

○○○

○

●

○

○○○

○

○

Sortare rapidă pe hipercub - descriere

- Să ne amintim că un hipercub cu m dimensiuni este format din două hipercuburi cu $m - 1$ dimensiuni.
- Numărul unităților de procesare, p , este mai mic decât numărul elementelor secvenței de sortat, n .
- Ideea este de a partitura secvența de sortat pe subcuburi și apoi de a repeta recursiv această operație.
- Selectarea pivotului este problema cheie.



Sortare rapidă pe hipercub - pseudocod

- **Notății:**

- $A[0..n-1]$ este un tablou de dimensiune $n = 2^m$.

- **Premise:**

- Datele de intrare sunt partitionate între unitățile de procesare.
- Fiecare unitate de proceare(nod al hipercubului) memorează o parte A_i din secventa de sortat, memorată inițial în tabloul $A[0..n-1]$.

SORTARE_RAPIDA_PE_HIPERCUB($A_i, m, p_i, pivot$)

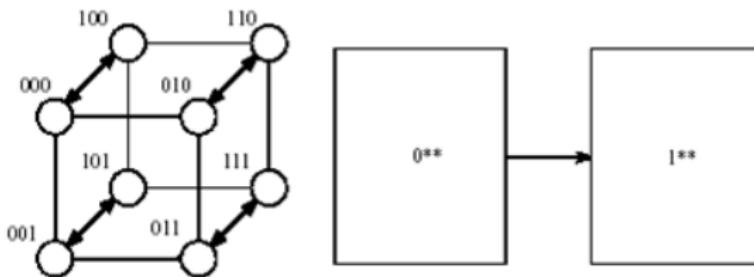
```

1    $i \leftarrow$  eticheta unitatii de procesare
2   for  $k \leftarrow m-1$  downto 0
3   do /* */
4      $x \leftarrow pivot$ 
5     partitioneaza  $A_i$  in  $A_{i_1}$  si  $A_{i_2}$  astfel incat  $A_{i_1} \leq x \leq A_{i_2}$ 
6     if al-k-lea bit este 0
7       then trimite  $A_{i_2}$  unitatii de procesare vecina pe dimensiunea k
8          $B_{i_1} \leftarrow$  subsecventa primita de la vecinul de pe dimensiunea k
9          $A_i \leftarrow A_{i_1} \cup B_{i_1}$ 
10    else trimite  $A_{i_1}$  unitatii de procesare vecina pe dimensiunea k
11       $B_{i_2} \leftarrow$  subsecventa primita de la vecinul de pe dimensiunea k
12       $A_i \leftarrow A_{i_2} \cup B_{i_2}$ 
13  Aplica lui  $A_i$  sortarea rapida sequentiala

```

○
○
○
○
○
○○
○
○
○
○
○○○
○○
○
○○○○
○○○○○○
○○○
○○
○
●○○
○
○

Exemplu de execuție a algoritmului de sortare rapidă pe hipercub ($m=3$)

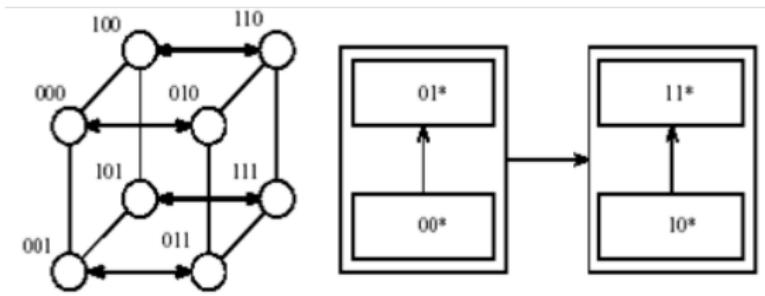


Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 11 : Partiționarea secvenței de sortat în două blocuri. Se utilizează dimensiunea a treia ($m - 1$).



Exemplu de execuție a algoritmului de sortare rapidă pe hipercub - continuare

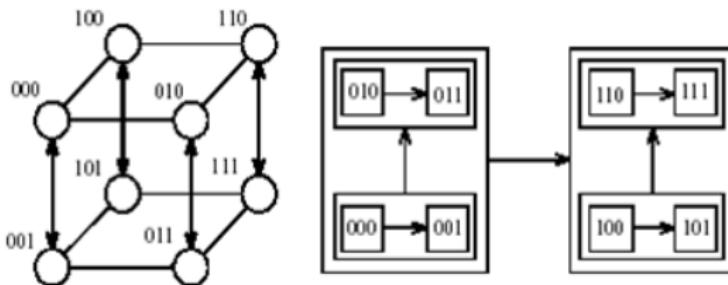


Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 12 : Partiționarea fiecărui bloc în două sub-blocuri. Se utilizează dimensiunea a două ($m - 2 = 1$).



Exemplu de execuție a algoritmului de sortare rapidă pe hipercub - continuare



Copyright ©1994 Benjamin/Cummings Publishing Co.

Figura 13 : Partiționarea sub-blocurilor. Se utilizează prima dimensiune ($m - 3 = 0$).



Complexitatea

Teorema (7)

Dacă pivotul este ales astfel încât să particioneze secvența în două subsecvențe de dimensiuni aproximativ egale, atunci

$$T_p = O\left(\frac{n}{p} \log \frac{n}{p}\right) + O\left(\frac{n}{p} \log p\right) + O(\log^2 p)$$



Comentarii

- Selectarea unui pivot care să particioneze secvența în două subsecvențe de dimensiuni aproximativ egale este dificilă.

Introducere	Algoritmul Muller-Preparata	Algoritmul Impar-Par (Odd-Even Sort)	Sortare bitonică	Sortare rapidă pe hipercub	Comentarii bibliografice
○	○		○○	○	
○	○		○○	○	
○	○		○	○○○	
○	○		○○○○	○	
○	○		○○○○○		
○	○		○○○○○○		
			○		

Comentarii bibliografice

- Capitolul sortare are la bază cartea

V. Kumar, A. Grama A. Gupta & G Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Addison Wesley, 2003
și ediția mai veche

V. Kumar, A. Grama A. Gupta & G Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, 1994

○
○○○○
○○
○
○

○
○○
○○○
○○○
○

Algoritmi paraleli și distribuiți **Calcul matriceal**

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

○
○○○○
○○
○
○

○
○○
○○○
○○○
○

Cuprins

Introducere

Transpusa unei matrice

- Descriere
- Pseudocod
- Implementare
- Exemplu de execuție
- Complexitatea

Inmulțirea de matrice pătratice

- Descriere
- Pseudocod
- Implementare
- Exemplu de execuție
- Complexitatea

Comentarii bibliografice

○
○○○○
○○
○
○

○
○○
○○○
○○○
○

Introducere

- Calculul matricial se pretează extrem de bine la procesare paralelă.
- Natura regulată a matricelor constituie un atu de neegalat pentru abordări specifice calculului paralel.
- Vom studia în această lecție problemele transpusei unei matrice și produsului de matrice.

●
○○○○○
○○
○
○

○
○○
○○○
○○○
○

Transpusa unei matrice - formularea problemei

- Dată fiind matricea pătratică $A_{n \times n} = (a_{i,j})_{i,j=0,1,\dots,n-1}$, se cere să se calculeze matricea $A_{n \times n}^T = (a_{i,j}^T)_{i,j=0,1,\dots,n-1}$, pentru care

$$a_{i,j}^T = a_{j,i}, \quad (\forall) i, j = 0, 1, \dots, n-1$$

- .
- Fără calcule.
- Doar mișcări de elemente.
- Timpul secvențial $T_s(n) = O(n^2)$.

○
●○○○○
○○
○
○

○
○○
○○○
○○○
○

Algoritmul secvențial standard - pseudocod

- *Notății:*

- $A[0..n-1, 0..n-1]$ este un tablou bidimensional, de dimensiune $n \times n$.

- *Premise:*

- Datele de intrare sunt memorate în tabloul A .
- Datele finale vor fi memorate în tabloul A .

TRANSPUNERE_MATRICE(A, n)

```

1  for  $i = 0$  to  $n - 1$ 
2  do for  $j = i + 1$  to  $n - 1$ 
3    do INTERSCHIMBA( $A[i, j], A[j, i]$ )

```

INTERSCHIMBA($A[i, j], A[j, i]$)

```

1  temp  $\leftarrow A[i, j]$ 
2   $A[i, j] \leftarrow A[j, i]$ 
3   $A[j, i] \leftarrow temp$ 

```



Exemplu de execuție a algoritmului de transpunere a unei matrice pe o plasă de unități de procesare

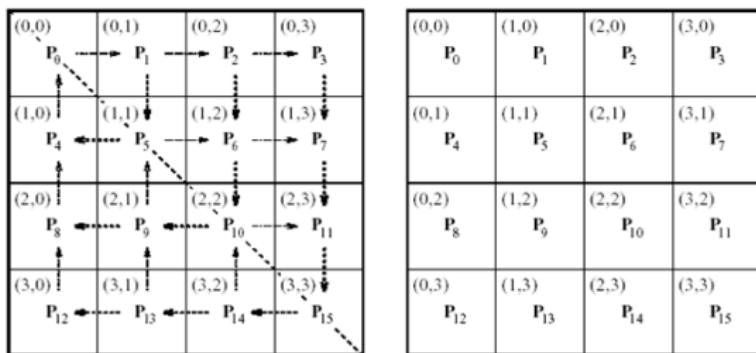


Figura 1: Exemplu de execuție a algoritmului de transpunere a unei matrice 4×4 pe o plasă de unități de procesare



Algoritmul recursiv - pseudocod

- *Notății:*

- $A[0..n-1, 0..n-1]$ este un tablou bidimensional, de dimensiune $n \times n$.

- *Premise:*

- $n = 2^m$.
- Datele de intrare sunt memorate în tabloul A .
- Matricea A este divizată în patru blocuri: *stânga-sus*, *dreapta-sus*, *stânga-jos*, *dreapta-jos*.
- Datele finale vor fi memorate în tabloul A .

TRANSPUNERE_MATRICE_RECURSIV($A[0..n-1, 0..n-1]$)

```

1  /* Interschimbarea blocurilor stânga-jos cu dreapta-sus */
2  for i =  $\frac{n}{2}$  to n-1
3  do for j = 0 to  $\frac{n}{2} - 1$ 
4    do INTERSCHIMBA( $A[i,j], A[i - \frac{n}{2}, j + \frac{n}{2}],$ )
5    TRANSPUNERE_MATRICE_RECURSIV( $A[0..\frac{n}{2}-1, 0..\frac{n}{2}-1]$ )
6    TRANSPUNERE_MATRICE_RECURSIV( $A[0..\frac{n}{2}-1, \frac{n}{2}..n-1]$ )
7    TRANSPUNERE_MATRICE_RECURSIV( $A[\frac{n}{2}..n-1, 0..\frac{n}{2}-1]$ )
8    TRANSPUNERE_MATRICE_RECURSIV( $A[\frac{n}{2}..n-1, \frac{n}{2}..n-1]$ )
  
```



Exemplu de execuție a algoritmului recursiv de transpunere a unei matrice

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(0,0)	(0,1)	(0,2)	(0,3)	(4,0)	(4,1)	(4,2)	(4,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,2)	(1,3)	(5,0)	(5,1)
(2,0)	(2,1)	(2,2)	(2,3)	(2,2)	(2,3)	(6,0)	(6,1)
(3,0)	(3,1)	(3,2)	(3,3)	(3,2)	(3,3)	(7,0)	(7,1)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Figura 2: Exemplu de execuție a algoritmului recursiv de transpunere a unei matrice 8x8

○
○○○●
○○
○
○

○
○○
○○○
○○○

Exemplu de execuție a algoritmului recursiv de transpunere a unei matrice - continuare

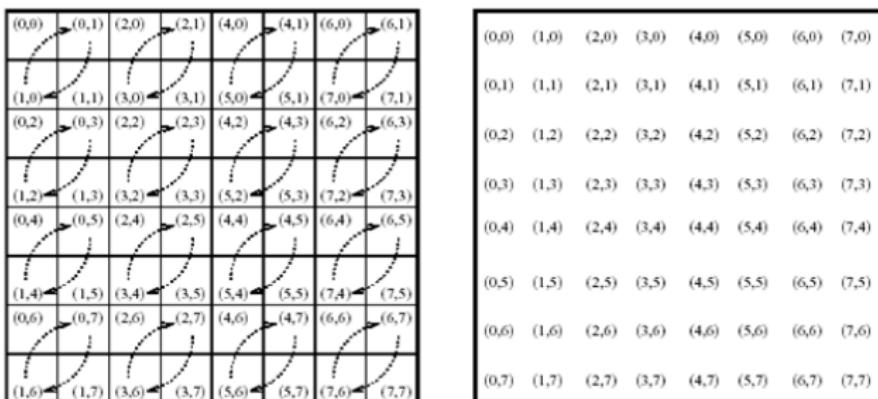


Figura 3: Exemplu de execuție a algoritmului recursiv de transpunere a unei matrice 8x8

○
○○○○
●○
○
○

○
○○
○○○
○○○
○

Implementare pe hipercub

- 1 Hipercubul este divizat în 4 subcuburi cu $\frac{p}{4}$ procesoare.
- 2 Sferturile de matrice (quadrantii) sunt mapate recursiv pe cele 4 subcuburi astfel:
 - cubul 00* conține sfertul *stânga-sus* al matricei A;
 - cubul 01* conține sfertul *dreapta-sus* sfert al matricei A;
 - cubul 10* conține sfertul *stânga-jos* al matricei A;
 - cubul 11* conține sfertul *dreapta-jos* al matricei A.
- 3 Se interschimbă blocurile *stânga-jos* cu *dreapta-sus*.
- 4 Se repetă pașii 1-3 în fiecare subcub.



Algoritmul recursiv implementat pe hipercub - pseudocod

- *Notății:*

- H este un hipercub cu m dimensiuni.
- i este indicele unității de procesare care executa algoritmul de transpunere a unei matrice A .
- M este blocul care urmează a fi transmis de o unitate de procesare către partener.

- *Premise:*

- Inițial, fiecare unitate de procesare p_i detine un bloc B_i din matricea A , conform cu regula de mapare, care urmează a fi transmis unei unități de procesare plasate într-un nod al hipercubului H .

TRANSPUNERE_MATRICE_PE_HIPERCUB(H, m, i, B_i)

```

1    $M \leftarrow B_i$ 
2   for  $k \leftarrow m-1$  downto 0 step 2
3   do partener  $\leftarrow i \text{ xor } 2^k$ 
4     if ( $\text{bit}_k(i) + \text{bit}_{k-1}(i) = 1$ )
5       then trimite  $M$  catre partener
6       else primește  $M$  de la partener
7     partener  $\leftarrow i \text{ xor } 2^{k-1}$ 
8     if ( $\text{bit}_k(i) + \text{bit}_{k-1}(i) \neq 1$ )
9       then trimite  $M$  catre partener
10      else primește  $M$  de la partener
11       $B_i \leftarrow M$ 
12  TRANSPUNERE_MATRICE( $B_i, \frac{n^2}{p}$ )
  
```



Exemplu de execuție a algoritmului de transpunere a unei matrice pe hipercub

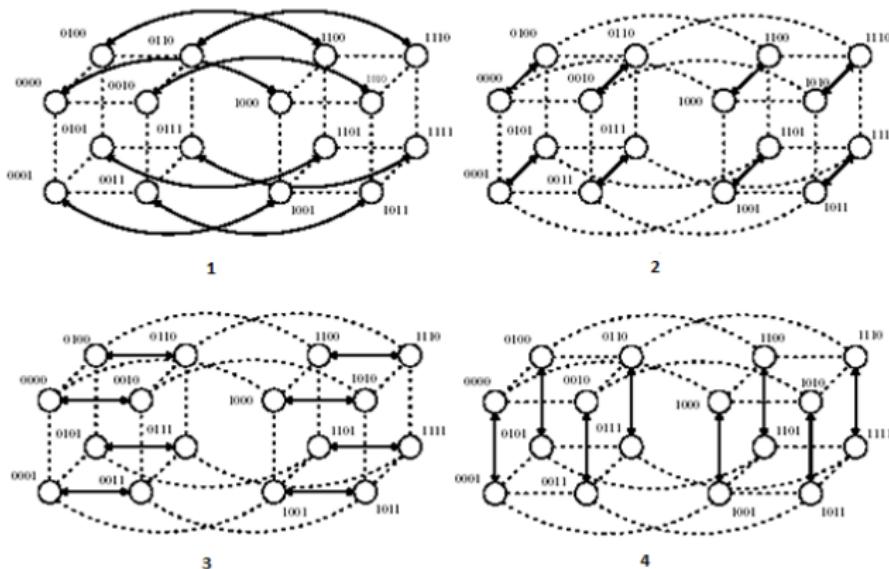


Figura 4: Exemplu de execuție a algoritmului de transpunere pe hipercub a unei matrice 4×4



Complexitatea implementării pe hipercub algoritmului recursiv de transpunere a unei matrice

Teorema (1)

Complexitatea timp a algoritmului recursiv de transpunere a unei matrice A_{nxn} , implementat pe un hipercub cu p unități de procesare este $O(\frac{n^2}{p} \log p)$. Eficiența algoritmului este $O(\frac{1}{\log n})$.

Demonstrație.

Divizarea recursivă se oprește când dimensiunea blocului este $\frac{n^2}{p}$. Numărul de pași de interschimbare de blocuri este $\log_4 p = \frac{\log_2 p}{2}$; fiecare pas de interschimbare se realizează pe două dintre dimensiunile hipercubului (două muchii). Timpul de transfer al unui bloc de dimensiune $\frac{n^2}{p}$ este $O(\frac{n^2}{p})$. Transpunerea locală necesită $O(\frac{n^2}{p})$ timp.

Rezultă că timpul total este $O(\frac{n^2}{p} \log p)$. Costul este $O(n^2 \log p)$ - nu este optimal. □

○
○○○○
○○
○
○

●
○○
○○○
○○○

Inmulțirea de matrice pătratice - formularea problemei

- Date fiind matricele pătratice $A_{n \times n} = (a_{i,j})_{i,j=0,1,\dots,n-1}$ și $B_{n \times n} = (b_{i,j})_{i,j=0,1,\dots,n-1}$, se cere să se calculeze matricea pătratică $C_{n \times n} = (c_{i,j})_{i,j=0,1,\dots,n-1}$, conform cu formula

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \times b_{k,j}, \quad (\forall) i,j = 0, 1, \dots, n-1$$

- Timpul secvențial $T_s(n) = O(n^3)$.

○
○○○○○
○○
○
○

○
●○
○○○
○○○
○

Algoritmul secvențial standard - pseudocod

- *Notății:*

- $A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$ și $C[0..n-1, 0..n-1]$ sunt tablouri bidimensionale, de dimensiune $n \times n$.

- *Premise:*

- Datele de intrare sunt memorate în tablourile A și B ;
 $A[i,j] = a_{i,j}$, $B[i,j] = b_{i,j}$, $i, j = 0, 1, \dots, n-1$.
- Datele finale vor fi memorate în tabloul C .

INMULTIRE_MATRICE(A, B, n)

```

1   for  $i = 0$  to  $n-1$ 
2     do for  $j = 0$  to  $n-1$ 
3       do  $C[i,j] \leftarrow 0$ 
4         for  $k = 0$  to  $n-1$ 
5           do  $C[i,j] \leftarrow C[i,j] + A[i,k] \times B[k,j]$ 
6   return  $C$ 
```

```

○
○○○○
○○
○
○

```

```

○
○●
○○○
○○○
○

```

Algoritmul înmulțirii de blocuri- pseudocod

- *Notății:*

- $A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$ și $C[0..n-1, 0..n-1]$ sunt tablouri bidimensionale, de dimensiune $n \times n$.
- Matricele A și B sunt divizate în blocuri $A_{i,k}$ și respectiv $B_{k,j}$, de dimensiuni $\frac{n}{q} \times \frac{n}{q}$.
- INMULTIRE_MATRICE($A_{i,k}, B_{k,j}, \frac{n}{q}$) semnifică înmulțirea blocului $A_{i,k}$ cu $B_{k,j}$ și returnarea rezultatului $A_{i,k} \times B_{k,j}$.

- *Premise:*

- Datele de intrare sunt memorate în tablourile A și B .
- Datele finale vor fi memorate în tabloul C .

INMULTIRE_MATRICE_BLOCURI(A, B, n, q)

```

1  for i = 0 to q - 1
2  do for j = 0 to q - 1
3    do  $C_{i,j} \leftarrow 0$ 
4      for k = 0 to q - 1
5        do  $C_{i,j} \leftarrow C_{i,j} + \text{INMULTIRE_MATRICE}(A_{i,k}, B_{k,j}, \frac{n}{q})$ 
6  return C

```

○
○○○○
○○
○
○

○
○○
●○○
○○○
○

Implementarea standard

- Arhitectura naturală este cea cu topologie de tip plasă, cu $p = q^2$ unități de procesare.
- Fiecare unitate de procesare $p_{i,j}$ dispune în memoria locală de blocul $A_{i,j}$ și un bloc $B_{i,j}$ și calculează $C_{i,j}$.
- Pentru a calcula $C_{i,j}$, unitatea de procesare $p_{i,j}$ are nevoie de $A_{i,k}$ și $B_{k,j}$, $k = 0, 1, \dots, q - 1$.
- Pentru fiecare k , blocurile $A_{i,k}$ și $B_{k,j}$ sunt obținute printr-o difuzie *toți-la-toți* pe linii și apoi pe coloane.

○
○○○○
○○
○
○

○
○○
○●○
○○○
○

Implementarea lui Cannon

- Similar cu implementarea standard, dar cu consum mai mic de memorie.
- Fiecare bloc este procesat la momente diferite, în locuri diferite. Pentru aceasta blocurile sunt deplasate ciclic.
- Sunt execuții $\sqrt{p} - 1$ pași de înmulțiri și adunari locale de blocuri, urmate de deplasări ciclice la stânga (în A) și în sus (în B).
- La final este executată o înmulțire și o adunare locală de blocuri.



Algoritmul lui Cannon - pseudocod

- **Notății:**

- P este o plasă formată din $qxq = \sqrt{px}\sqrt{p} = p$ unități de procesare.
- $p_{i,j}$ este unitatea de procesare care execută algoritmul de inmulțire de blocuri.
- M_1 este blocul care urmează a fi transmis de $p_{i,j}$ către $p_{i,j\oplus 1}$.
- M_2 este blocul care urmează a fi transmis de $p_{i,j}$ către $p_{i\oplus 1,j}$.
- M_3 este blocul calculat de $p_{i,j}$.
- \oplus și \ominus semnifică adunarea și scăderea modulo q .

- **Premise:** Inițial, fiecare $p_{i,j}$ deține în memoria locală blocurile $A_{i,j}$ și $B_{i,j}$.

- **Rezultate:** Fiecare unitate de procesare $p_{i,j}$ calculează blocul $C_{i,j}$.

INMULTIRE_MATRICE_CANNON($A_{i,j}, B_{i,j}, n, q, p_{i,j}$)

```

1   $M_1 \leftarrow A_{i,j}; M_2 \leftarrow B_{i,j}$ 
2  ALINIERE_INITIALA( $M_1, M_2, n, q, p_{i,j}$ )
3   $M_3 \leftarrow 0$ 
4  /*  $q-1$  pași de înmulțiri și adunari de blocuri */
5  for  $k \leftarrow 0$  to  $q-2$ 
6  do  $M_3 = M_3 + \text{INMULTIRE_MATRICE}(M_1, M_2, n/q)$ 
7    trimite  $M_1$  către  $p_{i,j\oplus 1}$ 
8    primește  $M_1$  de la  $p_{i,j\oplus 1}$ 
9    trimite  $M_2$  către  $p_{i\oplus 1,j}$ 
10   primește  $M_2$  de la  $p_{i\oplus 1,j}$ 
11    $C_{i,j} \leftarrow M_3 + \text{INMULTIRE_MATRICE}(M_1, M_2, n/q)$ 
12   return  $C_{i,j}$ 
  
```

ALINIERE_INITIALA($M_1, M_2, n, q, p_{i,j}$)

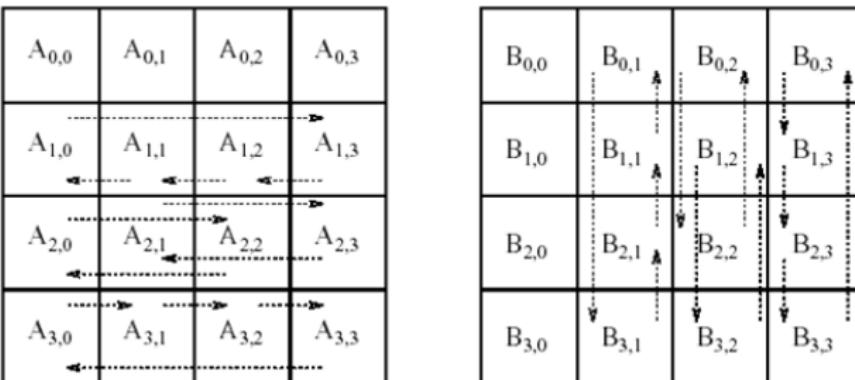
```

1  for  $k \leftarrow 1$  to  $i$ 
2  do trimite  $M_1$  către  $p_{i,j\oplus 1}$ 
3    primește  $M_1$  de la  $p_{i,j\oplus 1}$ 
4  for  $k \leftarrow 1$  to  $j$ 
5  do trimite  $M_2$  către  $p_{i\oplus 1,j}$ 
6    primește  $M_2$  de la  $p_{i\oplus 1,j}$ 
  
```

○
○○○○○
○○
○
○

○
○○
○○○
●○○
○

Exemplu de execuție a algoritmului lui Cannon



Alinierea inițială

Figura 5: Exemplu de execuție a algoritmului lui Cannon

```

○
○○○○
○○
○
○

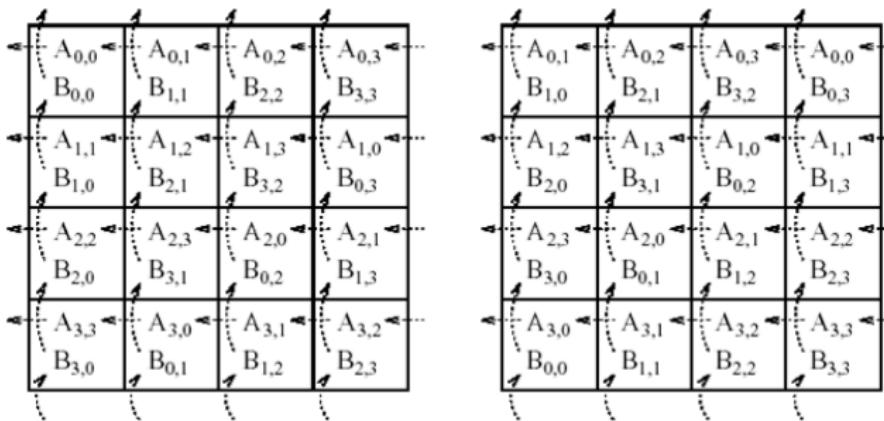
```

```

○
○○
○○○
○●○
○

```

Exemplu de execuție a algoritmului lui Cannon



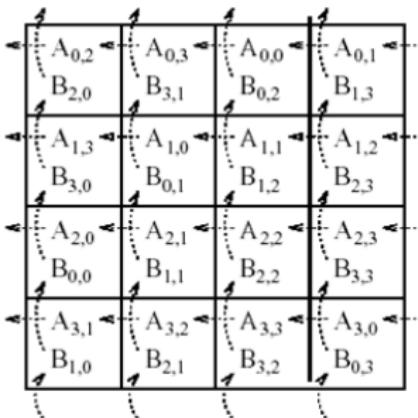
A și B după alinierea inițială Pozițiile blocurilor după prima deplasare

Figura 6: Exemplu de execuție a algoritmului lui Cannon

○
○○○○
○○
○
○

○
○○
○○○
○○●
○

Exemplu de execuție a algoritmului lui Cannon



A _{0,3} B _{3,0}	A _{0,0} B _{0,1}	A _{0,1} B _{1,2}	A _{0,2} B _{2,3}
A _{1,0} B _{0,0}	A _{1,1} B _{1,1}	A _{1,2} B _{2,2}	A _{1,3} B _{3,3}
A _{2,1} B _{1,0}	A _{2,2} B _{2,1}	A _{2,3} B _{3,2}	A _{2,0} B _{0,3}
A _{3,2} B _{2,0}	A _{3,3} B _{3,1}	A _{3,0} B _{0,2}	A _{3,1} B _{1,3}

Pozițiile blocurilor după a II-a deplasare

Pozițiile blocurilor după a III-a deplasare

Figura 7: Exemplu de execuție a algoritmului lui Cannon

○
○○○○
○○
○
○

○
○○
○○○
○○○
●

Complexitatea algoritmului lui Cannon

Teorema (2)

Complexitatea timp a algoritmului lui Cannon este $O(\frac{n^3}{p})$. Eficiența algoritmului este $O(1)$.

Demonstrație.

Numărul de transmisii și primiri de blocuri M_1 efectuate de $p_{i,j}$ este $i + q - 1 = i + \sqrt{p} - 1$.

Numărul de transmisii și primiri de blocuri M_2 efectuate de $p_{i,j}$ este tot $j + q - 1 = j + \sqrt{p} - 1$.

Se observă că $p_{q-1,q-1}$ efectuează cele mai multe transmisii și primiri de blocuri,

$q - 1 + q - 1 = 2(q - 1) = 2(\sqrt{p} - 1)$. Rezultă pentru deplasările de blocuri o complexitate timp de $O(\frac{n^2}{p}\sqrt{p})$. Numărul înmulțirilor și adunărilor efectuate de o unitate de procesare este

$O((\frac{n}{\sqrt{p}})^3\sqrt{p}) = O(\frac{n^3}{p})$; $O((\frac{n}{\sqrt{p}})^3)$ pentru fiecare înmulțire și adunare de blocuri. Costul este

$O(p)O(\frac{n^3}{p}) = O(n^3)$.



○
○○○○
○○
○
○

○
○○
○○○
○○○
○

Comentarii bibliografice

- Capitolul *Calcul matricial* are la bază cartea

V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003
și ediția mai veche

V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin-Cummings, 1994



Algoritmi paraleli și distribuiți

Sisteme de ecuații liniare

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași



Cuprins

[Introducere](#)
[Sistem de ecuații liniare](#)
[Descriere](#)
[Pseudocod](#)
[Implementare](#)
[Complexitatea](#)
[Comentarii bibliografice](#)



Introducere

- Sistemele de ecuații liniare pot fi rezolvate prin tehnici de calcul paralel datorită multiplelor operații independente una de alta.
- Vom studia în această lecție metoda lui Gauss de rezolvare a sistemelor de ecuații liniare.



Rezolvarea sistemelor de ecuații liniare prin metoda lui Gauss

- Se consideră un sistem format din n ecuații liniare cu n necunoscute:

$$a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0$$

$$a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} = b_1$$

.....

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

- Metoda lui Gauss constă în reducerea sistemului la forma triunghiulară

$$x_0 + a'_{0,1}x_1 + \cdots + a'_{0,n-1}x_{n-1} = b'_0$$

$$x_1 + \cdots + a'_{1,n-1}x_{n-1} = b'_1$$

.....

$$x_{n-1} = b'_{n-1}$$

- și rezolvarea ecuațiilor în ordine inversă (întâi ultima, apoi penultima și la sfârșit prima ecuație din sistem)



Algoritmul eliminării Gaussiene - varianta secvențială

- **Notății:**

- $A[0..n-1, 0..n-1]$ este un tablou bidimensional, de dimensiune $n \times n$.
- $B[0..n-1]$ este un tablou unidimensional, de dimensiune n .

- **Premise:**

- Coeficienții inițiali $a_{i,j}, i, j = 0, 1, \dots, n-1$ sunt memorati în tabloul A .
- Coeficienții $a'_{i,j}, i, j = 0, 1, \dots, n-1$ vor fi memorati tot în tabloul A .
- Tabloul B va memora coeficienții inițiali $b_i, i = 0, 1, \dots, n-1$ și soluția sistemului.

ELIMINARE_GAUSSIANA(A, B, n)

```

1  for k ← 0 to n-1 /* bucla exterioară */
2  do
3      for j ← k+1 to n-1
4          do  $A[k,j] \leftarrow \frac{A[k,j]}{A[k,k]}$  /* pasul de impărțire cu  $A[k,k]$  */
5               $B[k] \leftarrow \frac{B[k]}{A[k,k]}$ 
6               $A[k,k] \leftarrow 1$ 
7      for i ← k+1 to n-1
8          do for j ← k+1 to n-1
9              do  $A[i,j] \leftarrow A[i,j] - A[i,k] \times A[k,j]$  /* pasul de eliminare */
10              $B[i] \leftarrow B[i] - A[i,k] \times B[k]$ 
11              $A[i,k] \leftarrow 0$ 

```



Algoritmul eliminării Gaussiene - varianta paralelă

- *Notății:*

- $A[0..n-1, 0..n-1]$ este un tablou bidimensional, de dimensiune $n \times n$.
- $B[0..n-1]$ este un tablou unidimensional, de dimensiune n .

- *Premise:*

- Coeficienții inițiali $a_{i,j}, i, j = 0, 1, \dots, n-1$ sunt memorati în tabloul A .
- Coeficienții $a'_{i,j}, i, j = 0, 1, \dots, n-1$ vor fi memorati tot în tabloul A .
- Tabloul B va memora coeficienții inițiali $b_i, i = 0, 1, \dots, n-1$ și soluția sistemului.

ELIMINARE_GAUSSIANA_PARALELA(A, B, n)

```

1  for  $k \leftarrow 0$  to  $n-1$  /* bucla exterioară */
2  do
3      for  $j \leftarrow k+1$  to  $n-1$ 
4          do  $A[k,j] \leftarrow \frac{A[k,j]}{A[k,k]}$  /* pasul de împărțire cu  $A[k,k]$  */
5               $B[k] \leftarrow \frac{B[k]}{A[k,k]}$ 
6               $A[k,k] \leftarrow 1$ 
7          for  $i \leftarrow k+1$  to  $n-1$ 
8              do in parallel
9                  for  $j \leftarrow k+1$  to  $n-1$ 
10                 do  $A[i,j] \leftarrow A[i,j] - A[i,k] \times A[k,j]$  /* pasul de eliminare */
11                  $B[i] \leftarrow B[i] - A[i,k] \times B[k]$ 
12                  $A[i,k] \leftarrow 0$ 

```



Implementare pe un sistem cu n unități de procesare

- Se consideră un sistem format din n unități de procesare, p_0, p_1, \dots, p_{n-1} .
- Fiecare unitate de procesare $p_i, i \in \{0, 1, \dots, n-1\}$, dispune, în memoria locală, de coeficienții ecuației a-i-a (linia i a tabloului A plus $B[i]$).
- În prima fază, unitatea de procesare p_0 execută pasul de împărțirea cu $A[0,0]$:

$$A[0,j] = \frac{A[0,j]}{A[0,0]}, j = 0, \dots, n-1.$$
 Apoi trimite $A[0,j], j = 0, 1, \dots, n-1$ celorlalte unități de procesare, după care, fiecare unitate de procesare $p_i, i = 1, \dots, n-1$ execută pasul de eliminare: $A[i,j] = A[i,j] - A[i,0] \times A[0,j], i, j = 1, \dots, n-1$ și

$$A[i,0] = 0, i = 1, \dots, n-1.$$
- În faza a doua, p_1 preia rolul lui p_0 și execută împărțirea cu $A[1,1]$:

$$A[1,j] = \frac{A[1,j]}{A[1,1]}, j = 1, \dots, n-1.$$
 Apoi trimite $A[1,j], j = 1, \dots, n-1$ unitășilor de procesare p_2, \dots, p_{n-1} , după care, fiecare unitate de procesare $p_i, i = 2, \dots, n-1$ execută pasul de eliminare: $A[i,j] = A[i,j] - A[i,1] \times A[1,j], i, j = 2, \dots, n-1$ și

$$A[i,1] = 0, i = 2, \dots, n-1.$$
- În faza k , p_k execută împărțirea cu $A[k,k]$: $A[k,j] = \frac{A[k,j]}{A[k,k]}, j = k, \dots, n-1.$
 Apoi trimite $A[k,j], j = k, \dots, n-1$ unitășilor de procesare p_{k+1}, \dots, p_{n-1} , după care, fiecare unitate de procesare $p_i, i = k+1, \dots, n-1$ execută pasul de eliminare:

$$A[i,j] = A[i,j] - A[i,k] \times A[k,j], i, j = k+1, \dots, n-1$$
 și $A[i,k] = 0, i = k+1, \dots, n-1.$



Exemplu de implementare pe un sistem cu 8 unități de procesare

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Figura 1 : Pasul de împărțire cu $A[3,3]$:

$$A[3,j] = \frac{A[3,j]}{A[3,3]}, j = 4, \dots, 7, \text{ și } A[3,3] = 1$$



Exemplu de implementare pe un sistem cu 8 unități de procesare - continuare

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Figura 2 : Broadcast linia 3: $A[3,j], j = 4, \dots, 7$



Exemplu de implementare pe un sistem cu 8 unități de procesare - continuare

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Figura 3 : Pasul de eliminare:

$$A[i,j] = A[i,j] - A[i,3] \times A[3,j], i,j = 4, \dots, 7, \text{ și}$$

$$A[i,3] = 0, i = 4, \dots, 7$$



Complexitatea timp a implementării algoritmului eliminării Gausiene pe un lanț de n unități de procesare

Teorema (1)

Complexitatea timp a implementării algoritmului eliminării Gausiene pe un lanț de n unități de procesare este $\Theta(n^2)$.

Demonstrație.

Timpul consumat în iterația k este $\Theta(n - k - 1)$. Timpul total este $\sum_{k=0}^{n-1} \Theta(n - k - 1) = \Theta(n - 1) + \Theta(n - 2) + \dots + \Theta(0) = \Theta(n^2)$. □



Comentarii bibliografice

- Capitolul *Sisteme de ecuații liniare* are la bază cartea
V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003
și ediția mai veche
V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin-Cummings, 1994

○○○○○
○○○○○○○○○○
○
○○○○

○○○

Algoritmi paraleli și distribuiți

Alegerea liderului

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași



Alegerea liderului

Introducere

Alegerea liderului în rețele cu topologie de comunicare inel

Aspecte generale

Algoritm sincron de alegere a liderului în inele etichetate uniforme

Descriere

Pseudocod

Corectitudinea

Complexitatea

Algoritm sincron de alegere a liderului în inele etichetate neuniforme

Descriere

Algoritm asincron de alegere a liderului în inele etichetate uniforme

Descriere

Exemplu de execuție

Pseudocod

Corectitudinea

Complexitatea

Alegerea liderului în rețele cu topologie de comunicare graf oarecare

Algoritmul FloodMax sincron

Descriere

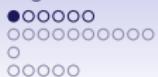
Pseudocod

Complexitatea



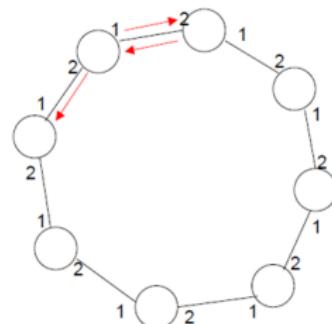
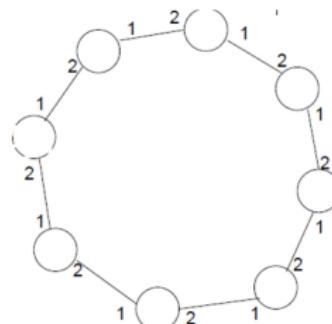
Introducere

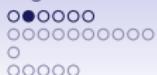
- *Problemă fundamentală: în sistemele distribuite este frecvent nevoie de un coordonator.*
- *Aplicații diverse:*
 - defecțiuni ale componentelor unui sistem distribuit (de exemplu: alegerea unui nou server coordonator pentru continuarea funcționării unui serviciu);
 - excludere mutuală în sisteme bazate pe comunicarea prin mesaje;
 - rețele mobile – alegerea unui alt lider când liderul cunoscut părăsește rețeaua.
- *Topologii de comunicare:*
 - inelul;
 - graful.



Alegerea liderului în rețele cu topologie de comunicație inel - aspecte generale

- Fiecare nod al inelului este asociat unei unități de procesare.
- Se stabilesc sensuri în inel. De exemplul stânga și dreapta unui nod este fixată din perspectiva așezării nodului cu "fața" la centru.
- Legăturile între noduri pot fi bidirectionale.

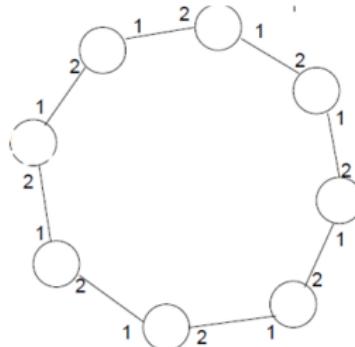




Aspecte generale - continuare

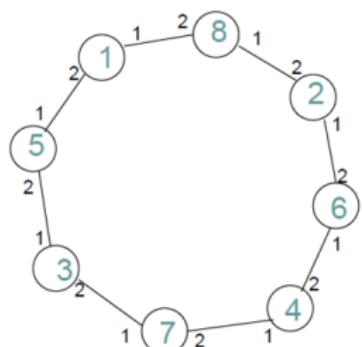
- Inel anonim:

- Unitățile de procesare nu au identificatori.
- Au mecanisme identice pentru schimbarea stărilor, adică funcțiile de tranziție de stare sunt identice.
- Sunt exprimate în termeni de stânga și dreapta.



- Inel etichetat:

- Fiecare unitate de procesare $p_i, i \in \{0, 1, \dots, n-1\}$ are asociat un identificator unic id ;
- Funcțiile de tranziție de stare sunt diferite.
- Sunt identificate prin id -urile lor.





Aspecte generale - continuare

- Inel uniform:
 - Unitățile de procesare au identificatori unici.
 - Numarul nodurilor inelului (n) nu este cunoscut de către unitățile de procesare.
 - Mecanismele de schimbare a stărilor nu depind de n (Funcțiile de tranziție de stare nu au variabila n ca parametru).
- Inel neuniform:
 - Unitățile de procesare au identificatori unici.
 - Numarul nodurilor inelului (n) este cunoscut de către unitățile de procesare.
 - Mecanismele de schimbare a stărilor depind de n (Funcțiile de tranziție de stare au variabila n ca parametru).



Aspecte generale - continuare

- Algoritmii de alegere a liderului depind de:
 - Tipul inelului: anonim sau etichetat;
 - Numărul nodurilor inelului: cunoscut sau necunoscut;
 - Tipul de ceas: global (execuție sincronă) sau local (execuție asincronă).



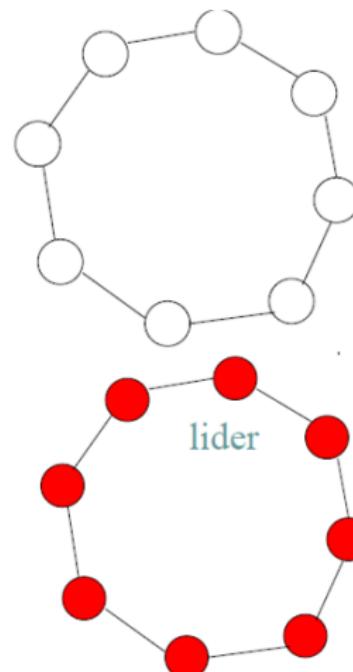
Alegerea liderului în inele anonime

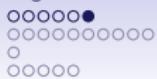
- Inele anonime sincrone:

- Ceasul este global.
- Unitățile de procesare execută același program.
- La un moment dat, unitățile de procesare efectuează aceeași operație (execută aceeași instrucțiune)
- Unitățile de procesare au aceeași evoluție, deoarece sunt identice.
- La final, toate unitățile de procesare sunt lideri.

- Inele anonime asincrone:

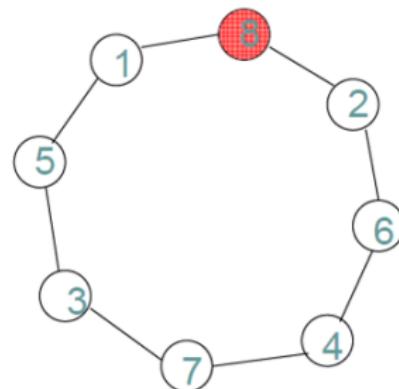
- Ceasul este local.
- Unitățile de procesare au aceeași evoluție, deoarece sunt identice.
- La final, toate unitățile de procesare sunt lideri.





Alegerea liderului în inele etichetate

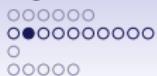
- Unitatea de procesare cu cel mai mare/mic identificator (id) este aleasă lider





Algoritm sincron de alegere a liderului în rețele etichetate uniforme - descriere

- Startul nu este sincronizat. Aceasta înseamnă că unitățile de procesare se "trezesc" spontan sau după primirea unui mesaj, adică încep execuția programului de alegere a liderului în momente diferite.
- La trezire, dacă nu a primit nici un mesaj, unitatea de procesare devine *participant*; dacă a primit un mesaj, devine *releu*.
- Atunci când o unitate de procesare cu rol de *participant* primește un mesaj de la p_i , dacă id -ul primit este mai mare decât identificatorul cel mai mic cunoscut (inclusiv propriul identificator), îl ignoră; altfel, îl reține $2^{id} - 1$ runde.
- Atunci când o unitate de procesare cu rol de *releu* primește un mesaj de la p_i , dacă id -ul primit este mai mare decât identificatorul cel mai mic cunoscut (fără propriul identificator), îl ignoră; altfel, îl retransmite imediat.



Algoritm sincron de alegere a liderului în inele etichetate uniforme

Pseudocod pentru unitatea de procesare p_i :

ASAL-IEU($S_i, R_i, W_i, n, stare_i, statut_i, id_i, p_i$)

```

1   while true
2   do if  $stare_i = adormit$ 
3       then if  $R_i = \emptyset$ 
4           then  $stare_i \leftarrow participant$ 
5                $min \leftarrow id_i$ 
6               memoreaza ( $id_i, 1$ ) in  $S_i$ 
7       else  $stare_i \leftarrow releu$ 
8            $min \leftarrow \infty$ 
9   for each  $(m, h)$  in  $R_i$ 
10  do if  $h \neq 3$ 
11      then if  $m < min$ 
12          then  $min \leftarrow m$ 
13          if  $stare_i = releu$  and  $h = 1$ 
14              then memoreaza ( $m, 1$ ) in  $S_i$ 
15              else memoreaza ( $m, 2$ ) impreuna cu numarul runde in  $W_i$ 
16          else if  $m = id$ 
17              then  $statut_i \leftarrow lider$ 
18                  memoreaza ( $m, 3$ ) in  $S_i$ 
19          else if  $statut_i = lider$ 
20              then break /* termină execuția algoritmului */
21              else  $statut_i \leftarrow non\_lider$ 
22                  memoreaza ( $m, 3$ ) in  $S_i$ 
23  for each  $(m, 2)$  in  $W_i$ 
24  do if  $(m, 2)$  a fost primit in urma cu  $2^m - 1$  runde
25      then sterge  $(m, 2)$  din  $W_i$  si memoreaza in  $S_i$ 
26      trimite  $S_i$  spre stanga
27      if  $statut_i = non\_lider$ 
28          then break /* termină execuția algoritmului */

```

- Notății:**

- R_i memorează mesajele recepționate de unitatea de procesare p_i .
- S_i memorează mesajele care urmează a fi trimise de către unitatea de procesare p_i .
- W_i memorează mesajele întârziate (în aşteptare).
- $stare_i$ reprezintă starea curentă a unității de procesare p_i (*adormit*, *participant* sau *releu*).
- $statut_i$ memorează starea finală a unității de procesare p_i (*lider* sau *non_lider*).
- h reprezintă faza în care se află mesajul m .

- Premise:**

- Inițial $S_i = R_i = W_i = \emptyset$ și $stare_i = adormit$, $statut_i = nefedinit$.



Corectitudinea

Teorema (1)

Doar unitatea de procesare cu cel mai mic id primește înapoi propriul id.

Demonstrație.

Fie p_i unitatea de procesare participantă cu cel mai mic identificator (id_i). Cel puțin o unitate de procesare are statut de participant. Evident, nici o unitate de procesare nu poate "înghită" (aruncă la coș, fără a-l transmite mai departe) mesajul id_i . Mai mult, mesajul id_i este întârziat, în fiecare unitate de procesare, cel mult 2^{id_i} runde. Să presupunem că există o unitate de procesare p_j , $j \neq i$, care primește înapoi propriul identificator, id_j . Rezulă că id_j trece prin toate unitățile de procesare din inel, inclusiv prin p_i . Dar $id_i < id_j$ și p_i este o unitate de procesare participantă, care nu va retransmite id_j . Contradicție. □



Complexitatea

Mesajele care circulă prin inel pot fi clasificate în trei categorii:

1. mesaje în faza I ($h = 1$);
2. mesaje în faza II ($h = 2$) trimise înainte de intrarea mesajului liderului în faza II;
3. mesaje în faza II ($h = 2$) trimise după intrarea mesajului liderului în faza II.
4. mesaje în faza III ($h = 3$) trimise după ce a fost decis liderul.

Lema (1)

Numărul de mesaje din prima categorie este cel mult n .

Demonstrație.

Este suficient să demonstrăm că fiecare unitate de procesare retransmite cel mult un mesaj din prima categorie. Să presupunem că există o unitate de procesare p_i care retransmite două mesaje din prima categorie, id_j primit de la p_j și id_k de la p_k . Fără a restrângе generalitatea, presupunem că p_j este mai aproape de p_i decât p_k . Dacă id_k ajunge în p_j după ce p_j se "trezește" și trimite p_j , id_k trece în faza II și va fi întârziat în fiecare unitate de procesare participantă 2^{id_k} runde; altfel p_j nu participă și nu trimite identificatorul id_j . Prin urmare, fie id_k ajunge în p_i în faza II, fie id_j nu este trimis de p_j și prin urmare nu este primit de p_i . Contradicție. □



Complexitatea - continuare

Lema (2)

Fie r prima rundă în care se "trezește" măcar o unitate de procesare și începe execuția algoritmului. Fie p_i una dintre acestea. Dacă unitatea de procesare p_j este la distanță d față de p_i , atunci p_j primește un mesaj din prima categorie cel mult în runda $r + d$.

Demonstrație.

Inducție după d . Pentru $d = 1$ este evident că vecinul lui p_i primește mesajul id (din categoria 1) în runda $r + 1$. Să presupunem că unitatea de procesare p_k situată la distanță $d - 1$ față de p_i primește un mesaj din prima categorie, cel mult în runda $r + d - 1$. Dacă aceasta este "trezită", a trimis deja la vecinul p_j un mesaj din prima categorie; altfel, înseamnă că p_k are statut de *releu* și va retransmite în runda $r + d$ către p_j mesajul de categoria 1, primit anterior. \square



Complexitatea - continuare

Lema (3)

Numărul de mesaje din categoria a doua este cel mult n .

Demonstrație.

Din Lema 1 rezultă că fiecare unitate de procesare retransmite cel mult un mesaj din prima categorie, adică pe o muchie este transmis cel mult un mesaj din prima categorie. Din Lema 2 rezultă că în runda $r + n$ pe fiecare muchie a fost trimis un mesaj din categoria 1. Rezultă că după runda $r + n$ nu va mai fi transmis nici un mesaj din categoria 1. Din Lema 2 rezultă că mesajul liderului intră în faza a II-a după cel mult $r + n$ runde, adică nu mai târziu de a n -a rundă care urmează trezirii primei unități de procesare. Rezultă că mesajele din categoria 2 circulă cel mult n runde. Un mesaj m aflat în faza a II-a este întârziat $2^m - 1$ runde, după care este retransmis. Prin urmare, un mesaj m circulă cu statutul "categoria 2" cel mult de $\frac{n}{2^m}$ ori. Deoarece mesajul care conține cel mai mic identificator circulă de cele mai multe ori, rezultă că numarul maxim de mesaje se obține atunci când unitățile de procesare participante au identificatorii $0, 1, \dots, n - 1$. Dacă ținem cont că mesajul liderului nu poate face parte din categoria 2, rezultă că numărul mesajelor din categoria 2 este $\sum_{i=1}^{n-1} \frac{n}{2^i} \leq n$





Complexitatea - continuare

Lema (4)

Nici un mesaj nu este retransmis după ce liderul p_i primește propriul identificator id_i .

Demonstrație.

Toate mesajele care urmează lui id_i sunt "înghițite".





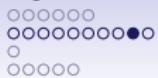
Complexitatea - continuare

Lema (5)

Numărul de mesaje din categoria a treia este cel mult $2n$.

Demonstrație.

Fie p_i liderul și p_j o unitate de procesare cu statut de *participant*. Din Teorema 1 rezultă că $id_i < id_j$. Din Lema 4 rezultă că nici un mesaj nu este retransmis după ce p_i primește înapoi propriul identificator id_i . Deoarece id_i este întârziat cel mult 2^{id_i} runde în fiecare unitate de procesare, sunt necesare cel mult $n2^{id_i}$ runde pentru ca p_i să primească înapoi propriul identificator id_i . Așadar, mesajele din categoria a treia sunt transmise de-a lungul a cel mult $n2^{id_i}$ runde. În timpul acestor runde, id_j este retransmis cel mult de $\frac{1}{2^{id_j}} n2^{id_i} = \frac{n}{2^{id_j - id_i}}$ ori. Astfel, numărul mesajelor din categoria 3 este cel mult $\sum_{j=0}^{n-1} \frac{n}{2^{id_j - id_i}}$. Pe baza faptului că numărul maxim de mesaje se obține atunci când unitățile de procesare participante au identificatorii $0, 1, \dots, n-1$ (Lema 3) se deduce că $\sum_{j=0}^{n-1} \frac{n}{2^{id_j - id_i}} = \sum_{k=0}^{n-1} \frac{n}{2^k} < 2n$. □



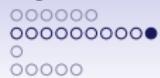
Complexitatea - continuare

Lema (6)

Numărul de mesaje din categoria a patra este n .

Demonstrație.

Fiecare unitate de procesare transmite mai departe mesajul liderului ($h = 3$), prin care acesta se declară câștigător. □



Complexitatea - continuare

Teorema (2)

Numărul de mesaje circulate prin algoritmul sincron de alegere a liderului în inele etichetate uniforme este cel mult $5n$.

Demonstrație.

Consecință imediată a lemelor 1,3,5 și 6. □



Algoritm sincron de alegere a liderului în inele etichetate neuniforme - descriere

- Algoritmul se desfășoară în faze. Fiecare fază este compusă din n runde.
- Startul este sincronizat. Aceasta înseamnă că toate unitățile de procesare se "trezesc" simultan, adică încep execuția programului de alegerea a liderului în același moment.
- Unitatea de procesare cu cel mai mic identificator este aleasă lider.
- În fiecare rundă, fiecare unitate de procesare efectuează următoarele operații:
 - analizează mesajele primite pe canalele din stânga și din dreapta;
 - își schimbă starea în funcție de starea curentă și de mesajele primite; dacă nu a primit niciun mesaj, își schimbă starea doar în funcție de starea curentă;
 - transmite mesaje la vecini, dacă are ceva de transmis.
- În faza i , dacă nimeni nu este ales, atunci unitatea de procesare cu id -ul i se autodeclară lider și transmite la vecini un mesaj prin care se declară lider; celelalte unități de procesare retransmitem mesajul primit.

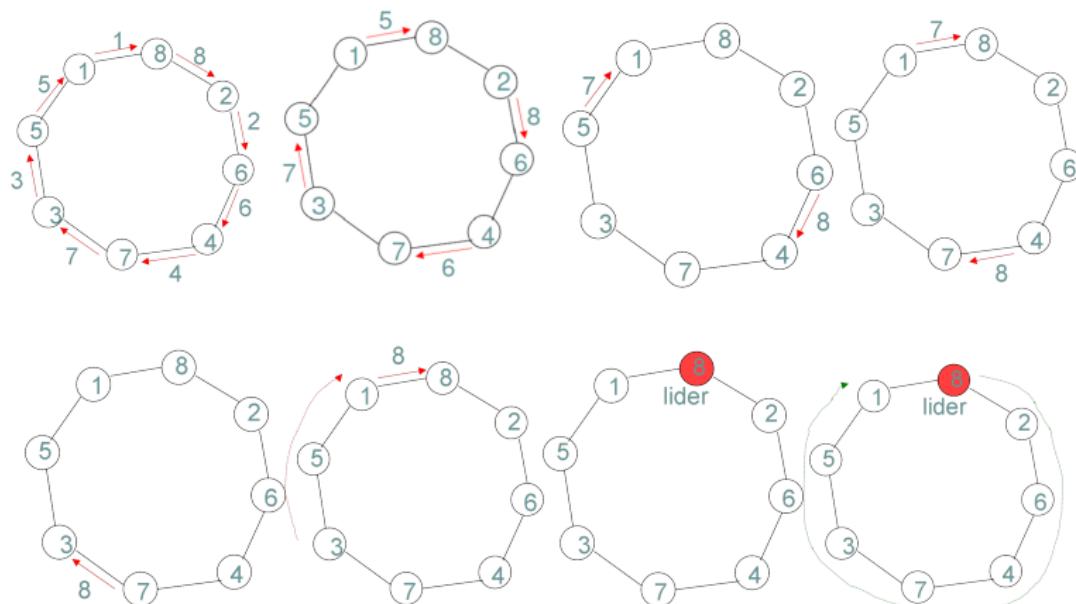


Algoritmul asincron LCR (LeLann, Chang si Roberts) - descriere

- Se consideră că unitățile de procesare "stau cu fața" spre centrul inelului.
- Fiecare unitate de procesare p_i transmite identificatorul său (id_i) vecinului din stânga.
- Apoi, așteaptă răspuns de la vecinul din dreapta
- Dacă identificatorul recepționat este mai mare decât propriul identificator, transmite identificatorul primit spre stânga.
- Dacă identificatorul primit e mai mic decat propriul identificator, ignoră ("inghită") mesajul.
- Dacă identificatorul recepționat este propriul identificator, unitatea respectivă se declară *lider* și transmite în inel un mesaj de terminare.
- Dacă o unitate de procesare primește mesaj de terminare, termina ca *non-lider*.



Algoritmul LCR - exemplu de execuție





Algoritmul LCR

Pseudocod pentru unitatea de procesare p_i :

- Notări:

- id_i reprezintă identificatorul unității de procesare p_i .
- $stare_i$ memorează starea curentă a unității de procesare p_i (*adormit*, *trezit*).
- $statut_i$ memorează starea finală a unității de procesare p_i (*lider* sau *non_lider*).

- Premise:

- Inițial $stare_i = \text{adormit}$ și $statut_i = \text{necunoscut}$.

```

LCR.LE( $p_i, stare_i, statut_i$ )
1    $stare_i \leftarrow \text{trezit}$ 
2   trimite <"id_nou",  $id_i$ > spre stanga
3   while true
4     do switch
5       case primește <"id_nou",  $id$ > dinspre dreapta :
6         if  $id = id_i$ 
7           then  $statut_i \leftarrow \text{lider}$ 
8             trimite <"lider",  $id$ > spre stanga
9         if  $id > id_i$ 
10          then trimite <"id_nou",  $id$ > spre stanga
11       case primește <"lider",  $id$ > dinspre dreapta :
12         if  $id \neq id_i$ 
13           then  $statut_i \leftarrow \text{non_lider}$ 
14             trimite <"lider",  $id$ > spre stanga
15         break /* termină execuția algoritmului */
    
```



Corectitudinea

Teorema (3)

Doar unitatea de procesare cu cel mai mare id primește primește înapoi propriul id.

Demonstrație.

Doar mesajul unității de procesare cu cel mai mare *id* nu va fi "înghițit" (aruncat la coș). □



Complexitatea

Teorema (4)

Numărul de mesaje care circulă prin inel este cel mult n^2 .

Demonstrație.

Cazul cel mai nefavorabil este acela în care unitățile de procesare au identificatori din mulțimea $\{0, 1, \dots, n-1\}$ și sunt plasate pe inel în ordinea $n-1, n-2, \dots, 0$. În această situație, mesajul $id_i = i$ al unității de procesare p_i este retransmis exact de $i+1$ ori. Astfel, numărul mesajelor care circulă prin inel (inclusiv mesajul de terminare) este $n + \sum_{i=0}^{n-1} (i+1) = O(n^2)$. □



Algoritmul FloodMax sincron - descriere

- Graful $G = (V, E)$ este conex. Unitățile de procesare cunosc diametrul $= d$.
- Fiecare unitate de procesare conține o înregistrare a identificatorului maxim cunoscut; inițial acesta este propriul identificator.
- La fiecare rundă, fiecare unitate de procesare trimite acest maxim vecinilor.
- După d runde, dacă maximul cunoscut este propriul identificator, atunci unitatea de procesare se declară *lider*; altfel se consideră *non-lider*.



Algoritmul FloodMax sincron

Pseudocod pentru unitatea de procesare p_i :

- **Notății:**

- id_i reprezintă identificatorul unității de procesare p_i .
- R_i memorează mesajele recepționate de unitatea de procesare p_i într-o rundă.
- max_id_i memorează cel mai mare identificator cunoscut de p_i .
- $statut_i$ memorează starea finală a unității de procesare p_i (*lider* sau *non-leader*).

- **Premise:**

- Inițial $max_id_i = id_i$ și $statut_i = \text{necunoscut}$.

FLOODMAX_LE($G, d, p_i, max_id_i, statut_i$)

```

1   for runda  $\leftarrow 1$  to  $d$ 
2   do for each ( $m$ ) in  $R_i$ 
3       do if  $max\_id_i < m$ 
4           then  $max\_id_i \leftarrow m$ 
5           trimite  $max\_id_i$  vecinilor
6   if  $max\_id_i = id_i$ ;
7       then  $statut_i \leftarrow \text{lider}$ 
8       else  $statut_i \leftarrow \text{non-leader}$ 
```



Complexitatea

Teorema (5)

Numărul de mesaje circulă prin graf este $2d|E|$.

Demonstrație.

Într-o rundă, pe fiecare muchie circulă două mesaje. Numărul rundelor este d . □

Algoritmi paraleli și distribuiți

Excluderea mutuală în sisteme "shared-memory"

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Introducere

Excluderea mutuală cu registri binari *test&set*

Descriere

Pseudocod

Excluderea mutuală cu registri *read-modify-write*

Descriere

Pseudocod

Excludere mutuală cu registri cu valori mărginite

Excludere mutuală cu registri cu valori mărginite, pentru două unități de procesare, cu flămânzire

Descriere

Pseudocod

Excludere mutuală cu registri cu valori mărginite, pentru două unități de procesare, fără flămânzire

Descriere

Pseudocod



Introducere

- Problemă fundamentală: în sistemele distribuite este frecvent nevoie de a arbitra accesul concurrent la o resursă.
- Problema excluderii mutuale apare la un grup de unități de procesare care periodic accesează anumite resurse care nu pot fi utilizate simultan de mai mult de o singură unitate de procesare (de exemplu o imprimantă).
- Fiecare unitate de procesare poate dori să execute un segment de cod numit secțiune critică, astfel încât, la orice moment, cel mult o unitate de procesare este în secțiunea critică (*excluderea mutuală*).
- *Dacă una sau mai multe unități de procesare încearcă să intre în secțiunea critică, atunci una dintre ele va reuși în cele din urmă, atâta timp cât nici o unitate de procesare nu stă la infinit în secțiunea critică.*
- Proprietatea de mai sus nu dă nici o garanție de reușită, deoarece o unitate de procesare poate încerca să intre în secțiunea critică și totuși să nu reușească, aceasta fiind ignorată mereu de alte unități de procesare.
- O proprietate mai puternică, care elimină *blocarea (deadlock)*, este *fără flămânzire (no lockout sau no starvation)*: *Dacă o unitate de procesare dorește să intre în secțiunea critică, aceasta va reuși în cele din urmă, atâta timp cât nici o unitate de procesare nu ramâne la infinit în secțiunea critică.*

Introducere - continuare

- Programul unei unități de procesare este împărțit în următoarele secțiuni:

Entry: codul executat la pregatirea pentru intrarea în secțiunea critică;
Critical: codul care trebuie protejat de execuție concurentă;
Exit: codul executat după părăsirea secțiunii critice;
Remainder: restul codului.

Excluderea mutuală cu registri binari *test&set* - descriere

Un registru *test&set* memorează valori binare și suportă două operații atomice, *test&set* și *reset*, definite astfel:

Notatii:

- r este un registru *test&set*.

TEST&SET(r)

- ```

1 temp $\leftarrow r$
2 r $\leftarrow 1$
3 return ter

```

RESET( $r$ )

- $$1 \quad r \leftarrow 0$$

## Excluderea mutuală cu regiștri binari *test&set* - descriere (continuare)

- Se presupune că valoarea inițială a registrului  $r$  este 0.
- În secțiunea de intrare, unitatea de procesare  $p_i$  testează  $r$  în mod repetat  $r$ , până când testul returnează 0; după ultimul test  $r$  va conține 1
- Orice test ulterior va returna 1, ceea ce interzice oricărei alte unități de procesare să intre în secțiunea critică.
- În secțiunea de ieșire,  $p_i$  reseteaza  $r = 0$ , astfel încât una dintre unitățile de procesare, care așteaptă în secțiunea de intrare, să poată intra în secțiunea critică.
- Excluderea mutuală realizată prin acest algoritm este fără blocaj (*no deadlock*), dar cu flămânzire (*lockout*).

## Excluderea mutuală cu registri binari *test&set* - pseudocod

*Notății:*

- $r$  este un registr *test&set*.

*Premise:* Inițial,  $r = 0$

**Pseudocod pentru unitatea de procesare  $p_i$ ,  $i \in \{0, 1, \dots, n-1\}$**

$\langle \text{Entry} \rangle /* p_i \text{ are nevoie de resursa critică} */$

ENTRY( $r$ )

1 asteapta pana cand TEST&SET( $r$ ) = 0

$\langle \text{Critical} \rangle /* p_i \text{ intră în secțiunea critică} */$

$\langle \text{Exit} \rangle /* p_i \text{ părăsește secțiunea critică} */$

EXIT( $r$ )

1 RESET( $r$ )

$\langle \text{Remainder} \rangle /* p_i \text{ nu are nevoie de resursa critică} */$



## Excluderea mutuală cu registri *read-modify-write* - descriere

- Un registru  $r$ , de tipul *read-modify-write*, are următoarele proprietăți:
  - Valorile  $v$  memorate de registrul  $r$  sunt numere întregi.
  - Operația suportată este  $RMW(r, f)$ , unde  $f$  este o funcție definită pe  $\mathbb{Z}$  cu valori în  $\mathbb{Z}$ , adică  $f : \mathbb{Z} \leftarrow \mathbb{Z}$ .
  - Operația  $RMW(r, f)$  este atomică, întoarce valoarea  $v$  memorată în registrul  $r$  și schimbă conținutul lui  $r$ , înscriind acolo valoarea  $f(v)$ .
- Operația *test&set* este un caz particular al operației  $RMW$ , în care  $f(r) = 1$ , pentru orice valoare a lui  $r$ .

$RMW(r, f)$

- 1     $v \leftarrow r$
- 2     $r \leftarrow f(v)$
- 3    **return**  $v$

- Unitățile de procesare sunt organizate într-o coadă de tip *FIFO*
- Numai unitatea de procesare de la începutul cozii poate intra în secțiunea critică și rămâne la începutul cozii până părăsește secțiunea critică, nepermittând în acest fel celorlalte unități de procesare să intre în secțiunea critică.
- Excluderea mutuală realizată prin acest algoritm este fără blocaj (*no deadlock*) și fără flămânzire (*no lockout*).



## Excluderea mutuală cu registri *read-modify-write* - descriere (continuare)

- Fiecare unitate de procesare dispune de două variabile locale, *pozitie* și *coada*.
- Algoritmul utilizează un registru  $r$  de tip *read-modify-write*, care memorează o structură formată din două câmpuri: *primul* și *ultimul*.
- $r.\text{primul}$  și  $r.\text{ultimul}$  conțin "bilete" pentru prima și respectiv pentru ultima unitate de procesare din coadă.
- Atunci când o unitate de procesare ajunge în secțiunea de intrare, se "așează" la coada, prin asignarea valoarei lui  $r$  unei variabile locale și prin incrementarea lui  $r.\text{ultimul}$ . Toate acestea sunt realizate printr-o singură operatie, indivizibilă.
- Valoarea curentă a lui  $r.\text{ultimul}$  îndeplinește rolul de "bilet" pentru unitatea de procesare.
- O unitate de procesare asteaptă în secțiuneade intrare pâna devine prima din coadă, adică până când  $r.\text{primul}$  este egal cu biletul său. În acest moment, unitatea de procesare intră în secțiunea critică.
- După părăsirea secțiunii critice, unitatea de procesare se autoelimină din coadă prin incrementarea lui  $r.\text{primul}$ , permitând în acest fel urmatoarei unități de procesare din coadă să intre în secțiunea critică.

## Excluderea mutuală cu registri *read-modify-write* - pseudocod

*Notății:*

- $r$  este un registrul de tipul *read-modify-write*

*Premise:* Inițial,  $r = (0,0)$

**Pseudocod pentru unitatea de procesare  $p_i$ ,  $i \in \{0, 1, \dots, n-1\}$**

$\langle \text{Entry} \rangle /* p_i are nevoie de resursa critică */$

ENTRY( $r$ )

- 1  $\text{pozitie} \leftarrow \text{RMW}(r, \langle r.\text{primul}, r.\text{ultimul} + 1 \rangle) /*$  se "așează" la coadă \*/
- 2 **repeat**
- 3         $\text{coada} \leftarrow \text{RMW}(r, r) /*$  "citește" coada \*/
- 4        **until**  $\text{coada}.\text{primul} = \text{pozitie}.\text{ultimul} /*$  până ajunge prima \*/

$\langle \text{Critical} \rangle /* p_i intră în secțiunea critică */$

$\langle \text{Exit} \rangle /* p_i părăsește secțiunea critică */$

EXIT( $r$ )

- 1  $\text{RMW}(r, \langle r.\text{primul} + 1, r.\text{ultimul} \rangle)$

$\langle \text{Remainder} \rangle /* p_i nu are nevoie de resursa critică */$



## Excludere mutuală cu registri cu valori mărginite, pentru două unități de procesare, cu flămânzire - descriere

- Fiecare unitate de procesare  $p_i$  dispune de o variabilă booleană  $vreau[i]$  partajată, a cărei valoare este 1, dacă  $p_i$  este interesată să intre în secțiunea critică și 0 în caz contrar.
- Algoritmul este asimetric:
  - $p_0$  intră în secțiunea critică fără să țină seama de încercările lui  $p_1$  de a face același lucru.;
  - $p_1$  intră în secțiunea critică numai dacă  $p_0$  nu este interesat de aceasta.
- Acest algoritm utilizează un mecanism cu fanioane pentru coordonarea unităților de procesare care își dispută secțiunea critică:
  - $p_i$  ridică un fanion (prin setarea  $vreau[i] \leftarrow 1$ ) și apoi urmărește fanionul celeilalte unități de procesare (prin citirea lui  $vreau[1 - i]$ ).
  - Cel puțin una dintre unitățile de procesare observă fanionul ridicat al celeilalte unități de procesare și evită intrarea în secțiunea critică (excludere mutuală).
- Dacă  $p_0$  este interesată continuu să intre în secțiunea critică, este posibil ca  $p_1$  să nu intre niciodată în secțiunea critică, având în vedere faptul că aceasta cedează de fiecare dată (*lockout*).



## Excludere mutuală cu registri cu valori mărginite, pentru două unități de procesare, cu flămânzire - pseudocod

**Premise:** Inițial,  $vreau[0] = vreau[1] = 0$

**Pseudocod pentru unitatea de procesare  $p_0$**

$\langle \text{Entry} \rangle /* p_0 \text{ are nevoie de resursa critică} */$

ENTRY( $vreau$ )

1  $vreau[0] \leftarrow 1$

2 *asteapta pana cand*  $vreau[1] = 0$

$\langle \text{Critical} \rangle /* p_0 \text{ intră în secțiunea critică} */$

$\langle \text{Exit} \rangle /* p_0 \text{ părăsește secțiunea critică} */$

EXIT( $r$ )

1  $vreau[0] \leftarrow 0$

$\langle \text{Remainder} \rangle /*$

$p_0$  nu are nevoie de resursa critică\*/

**Pseudocod pentru unitatea de procesare  $p_1$**

$\langle \text{Entry} \rangle /* p_1 \text{ are nevoie de resursa critică} */$

ENTRY( $vreau$ )

1  $vreau[1] \leftarrow 0$

2 *asteapta pana cand*  $vreau[0] = 0$

3  $vreau[1] \leftarrow 1$

4 **if**  $vreau[0] = 1$

5     **then go to** linia 1

$\langle \text{Critical} \rangle /* p_1 \text{ intră în secțiunea critică} */$

$\langle \text{Exit} \rangle /* p_1 \text{ părăsește secțiunea critică} */$

EXIT( $r$ )

1  $vreau[1] \leftarrow 0$

$\langle \text{Remainder} \rangle /*$

$p_1$  nu are nevoie de resursa critică\*/



## Excludere mutuală cu registri cu valori mărginite, pentru două unități de procesare, fără flămânzire - descriere

- Pentru a obține *no lockout* se modifică algoritmul precedent, astfel încât  $p_0$  să nu mai fie priorităț continuu.
- Fiecare unitate de procesare va acorda prioritate celeilalte, după ce ieșe din secțiunea critică.
- O variabilă partajată *prioritate* conține identificatorul (indicele) unității de procesare care este priorităț momentan; *prioritate* este initializată cu 0, adică, inițial  $p_0$  este priorităț.
- Variabila *prioritate* este citită și modificată de ambele unități de procesare.
- Unitatea de procesare priorităț joacă rolul lui  $p_0$  din algoritmul precedent; prin urmare, aceasta va intra în secțiunea critică.
- La ieșire, va da prioritate celeilalte unități de procesare, și va juca rolul lui  $p_1$  din algoritmul precedent. Aceasta asigura proprietatea *no lockout*.
- Algoritmul modificat indeplinește condițiile de excludere mutuală în aceeași manieră ca și primul algoritm.
  - O unitate de procesare ridică un fanion și apoi inspectează fanionul celeilalte unități de procesare;
  - Cel puțin una dintre unitățile de procesare observă că fanionul celeilalte unitati de procesare este ridicat și evită să intre în secțiunea critică.



**Excludere mutuală cu registri cu valori mărginite, pentru două unități de procesare, fără flămânzire - pseudocod**

Premise: Initial,  $vreau[0] = vreau[1] = 0$

#### Pseudocod pentru unitatea de procesare

*⟨Entry⟩ /\* p<sub>0</sub> are nevoie de resursa critică\*/*  
**ENTRY(vreau, prioritate)**

```

1 vreau[0] ← 0
2 asteapta pana cand vreau[1] = 0 sau
3 prioritate = 0
4 vreau[0] ← 1
5 if prioritate = 1
6 then if vreau[1] = 1
7 then go to linia 1
8 else asteapta pana cand vreau[1] = 0

```

*⟨Critical⟩ /\* Pășește în secțiunea critică\*/*

`<Exit> /* p0 părăsește secțiunea critică*/`

EXIT(*r*)

```
1 vreau[0] \leftarrow 0
2 prioritate \leftarrow 1
```

*<Remainder>* /\*

*p<sub>0</sub>* nu are nevoie de resursa critică\*/

#### Pseudocod pentru unitatea de procesare $p_1$

$\langle \text{Entry} \rangle /* p_1 \text{ are nevoie de resursa critică*/$   
 $\text{ENTRY}(vreau)$

```

1 vreau[1] ← 0
2 asteapta pana cand vreau[0] = 0 sau
3 prioritate = 1
4 vreau[1] ← 1
5 if prioritate = 0
6 then if vreau[0] = 1
7 then go to linia 1
8 else asteapta pana cand vreau[0] =

```

*{Critical}* /\* p<sub>1</sub> intră în secțiunea critică\*/

$\langle \text{Exit} \rangle /* p_1 \text{ părăsește secțiunea critică*/$   
 $\text{EXIT}(r)$

```

1 vreau[1] = 0
2 prioritate ← 0

```

*<Remainder>* /\*

*P<sub>1</sub>* nu are nevoie de resursa critică\*/



# Algoritmi paraleli și distribuiți

## Echilibrarea încărcării

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași



## Cuprins

### Introducere

### Echilibrarea statică

Tipuri de echilibrare statică a încărcării

Dezavantaje

### Echilibrarea dinamică

Tipuri de echilibrare dinamică a încărcări

Echilibrarea dinamică centralizată

Echilibrarea dinamică descentralizată

Echilibrarea dinamică prin metoda initializării transferurilor de task-uri de către transmițător

### Bibliografie



## Introducere

- Într-o execuție paralelă, fiecare unitate de procesare primește un grup de sarcini (*task-uri*).
- Pentru a crește eficiența procesării paralele ar trebui ca execuția grupurilor de *task-uri* să se termine aproximativ simultan.
- Echilibrarea statică:
  - În literatura de specialitate este denumită *problema mapării* (Bokhari, 1981) sau *problema planificării (scheduling problem)*.
  - *Task-urile* sunt alocate unităților de procesare înainte de startarea aplicațiilor.
  - Se bazează îndeosebi pe estimarea volumului de lucru și pe partaționarea acestuia în segmente alocate fiecărui *task* în parte. Această estimare are foarte puține șanse să fie exactă.
- Echilibrarea dinamică:
  - *Task-urile* sunt alocate unităților de procesare în timpul rulării aplicațiilor.
  - Unitățile de procesare se pot “evalua” între ele și apoi pot redistribui sarcini.



## Tipuri de echilibrare statică a încărcării

- *Round robin*: Sunt transmise *task-uri* în ordinea secvențială a unităților de procesare, revenind la prima atunci când tuturor unităților de procesare li s-a acordat un *task*.
- *Selecția aleatorie*: Sunt selectate unități de procesare la întâmplare și li se acordă *task-uri*.
- *Bisecția recursivă*: Problema este divizată în mod recursiv în subprobleme de efort computațional egal, minimizându-se în același timp transmiterea de mesaje.
- *Tehnici bazate pe algoritmi genetici*.
- Pentru unitățile de procesare interconectate cu o rețea de legături statice, *task-urile* comunicante trebuie executate pe unități de procesare cu căi directe de comunicație pentru a reduce întârzierile; aceasta este un element esențial al problemei „mapării” pentru astfel de sisteme.
- În general, maparea este o problemă din clasa problemelor  $\text{NP-dificele}$ , pentru care nu există un algoritm de rezolvare cu un timp de execuție polinomial. De aceea, de multe ori se folosesc metode euristiche pentru selecția unităților de procesare.



## Dezavantaje

- Este foarte dificil de estimat cu acuratețe timpii de execuție ale diverselor părți ale unui program, fără executarea efectivă a acestora.
  - Planificarea este de la început inexactă.
- Unele sisteme pot avea întârzieri în comunicații care variază în funcție de diverse circumstanțe.
  - Integrarea întârzierilor variabile în echilibrarea statică a încărcării poate fi foarte dificilă.
- Unii algoritmi au un număr de pași care nu poate fi calculat în prealabil (de exemplu, algoritmii bazați pe *backtracking*).



## Tipuri de echilibrare dinamică a încărcării

- *Echilibrare centralizată*: există o structură clară *master-slave*; sarcinile sunt distribuite de către unitatea de procesare *master*.
- *Echilibrare descentralizată*: unitățile de procesare interacționează între ele și pot face schimb de *task-uri* conform cu nivelul de încărcare.
- *Sarcinile sunt transferate prin metoda initializării de către receptor (receiver-initiated)*: unitatea de procesare subîncărcată trimite cereri de *task-uri* la celelalte unități de procesare.
- *Transferurile sunt efectuate prin metoda initializării de către transmițător*: unitatea de procesare supraîncărcată trimite cereri de transfer.



## Echilibrarea dinamică centralizată - Descriere

- Unitatea de procesare *master* dispune de colecția de *task-uri* care urmează a fi executate.
- *Task-urile* sunt trimise procesoarelor *slave*.
- Atunci când o unitate de procesare *slave* termină un *task*, cere un altul de la unitatea de procesare *master*.
  - Mecanismul este numit “*work pool*”.



## Echilibrarea dinamică centralizată - Cum și când se aplică?

- Task-urile sunt cunoscute în prealabil, sunt sensibil diferite.
  - Este cazul problemelor rezolvabile prin metoda *divide-and-conquer*.
  - Sunt trimise spre execuție întâi task-urile care durează mai mult și folosesc multă memorie.
  - Dacă un task consumator intensiv de resurse ("mare") este trimis mai târziu, task-urile mai "mici" pot fi terminate de către unitătile de procesare *slave*, care vor sta apoi inactive în aşteptarea terminării task-ului mai "mare".
- Numărul de task-uri se modifică în timpul execuției.
  - În unele aplicații, execuția unui task poate genera noi task-uri, deși în final numărul task-urilor trebuie să se reducă la 0, semnalizând terminarea procesului de calcul.
  - Dacă task-urile sunt la fel de importante, poate fi folosită o coadă simplă *FIFO*.
  - Dacă unele task-uri sunt mai importante decât altele (de exemplu, pot conduce mai repede către o soluție), va fi implementată o coadă cu priorități.
- Unitatea de procesare *master* poate deține și alte informații, cum ar fi soluția optimă curentă.
  - Este cazul problemelor rezolvabile prin metoda *greedy* sau metoda *programării dinamice*.
  - Task-urile vor fi trimise spre execuție în ordinea dictată de soluția optimă.



## Echilibrarea dinamică descentralizată - 5 faze

- Evaluarea încărcării:
  - Metode analitice de evaluare: metodele de acest tip "reacționează" în timp real la schimbări, dar nu sunt precise.
  - Metode empirice: sunt precise dar nu pot anticipa schimbările.
  - Metode hibride: sunt combinații între cele două metode; s-au dovedit a fi cele mai eficiente.
- Determinarea eficienței redistribuirii *task*-urilor:
  - Transferul de *task*-uri între unitățile de procesare poate avea un cost destul de mare.
  - Este bine să se estimeze de la început dacă transferul de *task*-uri va micșora timpul total de procesare.
- Calculul vectorilor de transfer:
  - Metoda difuziei caldurii.
  - Metoda gradientului.
  - Metode ierarhice: divizarea recursiva și reechilibrarea grupului de unități de procesare.
  - Metode specifice domeniului: metoda fâșiei.
- Selecția *task*-urilor pentru transfer:
  - Căutari exhaustive: consumatoare de resurse.
  - Primul potrivit (*first fit*): metodele de acest tip nu sunt optimale.
- Schimbul efectiv de *task*-uri:
  - *Task*-urile trebuie conservate perfect în timpul transferului.
  - Trebuie avut în vedere și un eșec al transferului din cauza lipsei de memorie pe mașina destinație.
  - Se poate face sincron și asincron.



## Echilibrarea dinamică prin metoda inițializării transferurilor de task-uri de către transmițător Algoritmul *SID* (*Sender Initiated Diffusion*)

- Propus de Willebeek.
- Se presupune că încărcările sunt infinit divizibile; încărcarea unei unități de procesare se poate reprezenta printr-un număr real.
  - Presupunerea este aplicabilă în programele paralele care exploatează o granularitate foarte fină.
  - Pentru a fi extins în cazurile granularităților medii sau mari, algoritmul trebuie să poată gestiona task-uri indivizibile, deci să reprezinte încărcarea unui procesor printr-un număr întreg nenegativ.



## Algoritmul SID (*Sender Initiated Diffusion*) - Schema de funcționare

- Fiecare unitate de procesare își trimit informațiile despre încărcare către toți vecinii și primește de asemenea informațiile despre vecini.
- Fiecare unitate de procesare calculează încărcarea medie a domeniului său, incluzând încărcările vecinilor și cea proprie.
- Fie  $w_i(t)$  numărul unităților de încărcare ale unității de procesare  $i$  la momentul  $t$ .  
Fiecare unitate de procesare memorează o estimare a încărcării fiecărei unitate de procesare vecină.
- Fie  $w_{ij}(t)$  estimarea încărcării unității de procesare  $j$  păstrată în memoria unității de procesare  $i$  la momentul  $t$ .
  - Dacă unitățile de procesare  $i$  și  $j$  nu sunt conectate direct, estimarea  $w_{ij}(t) = 0$ .
  - Altfel, dacă unitățile de procesare  $i$  și  $j$  sunt vecine, estimarea încărcării se definește astfel:

$$w_{ij}(t) = w_j(\tau_{ij}(t))$$

unde  $\tau_{ij}$  este o variabilă întreagă care satisfacă relația  $0 \leq \tau_{ij}(t) \leq t$ .



## Algoritmul SID (*Sender Initiated Diffusion*) - Schema de funcționare (continuare)

- Fie  $w_{ii}(t)$  încărcarea unității de procesare  $i$  la momentul  $t$ , ceea ce înseamnă că fiecare unitate de procesare își poate preciza propria încărcare la momentul  $t$ .
- Există o mulțime de momente de timp,  $T$ , numită *mulțimea timpilor de echilibrare a încărcării*, la care fiecare unitate de procesare  $i$  își compară încărcarea proprie cu estimarea memorată a încărcării vecinilor.
- Media estimată a încărcării procesorului  $i$  și a vecinilor săi este memorată de variabila  $w_i(t)$ .
- Definim  $(a_{ij})$  = matricea de adiacență la care se adaugă matricea unitate:

$$(a'_{ij}) = I_n + (a_{ij})$$

- Media estimată a încărcării domeniului unității de procesare  $i$  este calculată astfel:

$$\overline{w}_i(t) = \frac{(a'_{i1} \dots a'_{in}) \begin{pmatrix} w_{i1}(t) \\ \vdots \\ w_{in}(t) \end{pmatrix}}{(a'_{i1} \dots a'_{in}) \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}}$$



## Algoritmul SID (*Sender Initiated Diffusion*) - Schema de funcționare (continuare)

- Dupa ce unitatea de procesare  $i$  și-a calculat media de încărcare locală,  $\bar{w}_i(t)$ , aceasta calculează și deficitul  $d_{ii}(t) = \bar{w}_i(t) - w_i(t)$ , pentru a determina dacă este supraîncărcat sau nu.
- Dacă este supraîncărcat,  $d_{ii}(t) < 0$
- Dacă este subîncărcat,  $d_{ii}(t) \geq 0$
- O unitate de procesare supraîncărcată repartizează excesul de încărcare vecinilor subîncărcăți.
- Pentru a evalua corect acest exces, este calculată o nouă pondere pentru toți vecinii deficitari:

$$d_{ij}^+ = \begin{cases} d_{ij}(t) & , \text{dacă } d_{ij}(t) > 0 \\ 0 & , \text{altfel} \end{cases}$$



## Algoritmul SID (*Sender Initiated Diffusion*) - Schema de funcționare (continuare)

- Cantitatea totală de deficit este:

$$dt = \sum_{j=1}^n d_{ij}^+(t)$$

- Porțiunea din excesul unității de procesare  $i$  care este atribuită vecinului  $j$  este:

$$P_{ij} = \begin{cases} \frac{d_{ij}^+(t)}{dt} & , \text{ dacă } d_{ij}(t) < 0 \\ 0 & , \text{ altfel} \end{cases}$$

- În momentul  $t$  este transferată o cantitate nenegativă de încărcare,  $s_{ij}(t)$ , de la  $i$  la  $j$ :

$$\sigma_{ij}(t) = \lfloor -P_{ij}(t) \cdot d_{ij}(t) \rfloor$$

- Algoritmul se termină atunci când nimeni nu mai dispunde de volum de lucru de transferat.

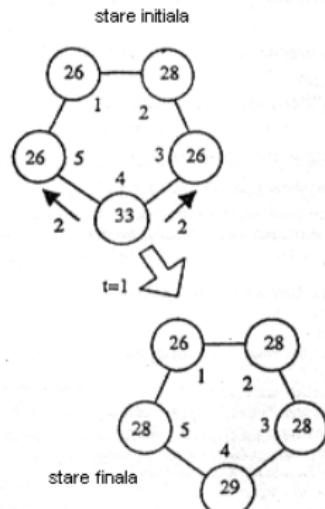


## Algoritmul SID (*Sender Initiated Diffusion*) - Exemplu de execuție

| procesorul (i) | 1           | 2             | 3           | 4             | 5           |
|----------------|-------------|---------------|-------------|---------------|-------------|
| $w_i(l)$       | 26          | 28            | 26          | 33            | 26          |
| $\bar{w}_i(l)$ | 26,6        | 26,6          | 29          | 28,3          | 28,3        |
| $d_{ii}(l)$    | 0,6         | -1,3          | 3           | -4,6          | 2,3         |
| stare          | subîncărcat | supraîncărcat | subîncărcat | supraîncărcat | subîncărcat |

|                                               |                     |
|-----------------------------------------------|---------------------|
| $d_{21}^+(l) = 0,6$                           | $d_{23}^+(l) = 0,6$ |
| $td = 1,3$                                    |                     |
| $P_{21}(l) = 0,5$                             | $P_{23}(l) = 0,5$   |
| $s_{21}(l) = 0$                               | $s_{23}(l) = 0$     |
| <i>nu se generează schimburi de încărcare</i> |                     |

|                                         |                     |
|-----------------------------------------|---------------------|
| $d_{45}^+(l) = 2,3$                     | $d_{43}^+(l) = 2,3$ |
| $td = 4,6$                              |                     |
| $P_{45}(l) = 0,5$                       | $P_{43}(l) = 0,5$   |
| $s_{45}(l) = 2$                         |                     |
| <i>trimite 2 unități procesorului 5</i> |                     |
| $s_{43}(l) = 2$                         |                     |
| <i>trimite 2 unități procesorului 3</i> |                     |





## Biblografie

1. J. Watts, "A Practical Approach to Dynamic Load Balancing", Masters Thesis, California Institute of Technology, 1995, USA
2. R. Diekmann, B. Monien, R. Preis, "Load Balancing Strategies for Distributed Memory Machines", Technical Report, University of Paderborn, 1997, Germany
3. A.Cortés, A. Ripoll, M.A. Senar, F.Cedó and E. Luque, "On the convergence of SID and DASUD load-balancing algorithms", Technical Report, Universitat Autònoma de Barcelona, 1998, Spain
4. B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems", IEEE INFOCOM, 2004

|             |                                             |                                                                 |                                              |
|-------------|---------------------------------------------|-----------------------------------------------------------------|----------------------------------------------|
| Introducere | Euristici care asignează pe rând task-urile | Euristici care consideră toate task-urile în procesul de mapare | Compararea principalelor euristici de mapare |
| ○○○         | ○                                           | ○                                                               |                                              |
| ○           | ○                                           | ○                                                               |                                              |
| ○           | ○                                           | ○                                                               |                                              |
| ○           | ○○                                          | ○○                                                              |                                              |
| ○           | ○○                                          |                                                                 |                                              |
| ○           | ○                                           |                                                                 |                                              |
| ○           | ○                                           |                                                                 |                                              |

## Algoritmi paraleli și distribuiți

### Maparea task-urilor în sisteme distribuite eterogene

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

|             |                                             |                                                                 |                                              |
|-------------|---------------------------------------------|-----------------------------------------------------------------|----------------------------------------------|
| Introducere | Euristici care asignează pe rând task-urile | Euristici care consideră toate task-urile în procesul de mapare | Compararea principalelor euristici de mapare |
| ooo         | o                                           | o                                                               |                                              |
| ooo         | o                                           | o                                                               |                                              |
| ooo         | o                                           | o                                                               |                                              |
| ooo         | oo                                          | oo                                                              |                                              |
| ooo         | o                                           | o                                                               |                                              |
| ooo         | o                                           |                                                                 |                                              |

## Maparea task-urilor

### Introducere

Matricea ETC (Expected Time to Compute)

### Euristici care asignează pe rând task-urile

Algoritmul Oportunistic Load Balancing - OLB

Algoritmul Minimum Execution Time - MET

Algoritmul Minimum Completion Time - MCT

Algoritmul de alternare (Switching Algorithm)

Algoritmul  $k$ -Percent Best - KPB

### Euristici care consideră toate task-urile în procesul de mapare

Algoritmul Min-Min

Algoritmul Max-Min

Algoritmul Duplex

Algoritmul Genetic - GA

Algoritmul Simulated Annealing - SA

Algoritmul Genetic Simulated Annealing - GSA

Algoritmul A\*

### Compararea principalelor euristici de mapare a task-urilor

## Introducere

- Maparea task-urilor presupune asignarea task-urilor la unități de procesare și ordonarea lor pentru execuție la unitățile respective.
  - În practică, problema mapării task-urilor este rezolvată prin aplicarea euristicilor de mapare, respectiv, prin metode bazate pe experiență folosite în rezolvarea problemelor unde o căutare exhaustivă a soluției optime nu este posibilă.
  - Indicatorii cei mai utilizați în determinarea eficienței unei euristici de mapare sunt:
    - *Makespan-ul* - timpul în care toate unitățile de procesare termină de executat task-urile asignate;
    - *Dezechilibrul încărcării* - diferența dintre unitatea de procesare cea mai încărcată și cea mai puțin încărcată;
    - *Durata algoritmului* - timpul în care se face asignarea task-urilor.
  - O heuristică de mapare este considerată a fi eficientă dacă obține în cel mai scurt timp o soluție care are un *makespan* minim și o încărcare bine echilibrată.



## Matricea ETC (Expected Time to Compute)

- Euristicile de mapare folosesc estimări ale timpilor de execuție a fiecărui *task* la fiecare unitate de procesare.
- Timpii sunt reținuți într-o matrice ETC (Expected Time to Compute), unde fiecare rând al matricii reprezintă un task, iar fiecare coloană o unitate de procesare.
- Matricea ETC poate fi de trei tipuri: *consistentă*, *inconsistentă* sau *semi-consistentă*.
- Dacă există o relație de ordine astfel încât toți timpii de execuție a task-urilor la un procesor sunt fie toți mai mari sau toți mai mici decât timpii de execuție la un alt procesor, atunci matricea este considerată a fi *consistentă*.
- Dacă nu se poate stabili o astfel de relație de ordine, atunci matricea este *inconsistentă*, iar dacă există o submatrice consistentă într-o astfel de matrice, atunci matricea ETC este *semi-consistentă*.

● ● ○

|   |    |
|---|----|
| ○ | ○  |
| ○ | ○  |
| ○ | ○  |
| ○ | ○  |
| ○ | ○○ |
| ○ | ○○ |
| ○ | ○  |

## Matricea ETC (Expected Time to Compute) - continuare

**Tabela 1 :** Exemplu de matrice ETC consistentă ( $ETC_{i1} < ETC_{i3} < ETC_{i2}$ ,  
 $\forall i \in \{1, \dots, n\}$ )

|       | $P_1$ | $P_2$ | $P_3$ |
|-------|-------|-------|-------|
| $T_1$ | 1.9   | 5.9   | 2.3   |
| $T_2$ | 5.1   | 7.9   | 5.4   |
| $T_3$ | 2.1   | 3.1   | 2.8   |
| $T_4$ | 2.3   | 4.4   | 4.3   |
| $T_5$ | 1.1   | 3.5   | 1.5   |
| $T_6$ | 7.4   | 7.6   | 7.5   |
| $T_7$ | 5.9   | 6.7   | 6.4   |
| $T_8$ | 2.6   | 3.9   | 3.8   |

**Tabela 2 :** Exemplu de matrice ETC semi-consistentă ( $ETC_{i1} < ETC_{i3}$ ,  
 $\forall i \in \{1, \dots, n\}$ )

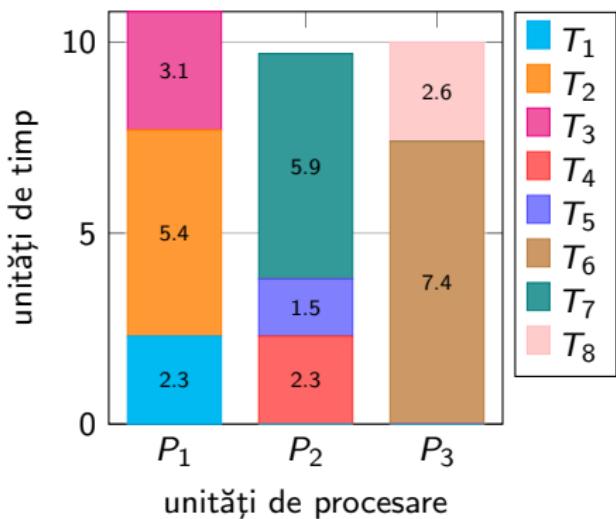
|       | $P_1$ | $P_2$ | $P_3$ |
|-------|-------|-------|-------|
| $T_1$ | 1.9   | 2.3   | 5.9   |
| $T_2$ | 5.1   | 7.9   | 5.4   |
| $T_3$ | 2.1   | 3.1   | 2.8   |
| $T_4$ | 2.3   | 4.4   | 4.3   |
| $T_5$ | 1.1   | 1.5   | 3.5   |
| $T_6$ | 7.4   | 7.6   | 7.5   |
| $T_7$ | 6.4   | 5.9   | 6.7   |
| $T_8$ | 2.6   | 3.9   | 3.8   |



## Matricea ETC (Expected Time to Compute) - continuare (2)

**Tabela 3 :** Exemplu de matrice ETC inconsistentă

|       | $P_1$ | $P_2$ | $P_3$ |
|-------|-------|-------|-------|
| $T_1$ | 2.3   | 5.9   | 1.9   |
| $T_2$ | 5.4   | 7.9   | 5.1   |
| $T_3$ | 3.1   | 2.8   | 2.1   |
| $T_4$ | 4.4   | 2.3   | 4.3   |
| $T_5$ | 3.5   | 1.5   | 1.1   |
| $T_6$ | 7.5   | 7.6   | 7.4   |
| $T_7$ | 6.4   | 5.9   | 6.7   |
| $T_8$ | 3.9   | 3.8   | 2.6   |



**Figura 1 :** O posibilă mapare pentru matricea ETC alăturată



Euristici care asignează pe rând task-urile

- Cea mai simplă modalitate de a mapa task-urile este de a le asigna într-o anumită ordine ținând cont doar de task-urile care au fost asignate deja.
  - Cele mai importante metode de mapare din această categorie sunt:
    - Algoritmul Oportunistic Load Balancing - OLB
    - Algoritmul Minimum Execution Time - MET
    - Algoritmul Minimum Completion Time - MCT
    - Algoritmul de alternare (Switching Algorithm)
    - Algoritmul  $k$ -Percent Best - KPB
  - Majoritatea acestor algoritmi de mapare au complexitatea timp  $O(nm)$ , unde  $n$  reprezintă numărul de task-uri, iar  $m$  numărul de unități de procesare.



## Algoritmul Oportunistic Load Balancing - OLB

- Fiecare task este asignat la următorul procesor disponibil, iar dacă sunt mai multe unități de procesare disponibile, atunci se alege una aleatoriu.
- Avantajul este simplitatea abordării, iar dezavantajul major îl constituie neconsiderarea estimării timpului de execuție a task-ului care urmează să fie asignat.
- Acest lucru poate duce la asignarea task-ului respectiv la o unitate de procesare care, în ciuda faptului că este mai puțin ocupată la momentul respectiv, va executa task-ul într-un timp mult mai mare comparativ cu celelalte unități.
- Complexitatea
  - Complexitatea timp a acestui algoritm reprezintă complexitatea parcurgerii tuturor task-urilor  $O(n)$ , înmulțită cu complexitatea dată de determinarea la fiecare pas a disponibilității minime a unităților de procesare.
  - Determinarea minimului se rezumă la determinarea valorii minime dintr-un vector de  $m$  elemente, având complexitatea  $O(m)$ .
  - Deoarece la fiecare iterație un singur task este asignat, doar o singură valoare a disponibilității unității de procesare se modifică, rezultând complexitatea timp de  $O(1)$ .
  - În concluzie, euristica OLB are complexitatea timp  $O(n) \cdot (O(1) + O(m))$ , care este egală cu  $O(nm)$ .



## Algoritmul Minimum Execution Time - MET

- Fiecare task este asignat unității de procesare care are un timp minim estimat de execuție pentru task-ul respectiv, ignorând disponibilitatea unității.
- Dezavantajul constă în faptul că această metodă poate cauza dezechilibrii semnificative ale încărcării.
- Dacă există un grup de unități de procesare care execută toate task-urile într-un timp mai scurt decât celelalte unități de procesare, atunci toate task-urile vor fi asignate la unitățile din grupul respectiv; ceea ce înseamnă că celelalte unități de procesare vor fi neutilizate.
- Complexitatea - este aceeași ca la algoritmul precedent ( $O(nm)$ ) cu observația că, pentru fiecare task, algoritmul MET calculează timpul minim de execuție (care se determină tot în  $O(m)$ ).



## Algoritmul Minimum Completion Time - MCT

- Fiecare task este asignat unității de procesare care are un timp minim estimat de terminare a execuției pentru task-ul respectiv.
  - Avantajul este combinarea precedentelor două metode prin evitarea situațiilor în care acestea dau rezultate nesatisfăcătoare.
  - Dezavantajul constă în faptul că, în unele situații, task-urile sunt asignate la unități de procesare care nu au un timp minim de execuție.
  - Complexitatea - este aceeași ca la algoritmii precedenți ( $O(nm)$ ) deoarece complexitatea pentru determinarea timpului minim de terminare a execuției este suma dintre calcularea timpului de terminare a execuției ( $O(1)$ ) și determinarea minimului ( $O(m)$ ), care este tot  $O(m)$ .



## Algoritmul de alternare (Switching Algorithm)

- Într-o manieră ciclică sunt alternate precedentele două metode de echilibrare a încărcării pentru a beneficia de părțile bune ale celor două metode.
  - Pentru început este utilizat algoritmul MET, iar când dezechilibrarea ajunge prea mare este folosit algoritmul MCT pentru a echilibra încărcarea, după care se revine la MET.
  - Complexitatea - este dată de complexitățile algoritmilor MET și MCT, și este, de asemenea, egală cu  $O(nm)$ .

○○○

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

## Algoritmul k-Percent Best - KPB

- În asignarea task-urilor este considerat doar un subset de unități de procesare, format din cele mai bune  $k\%$  unități în funcție de timpul de execuție al task-ului respectiv.
- Ideea acestui algoritm este de a asigna task-uri unităților de procesare celor mai potrivite, având în vedere o eventuală sosire a unor task-uri care s-ar potrivi mai bine doar anumitor unități.
- Algoritmul este următorul:
  1. Generarea unei valori  $k$  aleatoare, unde  $\frac{100}{m} \leq k \leq 100$ ;
  2. Selectarea primelor  $\frac{km}{100}$  procesoare în funcție de timpul minim de execuție pentru task-ul selectat;
  3. Aaignarea task-ului la unitatea de procesare cu cel mai bun timp de terminare a execuției din subsetul selectat anterior.
- Complexitatea
  - Determinarea complexității depinde de modalitatea de implementare a Pasului 2.
  - Dacă cele mai bune  $k$  unități de procesare sunt determinate prin utilizarea unui algoritm de sortare (de exemplu, Merge-Sort), atunci complexitatea pentru maparea unui task este  $O(m \cdot \log m)$ .
  - Astfel, complexitatea timp a algoritmului k-Percent Best este  $O(nm \cdot \log m)$ .



## Euristici care consideră toate task-urile în procesul de mapare

- Algoritmul Min-Min
- Algoritmul Max-Min
- Algoritmul Duplex
- Algoritmul Genetic - GA
- Algoritmul Simulated Annealing - SA
- Algoritmul Genetic Simulated Annealing - GSA
- Algoritmul A\*



## Algoritmul Min-Min

- Task-urile sunt asignate astfel încât modificarea disponibilităților unităților de procesare să fie minimă.
- Algoritmul este bazat pe metoda MCT și constă în:
  1. Calcularea timpilor estimați de terminare a execuției fiecărui task la fiecare unitate de procesare;
  2. Determinarea timpului minim pentru fiecare task;
  3. Așezarea task-ului cu timpul minim din setul de task-uri la unitatea de procesare corespunzătoare;
  4. Înlăturarea din lista de task-uri de mapat a task-ului asignat și repetarea pașilor până când nu mai sunt task-uri de asignat.
- Euristica Min-Min asignează întâi task-urile cu cel mai mic timp de execuție la unitățile de procesare care vor executa cel mai rapid task-urile.
- Complexitatea
  - Fiecare iterație a algoritmului are complexitatea  $O(nm)$ , care este dată de calcularea timpilor estimați de terminare a execuției fiecărui task la fiecare unitate de procesare.
  - Întrucât la fiecare iterație este asignat un task, complexitatea timp a algoritmului este  $O(n) \cdot O(nm)$ , și anume  $O(n^2m)$ .



## Algoritmul Max-Min

- Acest algoritm este similar cu cel precedent cu mențiunea că la pasul al 3-lea este asignat task-ul cu timpul maxim de terminare a execuției și nu cu cel minim.
- Max-Min oferă rezultate mai bune decât Min-Min dacă avem task-uri cu timpi de execuție mai mari, deoarece, în cazul Min-Min, s-ar executa întâi task-urile cu timpi mai mici, iar când se execută cele cu timpi mari se poate ca unele unități de procesare să stea în aşteptare mai mult ceea ce ar duce la o creștere a makesan-ului.
- Complexitatea - analog cu algoritmul Min-Min, complexitatea timp a algoritmului Max-Min este  $O(n^2m)$ .



## Algoritmul Duplex

- Întrucât în anumite situații algoritmul Min-Min produce soluții mai bune, iar în altele situații algoritmul Max-Min oferă rezultate mai satisfăcătoare, euristica Duplex utilizează cei doi algoritmi și alege soluția cea mai bună dintre cele două.
- Dezavantajul constă în dublarea timpului de execuție al algoritmului.
- Complexitatea - complexitatea timp este egală cu suma complexităților algoritmilor Min-Min și Max-Min și este tot  $O(n^2m)$ .



## Algoritmul Genetic - GA

- Algoritmii Genetici sunt folosiți pentru căutarea în spațiul soluțiilor, atunci când acesta este foarte mare.
- Pașii unui algoritm genetic sunt următorii:
  1. Generarea populației inițiale;
  2. Evaluarea populației;
  3. Atât timp cât nu este satisfăcută condiția de oprire se urmează pașii:
    - 3.1 Selectarea cromozomilor;
    - 3.2 Aplicarea operatorului de încrucișare;
    - 3.3 Aplicarea operatorului de mutație;
    - 3.4 Evaluarea cromozomilor și determinarea noii populații.



## Algoritmul Genetic - GA - continuare

- Algoritmii Genetici pot fi optimizați pentru a converge mai rapid către o soluție bună prin implementarea unor operatori de încrucișare și mutație orientați spre îmbunătățirea soluției candidate.
- De asemenea, se poate obține, într-un timp relativ mic, o soluție acceptabilă dacă un cromozom din populația inițială este inițializat cu soluția obținută prin execuția unei euristici de mapare mai ineficiente dar mult mai rapidă, cum ar fi: Min-Min, Max-Min sau Duplex.

| $P_1$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_3$ | $P_2$ | $P_3$ |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     |

Figura 2 : Reprezentarea cromozomului algoritmului genetic pentru soluția din Fig. 1



## Algoritmul Simulated Annealing - SA

- Această tehnică iterativă consideră doar o singură soluție posibilă pentru fiecare *task* în parte, soluția având aceeași reprezentare ca și în cazul algoritmilor genetici.
- Algoritmul SA permite acceptarea unor soluții mai puțin bune pentru a obține un spațiu mai bun al soluțiilor, cu o probabilitate bazată pe o temperatură a sistemului care descrește la fiecare iterație cu o rată de răcire.
- Chiar dacă în general, acest algoritm este mai rapid decât GA, soluțiile găsite sunt mai slabe decât Min-Min și GA.



## Algoritmul Simulated Annealing - SA - continuare

- Algoritmul Simulated Annealing este următorul:

1. Generarea unei soluții inițiale aleatoare;
2. Calcularea temperaturii sistemului pentru soluția generată (*makespan*-ul soluției inițiale);
3.  $\text{Temperatura\_veche} \leftarrow \text{temperatura\_sistemului};$
4. Atât timp cât temperatura sistemului nu a ajuns la o valoare limită:
  - 4.1 Aplicarea operatorului de mutație: asignarea aleatorie a unui *task* la o unitate de procesare;
  - 4.2  $\text{Temperatura\_noua} \leftarrow \text{makespan\_soluție\_nouă};$
  - 4.3 Deciderea asupra acceptării noii soluții: soluția este acceptată dacă noul *makespan* este mai bun (mai mic) decât vechiul sau dacă o valoare aleatoare din intervalul  $[0,1]$  este mai mică decât  $\frac{1}{1 + e^{\frac{\text{temperatura\_veche} - \text{temperatura\_noua}}{\text{temperatura\_sistemului}}}}$
  - 4.4  $\text{Temperatura\_sistemului} \leftarrow \text{temperatura\_sistemului} * \text{rata\_de\_racire};$
  - 4.5 Dacă soluția nouă este acceptată, atunci  $\text{Temperatura\_veche} \leftarrow \text{temperatura\_nouă}.$



## Algoritmul Genetic Simulated Annealing - GSA

- Acest algoritm folosește tehnica GA, doar că în cazul procesului de selecție GSA folosește temperatura sistemului pentru acceptarea sau refuzarea unui cromozom nou.



## Algoritmul A\*

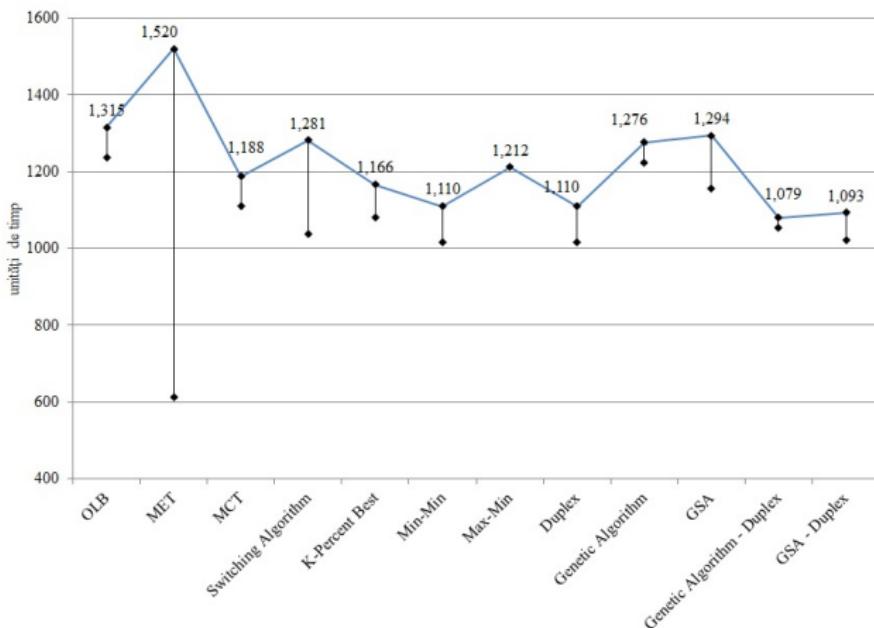
- A\* este o modalitate de căutare bazată pe arbori.
- Un nod din arbore reprezintă mapări parțiale (un subset de task-uri asignate la unități de procesare), iar nodul rădăcină este soluția nulă.
- Fiecare nod părinte are cu un task mai puțin față de fiecare copil al său.
- Un nod generează câte un nod copil pentru fiecare mapare posibilă a unui task. După ce părintele a generat nodurile copii, acesta devine inactiv.
- Fiecare nod are asociat un cost care reprezintă suma dintre timpul total de execuție a task-urilor din soluția parțială (*makespan-ul*) și estimarea diferenței dintre *makespan-ul* soluției parțiale și *makespan-ul* soluției complete care include soluția parțială respectivă.
- Algoritmul folosește metoda *branch-and-bound* cu cost minim.

3



## Compararea principalelor euristici de mapare a task-urilor

**Figura 3 :** Comparația makespan-ului și a dezechilibrării încărcării pentru 12 euristici al căror scop a fost maparea a 500 de task-uri la 20 de unități de procesare



|     |    |
|-----|----|
| ooo | o  |
| o   | o  |
| o   | o  |
| o   | oo |
| o   | oo |
| o   | o  |
| o   | o  |

## Bibliografie

-  Adrian Alexandrescu, Mitică Craus (prof. coord.). *Applications of Artificial Intelligence in Heterogeneous Computing Systems*. Teză de doctorat, 2012.
-  Frederic Magoules, Jie Pan, Kiat-An Tan and Abhinit Kumar. *Introduction to grid computing*. CRC Press, Inc., Boca Raton, FL, USA, 2009.
-  E. Ilavarasan and P. Thambidurai. *Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments*. Journal of Computer Sciences, no. 3, pg. 94–103, 2007.
-  Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bölöni, Muthucumara Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen and Richard F. Freund. *A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems*. J. Parallel Distrib. Comput., vol. 61, pg. 810–837, 2001.
-  Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen and Richard F. Freund. *Dynamic mapping of a class of independent tasks onto heterogeneous computing systems*. J. Parallel Distrib. Comput., vol. 59, pg. 107–131, 1999.