Algoritmi paraleli
Curs 6

Vlad Olaru

vlad.olaru@fmi.unibuc.ro

Calculatoare si Tehnologia Informatiei

Universitatea din Bucuresti

Operatii elementare de calcul matricial

- particularizam aspectele generale discutate anterior
- pastram ipotezele de baza asupra constrangerilor de comunicatie (absenta asteptarii in cozi, comunicatie simultana pe toate link-urile incidente unui nod)
- vizam calcule de tipul:

$$x(t+1) = Ax(t) + b \qquad pt.\ t = 0,1,\ \ldots$$

  - produs scalar, inmultire matrice-vector, matrice-matice pt metode Jacobi/Gauss-Seidel
  - calculul minimului unui set de numere, shortest paths, dynamic programming
- caracteristica importanta: dupa iteratie trebuie stocata valoarea vectorului $x(t+1)$ in procesoarele corespunzatoare
- pp. ca timpul initial de stocare pt A, b si $x(0)$ neglijabil, i.e. $O(1)$, daca pp ca nr de iteratii executate e mare

Operatii elementare de calcul matriceal

- **produs scalar**
- inmultire matrice-vector
- inmultire matrice-matrice
- puterea unei matrici

Produs scalar

- **fie $a, b \in R^n$ doi vectori de dimensiune n**
- **calculam: $c = \sum_i a_i b_i$**

- retea cu p procesoare ($n \geq p$); presupunem $k = \frac{n}{p}$
- P_i stocheaza (a_j, b_j) cu $j = (i-1)k + 1, \cdots, ik$
- P_i va calcula $c_i = \sum_{j=(i-1)k+1}^{ik} a_j b_j$
- se acumuleaza rezultatul final intr-un nod arbitrar al unui arbore de acoperire/spanning tree (single node accumulation)
  - ulterior, nodul poate transmite valoarea catre celelalte noduri prin single node broadcast

Produs scalar pe lant liniar

- **fie $a, b \in R^n$ doi vectori de dimensiune n**
- **calculam: $c = \sum_i a_i b_i$**
- timp executie adunare/inmultire = α
- timp transmisie produs partial = β
- alegere optimala pt. root node pe lant liniar e nodul din mijloc, la distanta maxima *floor(p/2)* de orice alt nod (procesor)
- pe topologia de lant liniar, timpul de calcul la root este

$$O\left(k\alpha + \frac{(\alpha+\beta)p}{2}\right) = O\left(\frac{n}{p}\alpha + \frac{(\alpha+\beta)p}{2}\right)$$

Produs scalar pe lant liniar

- **fie $a, b \in R^n$ doi vectori de dimensiune n**
- **calculam: $c = \sum_i a_i b_i$**
- functia de p este convexa, deci se poate minimiza dupa p (ignoram p intreg):

$$p \approx \left(\frac{2\alpha n}{\alpha + \beta}\right)^{\frac{1}{2}}$$

Complexitate in timp optima: $O\left(\frac{n}{p}\alpha + \frac{(\alpha+\beta)p}{2}\right) = \boldsymbol{O\left(n^{\frac{1}{2}}\right)}$

Obs: - p optimal e semnificativ mai mic decat max n

- daca $\beta > (2n - 1)\,\alpha \Rightarrow$ p = 1 , i.e. utilizarea unui singur procesor e optimala pt. a evita costurile mari ale comunicarii interprocesor => trade-off concurenta vs comunicatie

Produs scalar pe topologie hipercub

- **fie $a, b \in R^n$ doi vectori de dimensiune n**
- **calculam: $c = \sum_i a_i b_i$**

- timp executie adunare/inmultire = α
- timp transmisie produs partial = β
- pe topologia hipercub cu arborele de acoperire anterior:

$$O(k\alpha + (\alpha + \beta)\log p) = O\left(\frac{n}{p}\alpha + (\alpha + \beta)\log p\right)$$

Produs scalar pe hipercub

- **fie $a, b \in R^n$ doi vectori de dimensiune n**
- **calculam: $c = \sum_i a_i b_i$**
- functia de p este convexa, deci se poate minimiza dupa p:

$$p \approx \frac{\alpha n}{\alpha + \beta}$$

Complexitate in timp optima: $O\left(\frac{n}{p}\alpha + (\alpha + \beta)\log p\right) = \boldsymbol{O(log(n))}$

- hipercubul este mai eficient: cresterea nr. proc. asigura complexitate mai buna!
- Obs: *nr optimal de procesoare e mult mai mare si timpul de rezolvare a problemei este mai mic fata de lantul liniar, ceea ce denota comunicare mai eficienta pe hipercub*

  - diferenta de performanta intre cele 2 topologii se micsoreaza daca se fac mai multe produse scalare simultan (acumulare multinod)
    - pe lantul liniar timpul de comunicatie nu creste semnificativ, pe hipercub da

Operatii elementare de calcul matriceal

- produs scalar
- **inmultire matrice-vector**
- inmultire matrice-matrice
- puterea unei matrici

Inmultire matrice-vector

- **fie A** $\in \boldsymbol{R^{n \times n}}, \boldsymbol{x} \in \boldsymbol{R^n}$
- **calculam:** $\boldsymbol{c = Ax}$

Memorare pe linii

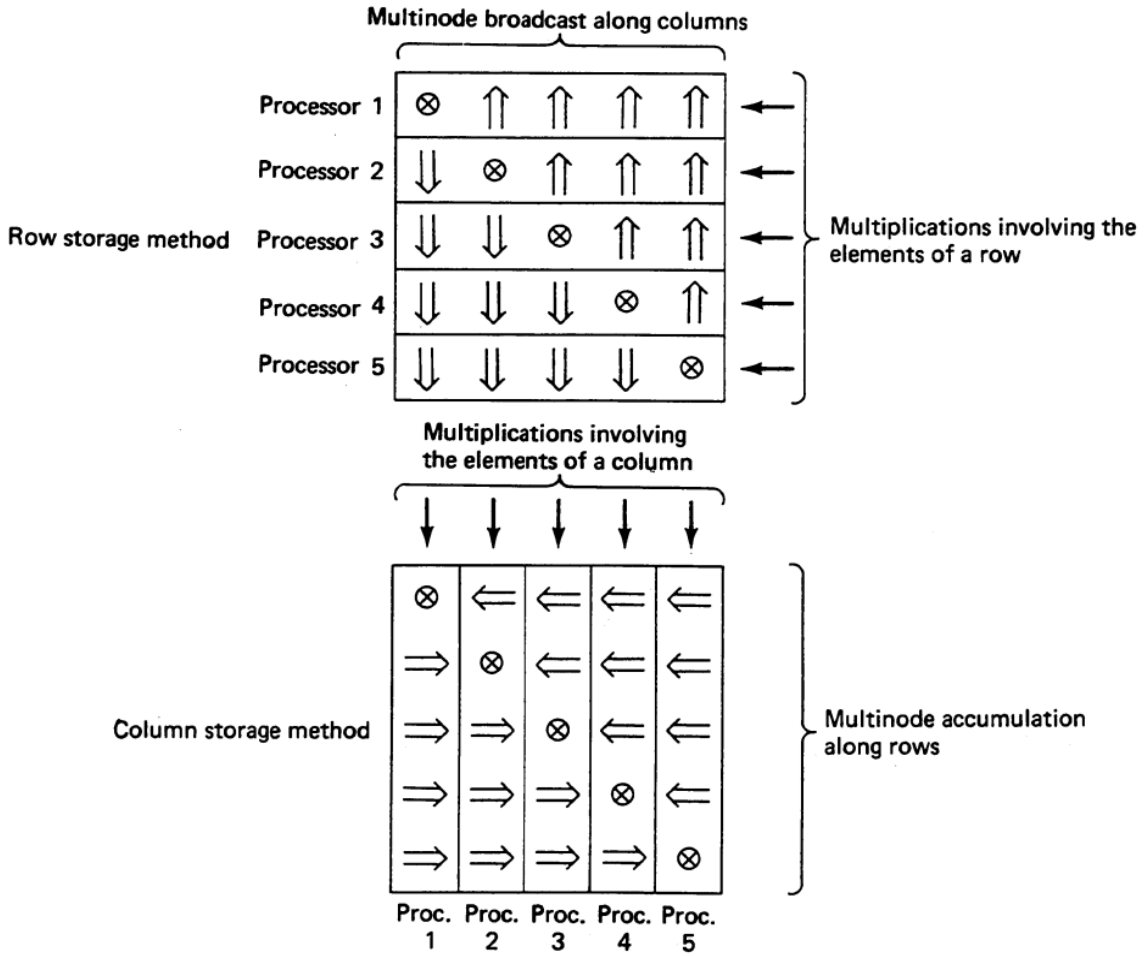
- retea cu p procesoare ($n \geq p$); presupunem $k = \frac{n}{p}$
- P_i stocheaza x si k linii A_j cu $j = (i-1)k + 1, \cdots, ik$
- P_i va calcula k coordonate $c_i = A_i x$
- c_i sunt trimise catre toate celelalte procesoare = > o problema de difuzare generala (broadcast multinod)
- se acumuleaza rezultatul final intr-un nod arbitrar
- Obs: pt memorare pe coloane, P_i va calcula k termeni din suma care defineste fiecare coordonata a lui Ax => acumulare multinod a acestor termeni ($(i-1)k + 1, \cdots, ik$) la nodul i

Obs. inmultire matrice-vector

- convertirea unei matrici din forma memorata pe linii in cea memorata pe coloane implica un schimb total (total exchange)
- pt matrici dense, cele doua metode necesita cam acelasi timp (v. slide urmator), indifferent de topologia de interconectare
  - ambele metode pp un nr egal de inmultiri + acumulare/broadcast multinod care necesita cam acelasi timp pe orice topologie
  - valabil si pt. matrici sparse *simetrice* !
- pt matrici sparse nesimetrice, dificil de ales, depinde de
  - structura matricii
  - topologia de interconectare a procesoarelor
- metoda orientata pe coloane stocheaza un subvector k-dimensional al lui x, metoda orientata pe linii stocheaza intregul vector x

  => stocarea matricii pe coloane e mai eficienta dpdv al consumului de memorie

Dualitate metode



Inmultire matrice-vector pe lant liniar

- **fie A** $\in \boldsymbol{R^{n \times n}}, \boldsymbol{x} \in \boldsymbol{R^n}$
- **calculam:** $\boldsymbol{c = Ax}$

Memorare pe linii

- retea cu p procesoare ($n \geq p$); presupunem $k = \frac{n}{p}$
- P_i stocheaza k linii A_j cu $j = (i-1)k + 1, \cdots, ik$
- P_i va calcula $c_i = A_i x$
- timp executie adunare/inmultire = α
- timp transmisie k numere pe un link intre vecini = $\beta + k\gamma$, unde $\beta, \gamma > 0$
- pe topologia de lant liniar:

$$O(\alpha kn + (p-1)(\beta + k\gamma)) = O\left(\frac{n^2}{p}\alpha + (p-1)(\beta + \frac{n\gamma}{p})\right)$$

Inmultire matrice-vector pe lant liniar

- **fie A** $\in \boldsymbol{R^{n \times n}}, \boldsymbol{x} \in \boldsymbol{R^n}$
- **calculam:** $\boldsymbol{c = Ax}$

Memorare pe linii

- retea cu p procesoare ($n \geq p$); presupunem $k = \frac{n}{p}$
- P_i stocheaza liniile A_j cu $j = (i-1)k + 1, \cdots, ik$
- P_i va calcula $c_i = A_i x$
- functia de p este convexa, deci se poate minimiza dupa p:

$$p \approx n\left(\frac{\alpha - \gamma/n}{\beta}\right)^{\frac{1}{2}}$$

Complexitate optima: $\boldsymbol{O(n)}$

Obs. inmultire matrice-vector pe lant liniar

- pt. n suficient de mare

$$\frac{p}{n} \approx \left(\frac{\alpha - \gamma/n}{\beta}\right)^{\frac{1}{2}}, \text{ adica}$$

$$k = \frac{n}{p} \approx (\beta/\alpha)^{1/2} \text{ nr de linii per procesor este constant}$$

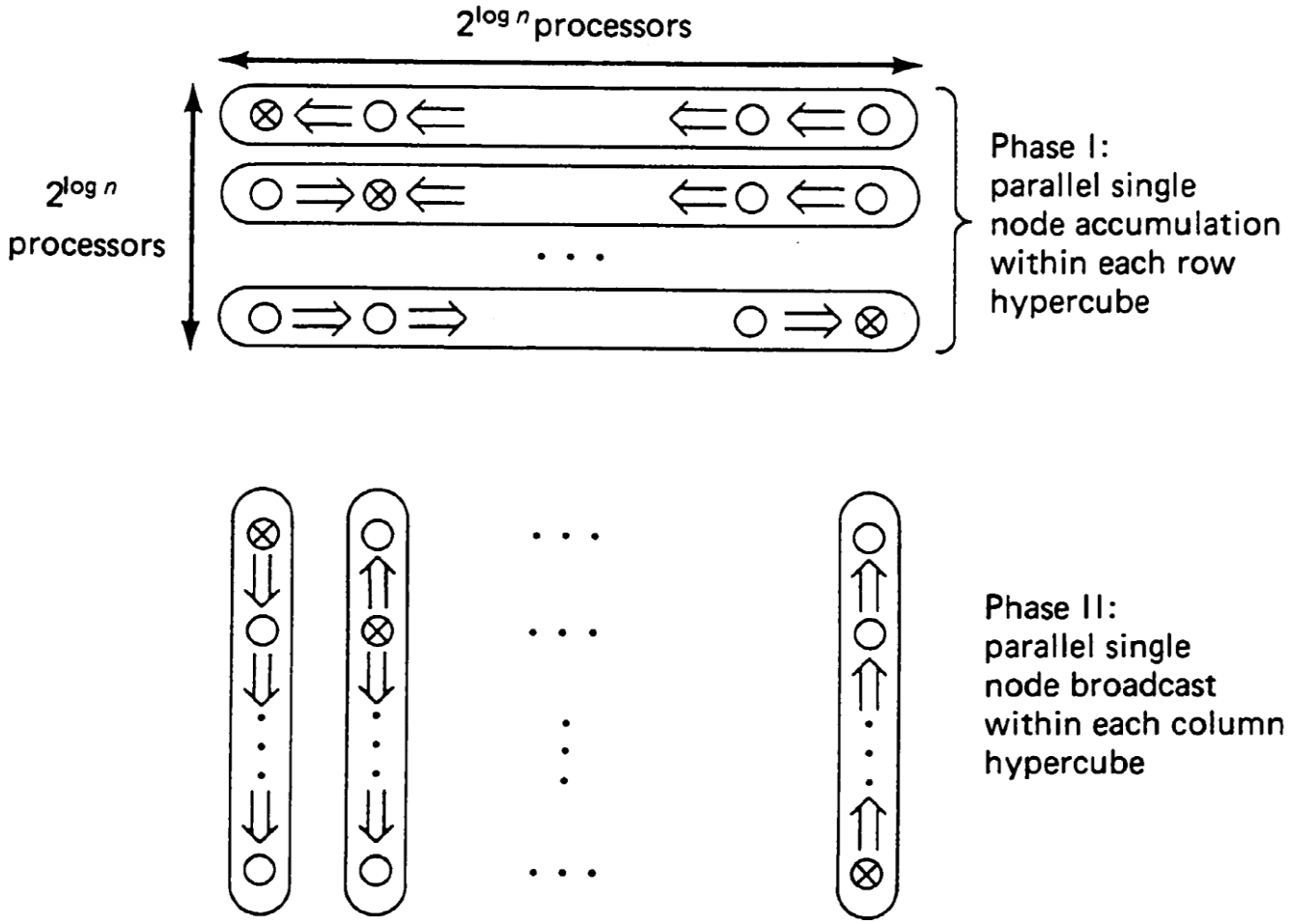
- timpul total optimal creste liniar cu n, i.e acelasi ordin de crestere ca si atunci cand neglijam costul comunicatiei !
- pt lant liniar mapat pe topologii mai performante (hipercub): timpul de rulare nu-l poate depasi pe cel de pe lantul liniar
- pe de alta parte, timpul nu poate scadea nici sub timpul solutiei care neglijeaza costurile comunicatiei (i.e., comunicatie instantanee)

=> pt. orice topologie de interes practic, produsul $\boldsymbol{Ax}$ folosind un nr optimal de procesoare p creste liniar cu n, **cand** $\boldsymbol{p \leq n}$

Inmultire matrice-vector pe hipercub cu $p \geq n$ procesoare

- folosim tot memorare pe linii, dar $p \geq n$
- mai exact, pt. $p = n^2$ procesoare si comunicatie instantanee, timpul de executie devine $O(log(n))$
- pe lant liniar e imposibil de atins aceasta limita pt ca produsul scalar (mai simplu decat inmultirea matrice-vector) este $O\left(n^{\frac{1}{2}}\right)$
- solutie: matrice n x n, mapata pe hipercub ca la grid (fiecare linie si coloana sunt noduri plasate intr-un hipercub de dimensiune n)
- initial, procesorul (i, j), contine elementul corespunzator din A si componenta j a lui x
  - la final, contine componenta j a produsului Ax
- algoritm in doua faze, fiecare de $O(log(n))$
  - acumulare pe fiecare linie, nodul diagonal i calculeaza coordonata i a produsului Ax
  - nodul diagonal j difuzeaza procesoarelor de pe coloana j coordonata j a produsului Ax acumulata in pasul anterior

Inmultire matrice-vector pe hipercub



Inmultire de matrici

- **fie $\mathbf{A}, \mathbf{B} \in \boldsymbol{R^{n \times n}}$**
- **calculam: $\mathbf{C} = \boldsymbol{AB}$**

$$C_{ij} = \sum_k A_{ik} B_{kj} = A_i B^j$$

- pentru $i = 0 : n - 1$
  - pentru $j = 0 : n - 1$
    - pentru $k = 0 : n - 1$
      - $C_{ij} = C_{ij} + A_{ik} B_{kj}$

- Complexitate seriala: 1 procesor $\rightarrow O(n^3)$

Inmultire matrice-matrice pe grid bidimensional

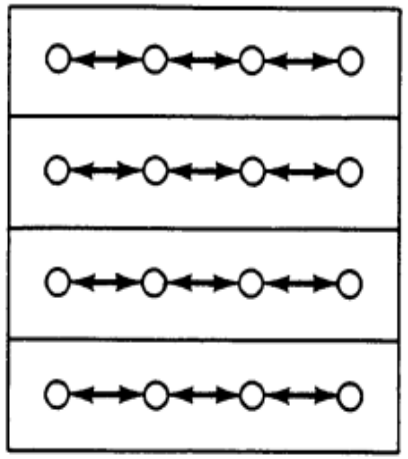
- **fie $\mathbf{A}, \mathbf{B} \in \boldsymbol{R^{n \times n}}$**
- **calculam: $\mathbf{C} = \boldsymbol{AB}$**
- folosim un grid de $p = n^2$ procesoare
- *A(i,j) si B(i,j)* stocate in procesorul (i, j), iar la final se stocheaza si *C(i,j)*
- algoritm in trei faze fiecare de complexitate $O(n)$
  - global, aceeasi complexitate ca la algoritmul care foloseste n^2 procesoare si comunicare instantanee !

Inmultire matrice-matrice pe grid bidimensional

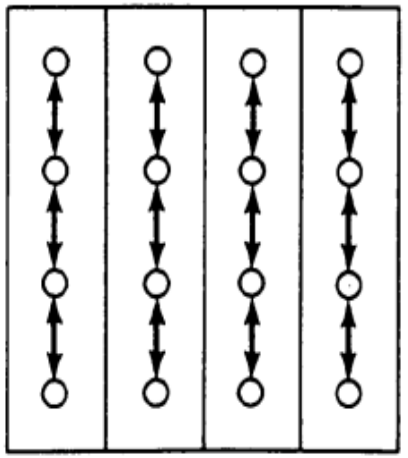
- Faza 1: fiecare procesor *(i,j)* trimite *A(i,j)* procesoarelor de pe linia i => n broadcast-uri multinod pe fiecare linie, adica $O(n)$
- Faza 2: fiecare procesor *(i,j)* trimite *B(i,j)* procesoarelor de pe coloana j => n broadcast-uri multinod pe fiecare coloana, adica $O(n)$
- dupa 1 & 2, procesorul *(i,j)* stocheaza valorile *A(i,m)* si *B(m,j) si* poate calcula

  $\sum_{m=1}^{n} a_{im} b_{mj}$ in paralel in $O(n)$
- complexitate totala folosind n^2 procesoare:

  $O(n)$
- Obs: 1 & 2 pot rula in paralel daca e posibila comunicarea simultana pe toate link-urile/arcele incidente unui nod



Phase 1



Phase 2

Calculul puterii unei matrici

- **fie A** $\in \boldsymbol{R^{n \times n}}$
- **calculam:** $\boldsymbol{A^k}$
- folosim un grid patrat de $p = n^2$ procesoare
- *A(i,j)* stocat in procesorul (i, j), iar la final se stocheaza elemental *(i,j)* al $\boldsymbol{A^k}$
- solutie: ridicare succesiva la patrat daca $k = 2^m$

$$A^k = A^2 \, x \, A^2 \ldots$$

- pt k $\neq 2^m$, calculam ca la cursurile trecute (calculam succesiv A^2, A^4, si inmultim intre ele rezultatele intermediare pt a obtine urmatoarea putere a matricii)
- complexitate $O(n \log k)$

Inmultire de matrici cu n^3 procesoare

- **fie $\mathbf{A}, \mathbf{B} \in \boldsymbol{R^{n \times n}}$**
- **calculam: $\mathbf{C} = \boldsymbol{AB}$**
- folosim $p = n^3$ procesoare
- inmultire matrice-vector in $O(\log(n))$ pe hipercub cu n^2 procesoare
- Obs: coloanele lui C contin produse matrice-vector de tip Ab_i, unde b_i sunt coloanele matricii B

=> aceste produse se pot calcula in $O(\log(n))$ cu n^3 procesoare

Inmultire de matrici cu n^3 procesoare

- pp. $n = 2^d$ si un hipercub de n^3 procesoare aranjate intr-un grid n x n x n
- pp procesorul (i, j, k) contine a_{ij} si b_{jk} si la final c_{ij} si c_{jk} unde $c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$
- algoritm in 3 faze de durata $O(\log(n))$ fiecare

1: c_{ik} este acumulat in procesorul (i, i, k) din

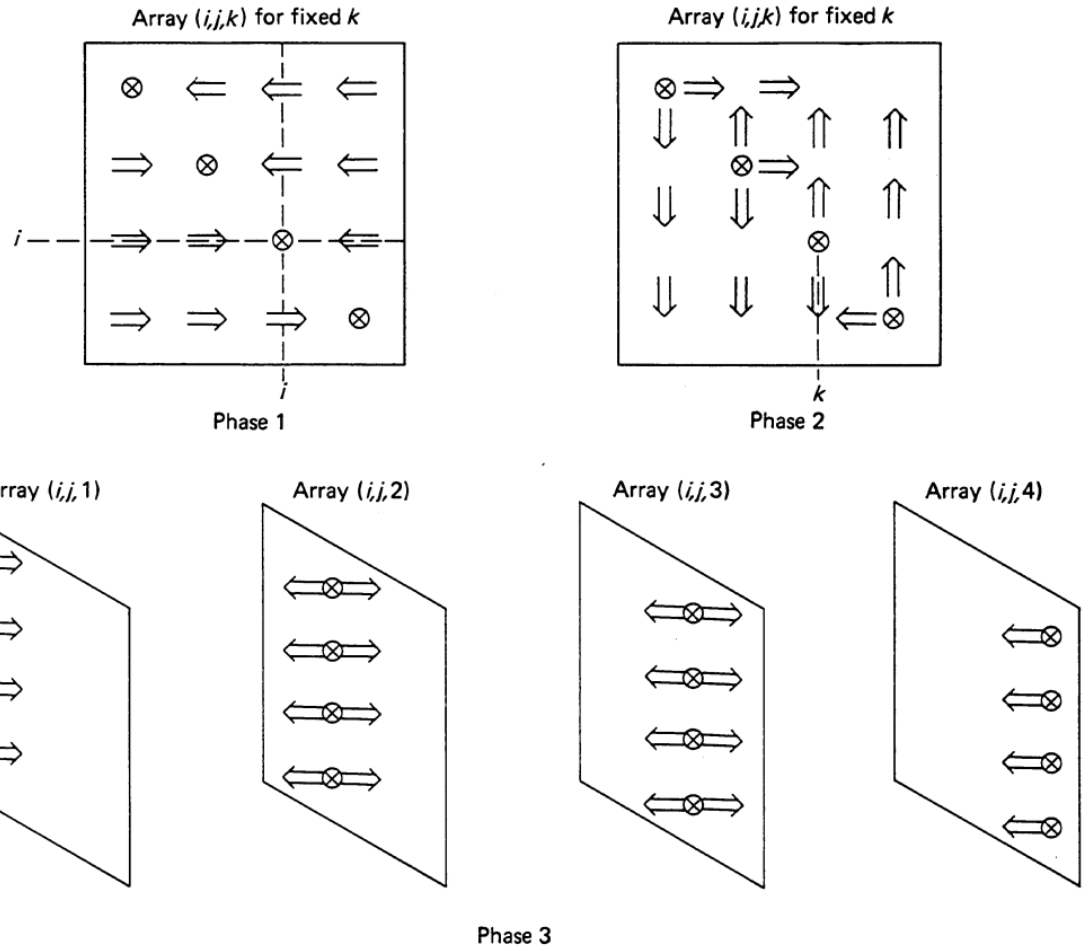
gridul (i, j, k), $j=1,\ldots n$

2: fiecare (i,i,k) trimite prin bcast c_{ik} in gridul

(j, i, k), $j=1,\ldots n$ ca si procesorului (i, k, k)

3: procesorul (i, j, j) trimite c_{ij} prin bcast in gridul (i, j, k), $k=1,\ldots n$

- complexitate hipercub (n^3 *procesoare*): $O(\log(n))$



Calcului puterii unei matrici cu n^3 procesoare

- **fie A** $\in \boldsymbol{R^{n \times n}}$
- **calculam:** $\boldsymbol{A^k}$
- folosim un hipercub cu $p = n^3$ procesoare
- $O(\log k)$ multiplicari ale puterilor lui A
- daca folosim algoritmul anterior de inmultire de matrici => $O(log\, n\, log\, k)$
- Obs: aceeasi performanta ca la cursurile trecute cand consideram comunicatia instantanee !

Timpi de rulare optimala operatii matriciale

Problem	Time	Corresponding Topology
Inner Product	$O(\log n)$	Hypercube w/ $p = n$ processors
Matrix–Vector Multiplication	$O(n)$	Linear Array w/ $p = n$ processors
Matrix–Vector Multiplication	$O(\log n)$	Hypercube w/ $p = n^2$ processors
Matrix–Matrix Multiplication	$O(n)$	Mesh w/ $p = n^2$ processors
kth Power of a Matrix	$O(n \log k)$	Mesh w/ $p = n^2$ processors
Matrix–Matrix Multiplication	$O(\log n)$	Hypercube w/ $p = n^3$ processors
kth Power of a Matrix	$O\big((\log n)(\log k)\big)$	Hypercube w/ $p = n^3$ processors

Concluzii operatii matriciale

- folosirea unor retele de interconectare potrivite (eg, hipercub) permite calculul unor operatii matriciale de baza in acelasi ordin de timp ca si atunci cand comunicatiile sunt instantanee
- nu inseamna ca CP e neglijabila !
- inseamna ca T_{comm} creste cu dimensiunea problemei cu o rata care nu e mai mare decat rata de crestere a T_{comp}

Problema sincronizarii in algoritmii paraleli

- coordonarea activitatilor se face adesea in *faze*
- in fiecare faza, procesorul executa calcule care depind de rezultate calculate in faze anterioare
- in general insa, nu exista sincronizare in timp a executiei procesoarelor in cadrul aceleiasi faze
- procesoarele interactioneaza in general la sfarsitul unei faze
- asemenea algoritmi se numesc *sincroni*, spre deosebire de cei *asincroni* care nu folosesc notiunea de faze si in care coordonarea procesoarelor e mult mai putin stricta
- cadrul general de discutie ramane cel al sistemelor message-passing
  - unele idei se pot adapta sistemelor cu memorie partajata

Algoritmi paraleli sincroni

- fie o secventa de operatii impartita in segmente numite *faze*
- calculele din fiecare faza sunt impartite intre procesoarele din sistemul de calcul
- in faza *t*, procesorul *i* face calcule dependente de problema, folosind si informatie primita de la alte procesoare in fazele anterioare *1,2,…, t-1*
- apoi procesorul *i* trimite informatii catre un subset de procesoare *P(i,t)* si procesul se repeta la faza *t+1*
- presupunere implicita: procesoarele ruleaza independent unele de altele in cadrul aceleiasi faze iar sincronizarea lor relativa e irelevanta

Exemplu de algoritm sincron

- metoda iterativa

$$x_i(t+1) = f_i(x_1(t), \dots, x_n(t)) \quad i = 1, \dots, n \quad \text{unde } x_i \text{ e actualizat de procesorul } i$$

- faza asociata cu timpul t
- procesorul i actualizeaza variabila x_i si trimite valoarea actualizata catre toate procesoarele j ale caror calcule depind de x_i =>

$$P(i, t) = \{j \mid (i,j) \text{ arc in graful de dependenta}\}$$

- un asemenea algoritm distribuit s.n. algoritm *sincron*
  - echivalent dpdv matematic cu un algoritm cu ceas global
  - toate procesoarele pornesc simultan o faza
  - la sfarsitul fazei mesajele sunt receptionate simultan de catre toate procesoarele
- pt. a implementa un asemenea algoritm intr-un sistem de calcul inerent asincron e nevoie de *mecanisme de sincronizare*
  - algoritm prin care fiecare procesor poate detecta sfarsitul fiecarei faze
  - sincronizare globala vs. locala

Sincronizare globala

- fiecare procesor trebuie sa poata detecta cand s-au primit toate mesajele trimise in cadrul unei faze
  - abia apoi poate porni calculul fazei urmatoare
- metoda simpla: timeout
  - nu exista ceas global, doar ceasuri locale procesoarelor
  - pp. se cunoaste limita max T_p a timpului de executie a unei faze pe orice procesor i si pt a trimite mesajele sale catre celelalte procesoare
  - pp. T_f= intervalul de timp in care procesoarele au inceput prima faza

  $\Rightarrow$ sincronizarea se produce efectiv daca procesorul i porneste faza k dupa $k(T_f + T_p)$ unitati de timp de la pornirea primei faze

  $\Rightarrow$ problema: limitele T_p si T_f pot fi fie prea conservatoare fie necunoscute

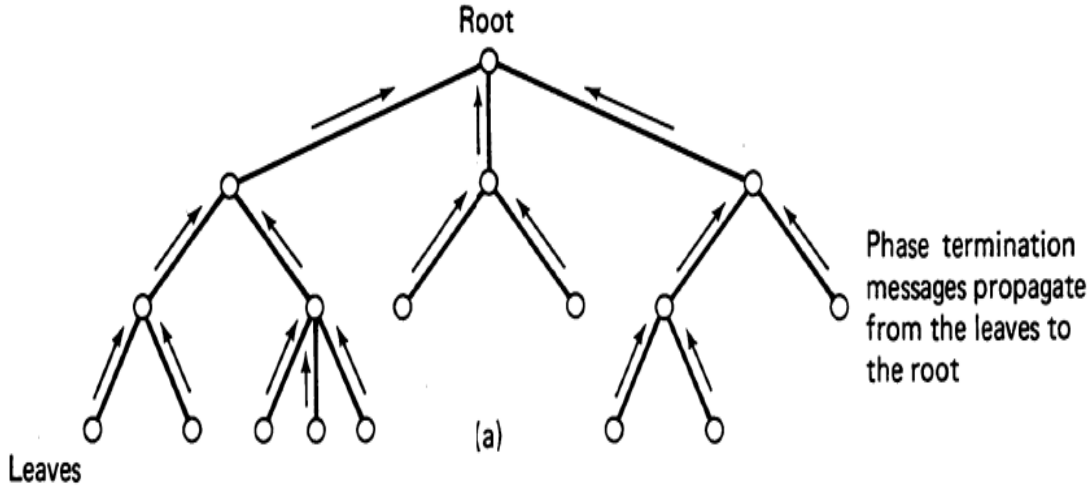
Sincronizare globala cu mesaje de terminare a fazei

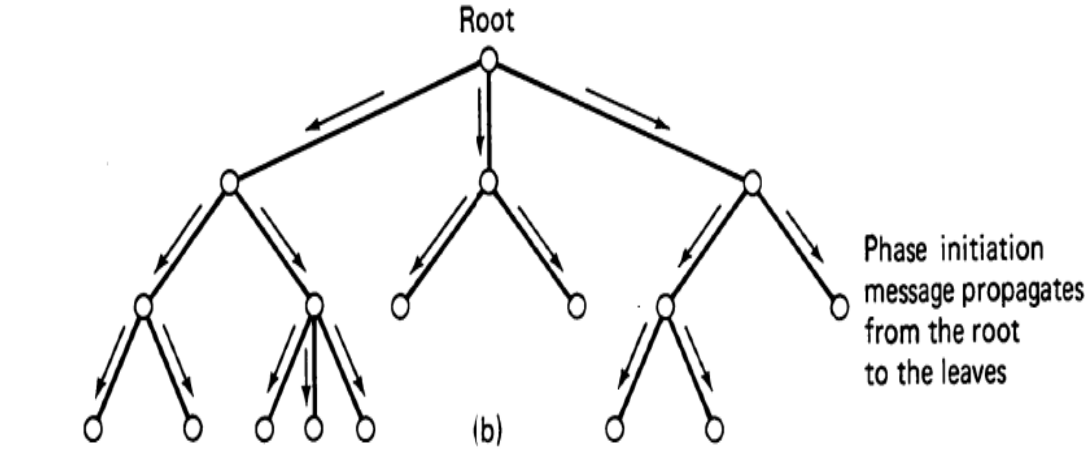
- fiecare procesor trimite tuturor celorlalte procesoare un *mesaj de terminare a fazei* cand stie ca toate mesajele sale pt o anumita faza au fost receptionate
- pp implicit ca receptia fiecarui mesaj este notificata transmitatorului (ACK)
  - astfel, fiecare procesor stie ca mesajele sale au fost receptionate si poate initia mesajul de terminare a fazei
- cand un procesor a trimis mesajul de terminare a fazei si a primit toate mesajele de terminare a fazelor celorlalte procesoare initiaza noua faza
- in sisteme cu memorie partajata, acest mecanism e implementat cu variabile speciale vizibile tuturor procesoarelor: spinlocks, semafoare, monitoare, etc
  - valorile acestor variabile indica procesoarelor ca faza s-a terminat si se poate continua cu urmatoarea faza

Sincronizare globala cu mesaje de terminare a fazei in sisteme message-passing

- mecanismul se implementeaza folosind un arbore de acoperire (spanning tree) al retelei de interconectare a procesoarelor
- mesajele de terminare a fazei se strang la radacina arborelui de acoperire, pornind de la frunze
- cand radacina a primit toate mesajele, initiaza un broadcast catre toate procesoarele din arborele de acoperire cu un mesaj de incepere a noii faze

=> complexitate $\Theta(r)$, unde r este diametrul retelei





Sincronizare locala

- daca un procesor stie ce mesaje trebuie sa primeasca in fiecare faza, dupa ce le-a primit poate porni noua faza
- procesorul j incepe faza $t+1$ cand a primit toate mesajele din faza t trimise de procesoarele i din multimea $\{ i \mid j \in P(i,t)\}$
- obs: procesorul nu trebuie sa stie daca mesajele trimise in faza t au fost receptionate (nici macar pt mesajele sale)
  - nu e nevoie sa se astepte receptia lor si notificarea (confirmarea) corespunzatoare
- cand fiecare procesor i cunoaste $\{ i \mid j \in P(i,t)\}$, sincronizarea locala nu genereaza penalizari de comunicare mai mari decat sincronizarea globala
  - cel mai adesea, penalizarea comunicarii e mai mica
  - e.g., cand timpul de transmisie al mesajelor e comparabil cu timpul de calcul al fazei, pt. ca sincronizarea globala are nevoie de mesaje ACK
  - chiar fara mesaje ACK, diferenta dintre timpii de executie pt cele doua mecanisme de sincronizare creste cu nr de faze, datorita variabilitatii timpilor de calcul si transmisie de mesaje pt fiecare procesor in fiecare faza

Algoritmi paraleli asincroni

- penalizarea de comunicare (si implicit timpul total de executie) se poate reduce substantial folosind implementari asincrone
- dat fiind un algoritm distribuit, pt fiecare procesor exista distinct
  - timpi asociati cu executia operatiilor
  - timpi pentru transmisia de mesaje catre alte procesoare
  - timpi pentru receptia de mesaje de la alte procesoare
- intr-un algoritm sincron, toti acesti timpi pot fi considerati in sens matematic echivalenti unui algoritm cu acesti timpi fixati *a priori*
- cand acesti timpi (si ordinea lor) pot varia semnificativ intre doua rulari ale algoritmului paralel avem de-a face cu un algoritm *asincron*
- in cazul sistemelor distribuite, algoritmul paralel/distribuit poate fi vazut ca o colectie de algoritmi locali care se executa pe procesoare diferite si schimba ocazional date

Algoritmi paraleli asincroni locali

- intr-un algoritm sincron, algoritmii locali
  - fie se sincronizeaza global folosind un ceas global
  - fie asteapta momente predeterminate in timp pt ca datele sa devina disponibile
    - fara a se cunoaste dinainte timpul de executie al operatiilor locale
- intr-un algoritm asincron, algoritmii locali nu asteapta niciun fel de date sa devina disponibile, ci continua sa calculeze avand la dispozitie doar datele de la un moment dat
- caz extrem: algoritmii asincroni tolereaza schimbari in datele problemei sau in arhitectura sistemului distribuit
  - ex: retele de calculatoare, unde link-uri si noduri se pot defecta si reveni in executie
  - retelele mobile isi schimba dinamic topologia
  - chiar daca aceste schimbari nu sunt frecvente, timpi de executie mari ai algoritmilor pot determina o probabilitate neneglijabila de aparitie a schimbarilor de topologie

Algoritmi asincroni locali (cont.)

- dificultati associate tolerantei la defecte si schimbarilor de arhitectura
  - toate nodurile trebuie notificate asupra defectelor si reparatiilor, pt ca aceste aspect influenteaza algoritmul (ex: rutare)
    - dificil pt ca informatia despre defecte este transmisa pe link-uri care se pot defecta si ele
  - algoritmii toleranti la asemenea fenomene trebuie sa fie adaptivi (sau minimal sa poata abandona executia si sa reporneasca)
    - netrivial, pt ca defecte suplimentare pot apare in timpul repornirii
    - algoritmii asincroni se pot reporni mai usor pt ca permit mai multa flexibilitate in alegerea conditiilor intiale
- Q: pot algoritmii asincroni reduce penalizarea comunicarii si pe cale de consecinta sa imbunatateasca timpii de rulare ai algoritmilor paraleli?

Metode iterative asincrone

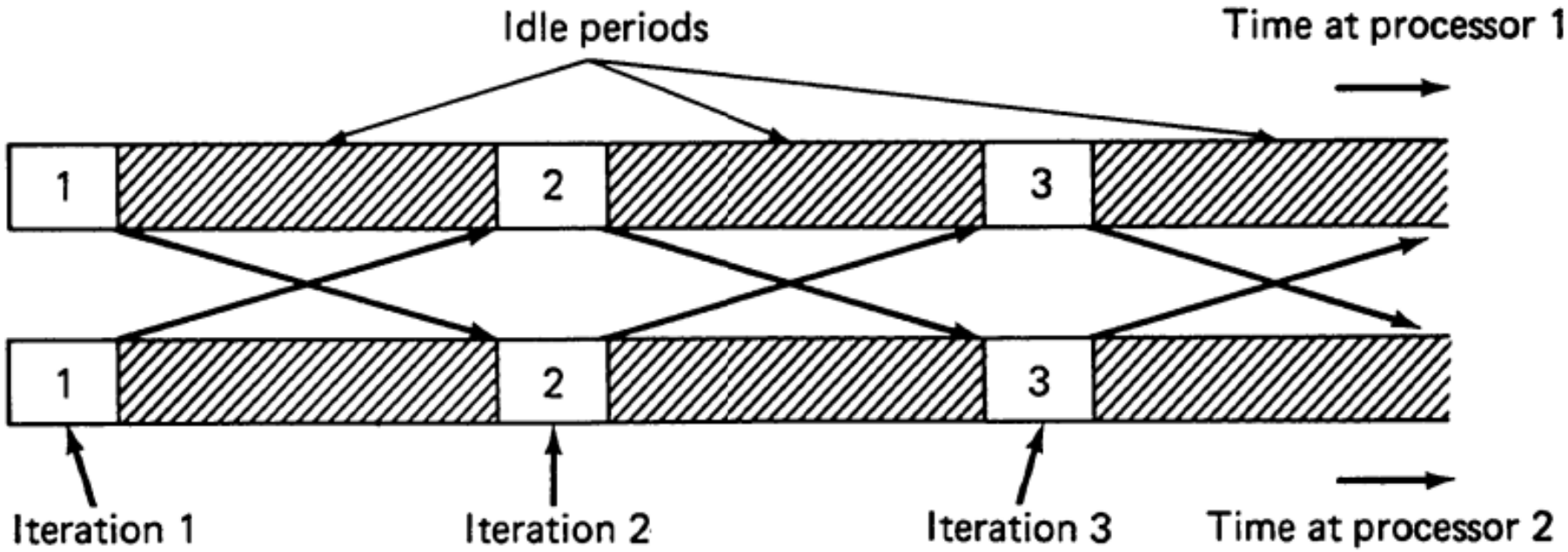
- fie un sistem cu n procesoare si o iteratie de punct fix n-dimensionala

$$x_i = fi(x_1, x_2, \dots, x_n) \qquad i = 1,2, \dots n$$

- procesorul i actualizeaza variabila x_i
- o implementare sincrona pp ca procesorul i executa actualizarea k doar dupa ce primeste rezultatele celor $(k-1)$ actualizari de la procesoarele din graful de dependente
- aceasta ineficienta inerenta solutiei alese se mai numeste si *penalizare de sincronizare*
- depinde de doi factori: viteza canalelor de comunicatie folosite si respectiv a procesoarelor implicate in calcul

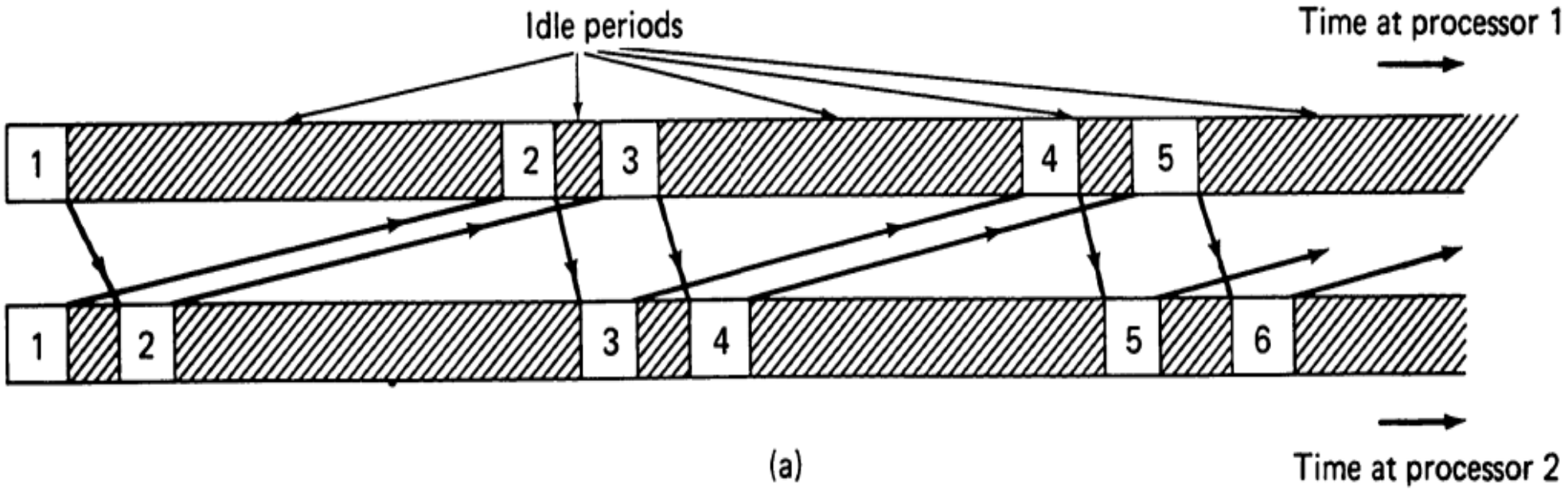
Influenta vitezei canalelor de comunicatie si a procesoarelor

- cand procesorul i actualizeaza variabila x_i trebuie sa astepte mesajele celorlalte procesoare



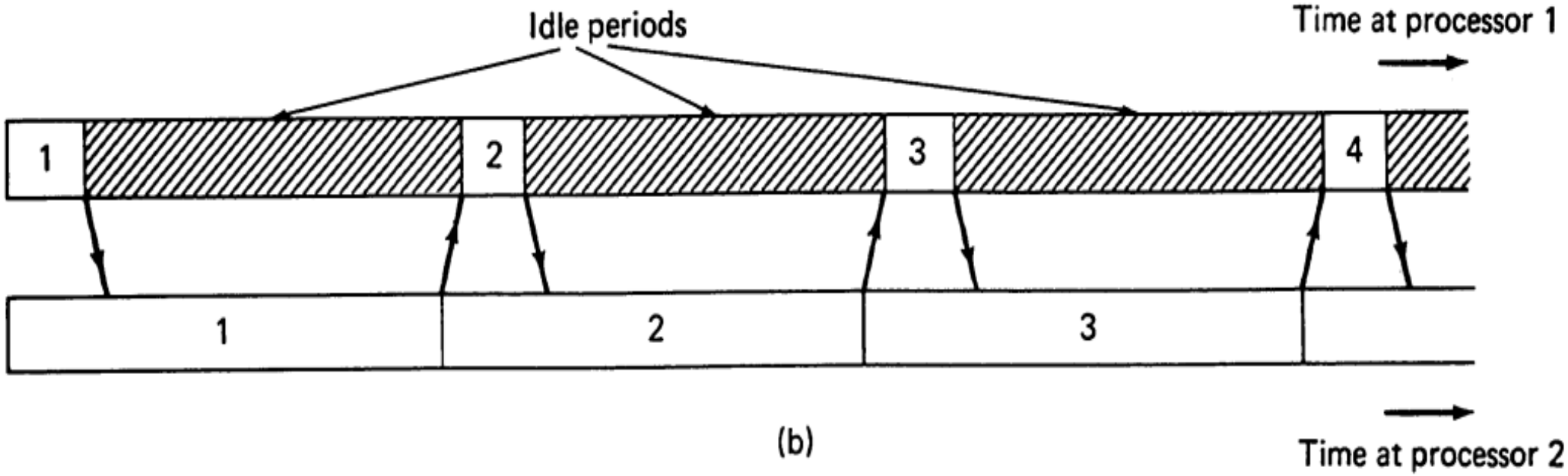
Influenta vitezei canalelor de comunicatie si a procesoarelor

- un canal de comunicatie lent poate incetini progresul intregului algoritm



Influenta vitezei canalelor de comunicatie si a procesoarelor

- procesoarele rapide sau cele care au incarcare usoara per iteratie trebuie sa astepte terminarea iteratiilor de pe procesoarele lente sau incarcate
- astfel, progresul algoritmului este dictat de cel mai lent procesor



Penalizarea de sincronizare

- cand e influentata de prezenta canalelor lente de comunicatie, contribuie la penalizarea comunicarii CP
- cand e influentata de vitezele de executie diferite ale procesoarelor, contribuie la o pierdere de eficienta prin dezechilibrul incarcarii procesoarelor (*poor load balancing*), chiar si cand comunicatiile sunt instantanee

Varianta asincrona a algoritmului

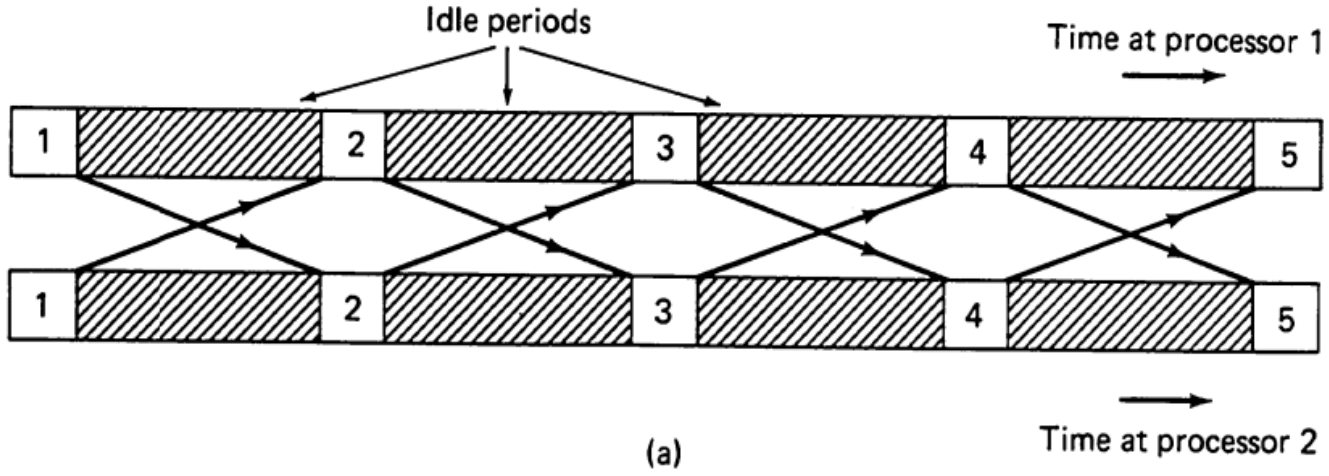
- procesorul i executa actualizarea k folosind *unele* dintre rezultatele actualizarilor anterioare, *nu neaparat cele mai recente* !
  - ex: procesorul i excuta actualizarea k folosind rezultatul actualizarii $(k + 10)$ a procesorului j in vreme ce acesta executa actualizarea $(k + 100)$ folosind rezultatul actualizarii $(k - 10)$ a procesorului i
- vitezele de executie si comunicatie pot diferi la procesoare diferite iar intarzierile de comunicare pot fi substantiale
  - mai mult, ele pot evolua impredictibil pe masura ce algoritmul progreseaza
- nici macar ordinea mesajelor pe link-urile de comunicatie nu e garantata
  - un procesor poate folosi actualizarea k de la un alt procesor la un moment dat si rezultatele actualizarii $(k - 10)$ ale aceluiasi procesor la un moment ulterior
- si conditiile initiale ale unui procesor pot fi flexibile
  - variabilele x_i de pe procesoare diferite pot avea valori diferite pt acelasi i la momentul inceperii algoritmului

Consecinte algoritm asincron

- algoritmii asincroni pot reduce penalizarea de sincronizare indusa de
  - procesoare rapide care asteapta procesoarele lente sa livreze actualizari
  - canale de comunicatie lente in a livra mesaje
- procesoarele rapide pot executa mai multe iteratii fara sa astepte cele mai recente rezultate ale celorlalte procesoare
- pericol: iteratiile bazate pe informatie veche pot fi ineficiente si pot afecta convergenta
- cand functioneaza, algoritmii asincroni reduc penalizarea de sincronizare si genereaza solutii mai rapide
- in schimb, creste potential volumul de comunicatie comparativ cu solutiile sincrone
- dezavantaj: afecteaza convergenta pe care solutiile sincrone ar putea-o avea, fiind nevoie sa se limiteze intarzierile de comunicatie pt a garanta convergenta
- in orice caz, analiza algoritmilor sincroni este mult mai dificila decat cea a algoritmilor sincroni

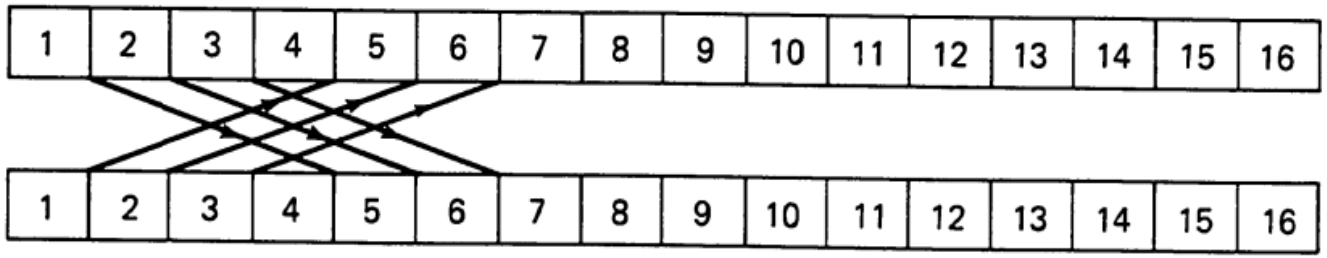
Comparatie algoritm asincron vs sincron

(a) algoritm sincron
- sagetile indica momentul primirii mesajelor, zonele hasurate indica procesor idle, numerele corespund perioadelor de actualizari
- perioadele de asteptare sunt de trei ori mai mari decat cele de calcul in exemplul nostru



(a)

(b) algoritm asincron
- procesoarele pot executa mai multe iteratii pe unitatea de timp (de 3 ori mai multe in exemplul nostru)
- nu e nevoie sa se astepte receptia mesajelor



(b)

Avantajele algoritmilor asincroni pt metode iterative

- Jacobi

$$x_i(t+1) = f_i(x_1(t), \dots, x_n(t)) \quad i = 1, \dots, n$$

  - potrivita pt implementare paralela sincrona

- Gauss-Seidel

$$x_i(t+1) = f_i\big(x_1(t+1), \dots, x_{i-1}(t+1), x_i(t), \dots, x_n(t)\big)$$

  - inerent secventiala (se poate paraleliza pt grafuri de dependenta sparse)

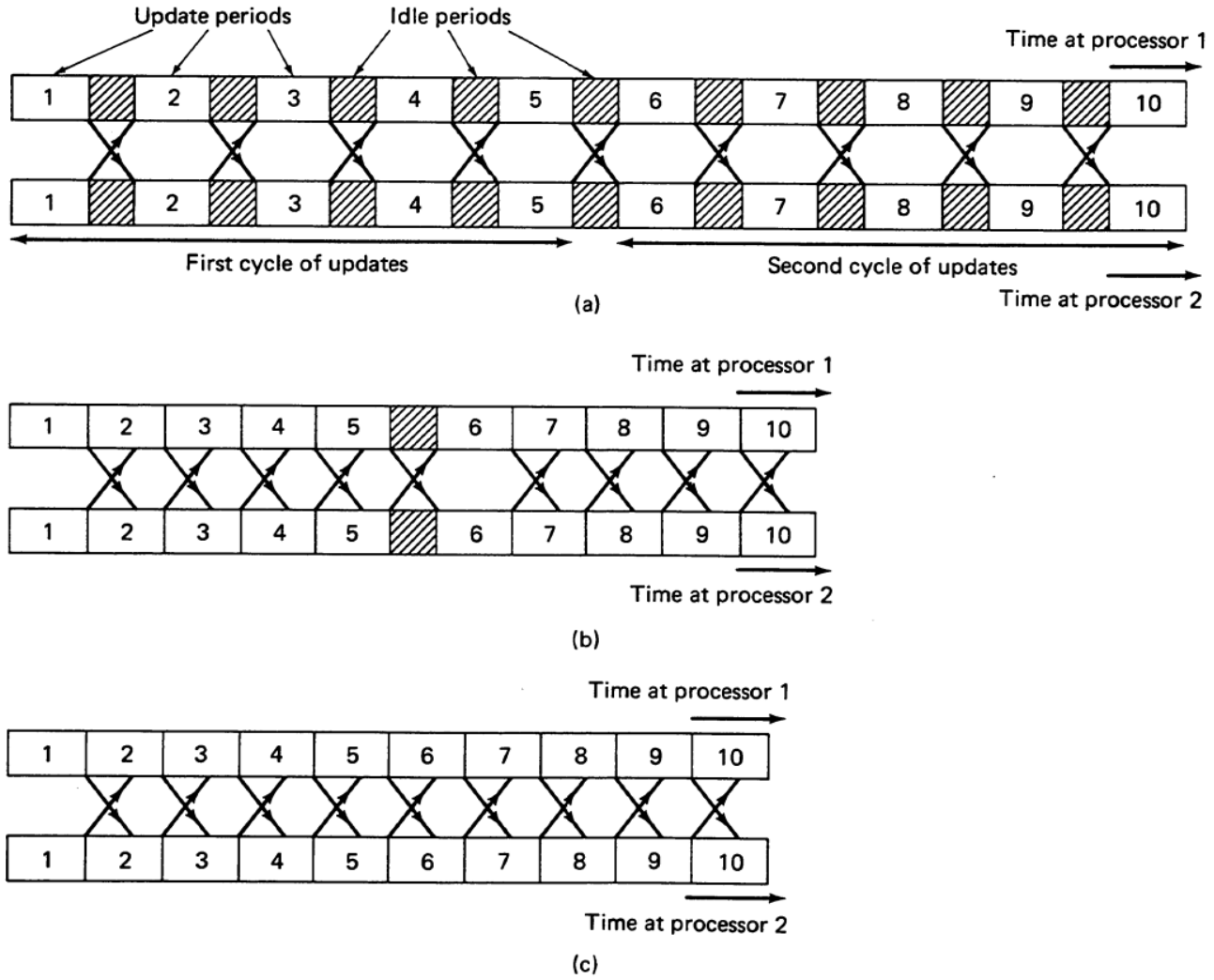
- in general, Gauss-Seidel converge mai rapid decat Jacobi

- in general, convergenta se poate accelera prin incorporarea valorilor actualizate ale variabilelor in valori urmatoare pt alte variabile cat mai devreme cu putinta
  - intr-un algoritm asincron exista potentialul de a se putea incorpora valorile actualizate in calcul mai repede decat intr-un algoritm sincron

- deci este posibil ca un algoritm asincron sa foloseasca avantajele de convergenta ale metodei Gauss-Seidel, fara a sacrifice potentialul de paralelism a metodei Jacobi

Gauss-Seidel asincron vs Jacobi

- se poate gandi o implementare asincrona Gauss-Seidel mai rapida care sa pastreze potentialul de paralelism din Jacobi

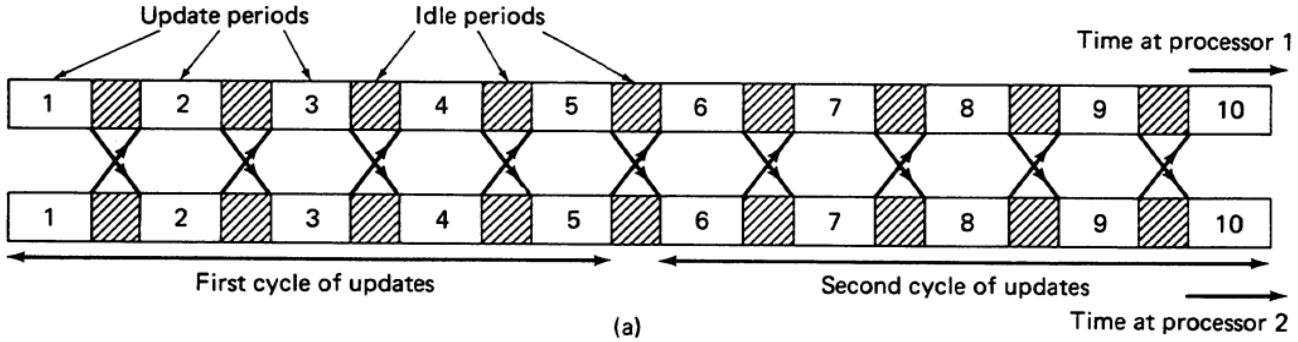


Gauss-Seidel asincron vs Jacobi

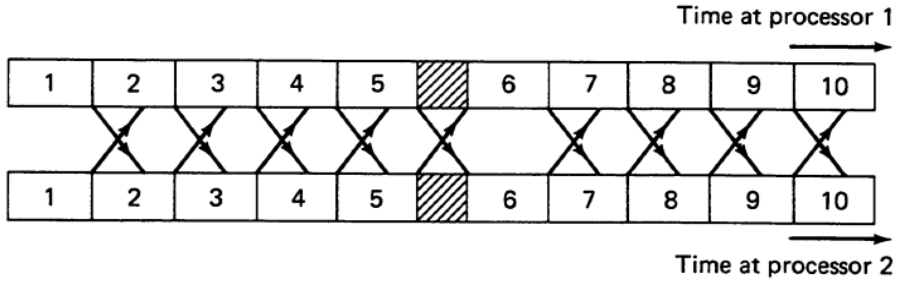
- 2 procesoare care actualizeaza 5 variabile fiecare

(a) algoritm sincron care incearca sa incorporeaze informatia noua cat mai repede posibil

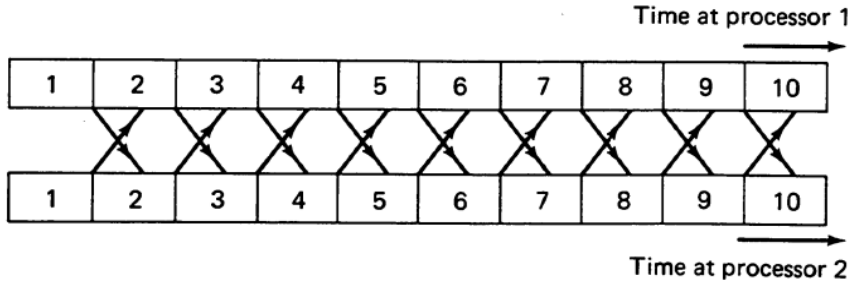
Penalizarea de sincronizare este mare.



(a)



(b)



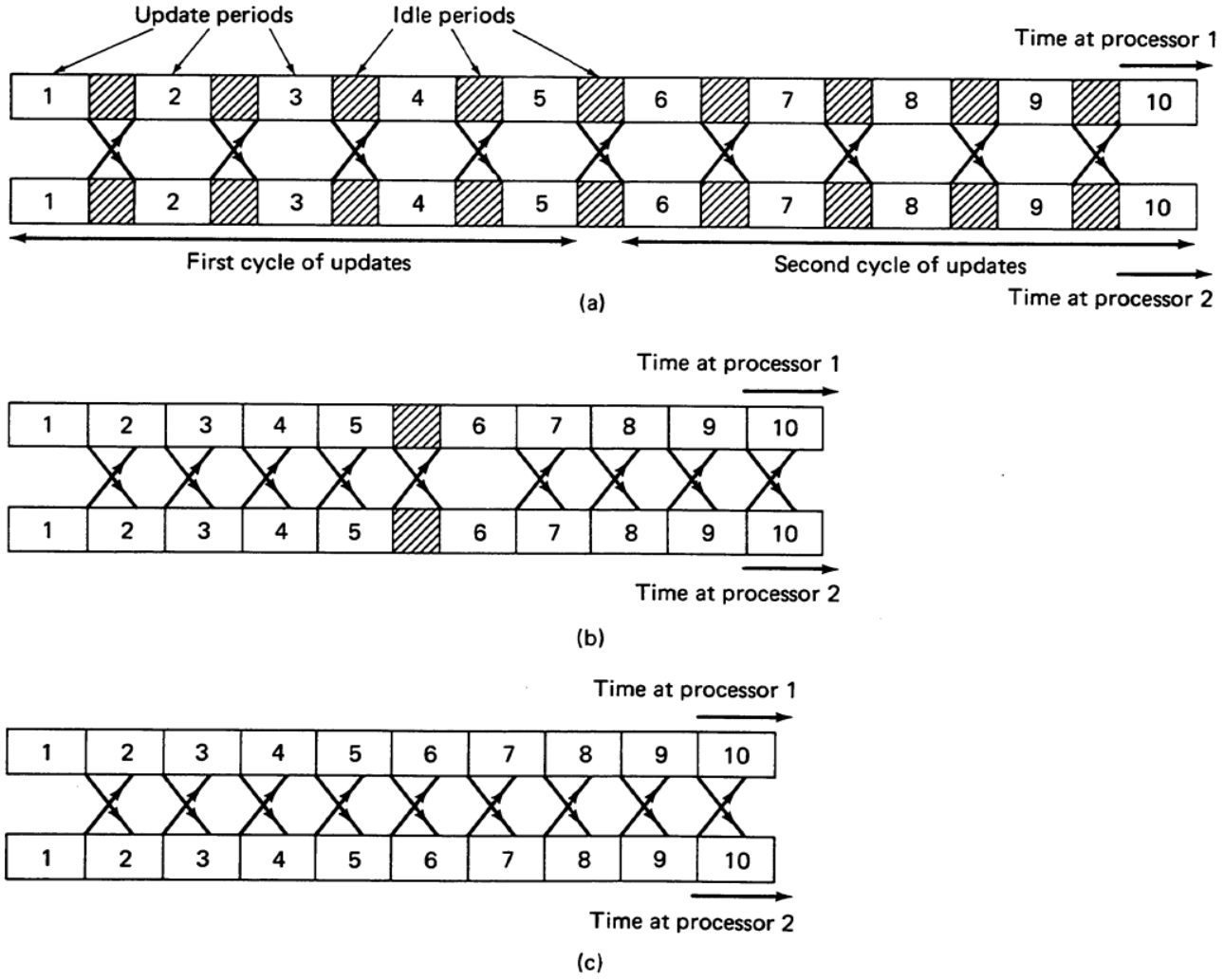
(c)

Gauss-Seidel asincron vs Jacobi

(b) algoritm sincron care reduce penalizarea de sincronizare prin pipelining-ul calculului cu comunicatia

- actualizarile per procesor se iau in considerare la fiecare 5 pasi

=> metoda Jacobi-like



Gauss-Seidel asincron vs Jacobi

(c) algoritm asincron, fara penalizare de sincronizare; rezultatul fiecarei actualizari e luat in calcul de alt procesor dupa o intarziere de o actualizare

- in acest exemplu toate actualizarile necesita acelasi timp
- penalizarile sunt datorate exclusiv comunicatiei
- Obs: timpi variabili de actualizare a variabilelor pot creste suplimentar penalizarea de sincronizare