V. Modele ale calculatoarelor paralele – dupa PRAM si modelele timpurii

Continut

- Arhitecturi dataflow
- Arhitecturi sistolice
- Modelul circuit
- Modelul graf
- LogP
- LogGP,
- Paradigma de transmitere a mesajelor
- Nivele de paralelism

Arhitecturi de flux de date

- Scop:
  - fac ca aspectele unui computer paralel să fie explicite la nivelul mașinii:
    - fără constrângeri artificiale care limitează paralelismul disponibil în program
- Idea:
  - Programul este reprezentat de un graf al dependențelor esențiale de date:
    - mai degrabă decât ca o colecție fixă de fire de control secvențiate explicit.
  - O instrucțiune se poate executa ori de câte ori sunt disponibile operanțele sale de date.
  - Graful poate fi răspândit în mod arbitrar pe o colecție de procesoare.
  - Fiecare nod specifică operatia și adresa fiecărui nod care au nevoie de rezultat.
- Un procesor dintr-o mașină cu flux de date operează o conductă circulară simplă.
  - Un mesaj (jeton, token), constă din date și o adresă (etichetă) a nodului său de destinație.
  - Eticheta este comparată cu cele dintr-un magazin de potrivire.
    - Dacă este prezent, tokenul potrivit se extrage și instrucțiunea este emisă pentru exec
    - Dacă nu, tokenul este plasat în stocare pentru a-și aștepta partenerul.
  - Atunci când este calculat un rezultat, este trimis un mesaj sau un jeton nou, care conține datele rezultatului, la fiecare dintre destinațiile specificate în instrucțiune.

Arhitecturi de flux de date

- Diviziunea primară este dacă graficul este
  - *static,* cu fiecare nod reprezentând o operație primitivă
  - *dinamică,* caz în care un nod poate reprezenta invocarea unei funcții arbitrare, ea însăși reprezentată de un graf.
    - În arhitecturi dinamice sau cu tag-token, extinderea dinamică a graficului pe invocarea funcțiilor se realizează prin transportarea informațiilor de context în etichetă.
- Caracteristica cheie a arhitecturilor de flux de date:
  - capacitatea de a numi operațiuni efectuate oriunde în mașină,
  - suport pentru sincronizarea operațiunilor independente,
  - programare dinamică la nivelul mașinii.
- Masina oferă posibilitatea de a numi un set foarte mare și dinamic de fire care pot fi mapate în mod arbitrar procesoarelor.
  - Dimpotrivă - în arhitecturi paralele de date, compilatorul sau secvențatorul mapează un set mare de operatii in „proc-uri virtuale” pe proc-uri, alocând iterații ale unui ciclu
- În mod obișnuit, aceste mașini ofereau un spațiu global de adrese.
  - Arhitecturile de flux de date au cunoscut o separare treptată a modelului de programare și a structurii hardware pe măsură ce abordarea a ajuns la maturitate.

Arhitecturi sistolice

- Înlocuit procesorul secvențial printr-o multime regulată de PE simplu și obțineți un randament foarte mare cu cerințe de lățime de bandă de memorie modeste.
- Propunerile timpurii au fost determinate de oportunitatea oferită de VLSI de a oferi cipuri ieftine cu scop special.
- Diferență față de unitățile funcționale conducte convenționale:
  - structura poate fi neliniară, de ex. hexagonal,
  - căile dintre PE pot fi multidirecționale,
  - fiecare PE poate avea o cantitate mică de instrucțiuni locale și memorie de date.
- Diferență de SIMD prin faptul că fiecare PE poate face o operație diferită.
- Aspecte în comun cu transmiterea mesajelor, modelele paralele de date și fluxul de date, dar preiau un caracter unic pentru o clasă specializată de probleme:
  - Alg. reprezentată ca o colecție de comp. unități conectate într-un model regulat.
  - Datele se mișcă cu „bătăi de inimă" obișnuite, așa cum este stabilit de statusul local.
- Exemplu: calculul unui produs interior:
  - la fiecare bataie, datele de intrare avansează spre dreapta,
  - se înmulțește cu o greutate locală,
  - este acumulat în secvența de ieșire pe măsură ce avansează și spre dreapta.

Arhitecturi sistolice

- Realizări practice ale acestor idei:
  - iWarp: programabilitate generală în noduri, pentru ca o varietate de algoritmi să poată fi realizati pe același hardware.
- Rețeaua poate fi configurată ca o colecție de canale dedicate, reprezentând modelul de comunicare sistolică,
  - datele pot fi transferate direct de la registrele de procesare la registrele de procesare de pe un canal.
  - cunoașterea globală a modelului de comunicare este exploatată pentru a reduce contenția și chiar pentru a evita impasul.
- Caracteristica cheie a arhitecturilor sistolice:
  - capacitatea de a integra calcule extrem de specializate în condiții de comunicare simple, regulate și foarte localizate.
- Alg. sistolici: în general pot fi soluții pe mașini generice
  - modelul de comunicare locală regulat produce o localitate bună,
  - lățimea de bandă de comunicare necesară este redusă,
  - cerințele de sincronizare sunt simple.

  ⇒ Alg. s-au dovedit eficienti pe întregul spectru de mașini paralele.

Modelul circuit

- Modelează mașina la nivel de circuit, astfel încât să poată fi luate în considerare toate întârzierile de propagare a calculului și a semnalului.
  - este imposibil pentru un supercomputer complex, deoarece
    - generarea și depanarea specificațiilor detaliate ale circuitului nu sunt cu mult mai ușoare decât o implementare completă,
    - un simulator de circuit ar lua eoni pentru a rula simularea.
- Dacă circuitul va fi implementat pe un cip VLSI dens, ar include efectul cablurilor, în termini de
  - zona de cip pe care o consumă (cost)
  - întârzierea propagării semnalului între și în cadrul blocurilor interconectate (timp).

  Nota: în designul modern VLSI întârzierile de sârmă și zona încep să umbrească întârzierile de comutare sau poartă și zona ocupată de dispozitive, respectiv.

  De exemplu,pPentru arhitectura hipercubului, întârzierile de comunicare intre interprocesoare pot domina întârzierile intra-procesoare,
    - ⇒ timpul de comunicare mult mai mare decât cel al arhitecturilor bazate pe grila și tor.

Modelul circuit

- Determinați limitele pentru parametrii zonei și lungimii firului pe baza proprietăților rețelei,
  - fără a fi nevoie să apeleze la specificații și aspect cu instrumente de proiectare VLSI.
- Exemple:
  - în implementarea VLSI 2D, lățimea bisecției unei rețele produce o limită inferioară pe aria de dispunere a acesteia, în sens asimptotic.
    - lățimea bisecției este B => cea mai mică dimensiune a cipului ar trebui să fie cel puțin Bw, unde w este lățimea minimă a firului.
  - În cazul grilelor 2D, zona inferioară va fi liniară în numărul p de procesoare.
    - Se spune că o astfel de arhitectură este scalabilă în sensul de dispunere VLSI.
  - Hipercub: aria necesară este o funcție pătratică a p și arhitectura nu este scalabilă.
- Consumul de energie electrică al circuitelor digitale este un alt factor limitativ:
  - Disiparea puterii în microprocesoarele moderne:
    - crește ~ liniar cu produsul zonei matriței și frecvenței ceasului (ambele cresc)
    - astăzi se ridică la câteva zeci de wați în modele de înaltă performanță.
  - Eliminarea căldurii generate de procente de 1 M este o mare provocare.

Modelul graf

- O arhitectură cu memorie distribuită caracterizată în principal de rețea.
- Rețeaua este de obicei reprezentată ca un grafic:
  - vârfuri corespund procesoarelor - nodurilor de memorie
  - arcele corespund legăturilor de comunicare.
    - Dacă legăturile de comunicare sunt unidirecționale, atunci se utilizează arce direcționate.
    - Arcele nedirecționate implică o comunicare bidirecțională.
- Printre parametrii unei rețele de interconectare se numără:
  1. ***Diametrul rețelei***:
     - cea mai lungă dintre cele mai scurte căi între diferite perechi de noduri,
     - ar trebui să fie relativ mică dacă latența rețelei este redusă la minimum.
  2. ***Lățimea bisectiei (de bandă)***:
     - cel mai mic nr. de legături ce trebuie tăiate pentru a rupe rețeaua în 2 subrețele de dimensiune 1/2.
     - important atunci când nodurile comunică între ele în mod aleatoriu;
     - o valoare mică limitează rata transferului de date între cele două jumătăți ale rețelei.
  3. ***Gradul modurilor sau varfurilor***:
     - Număr de porturi de comunicare necesare fiecărui nod,
     - Constanta independentă de dimensiunea rețelei dacă arhitectura este ușor scalabilă la dimensiuni mai mari.
     - Influențeaza costul fiecărui nod.

Model paralel sincron (BSP)-1990

- încearcă să ascundă latența comunicării cu ajutorul unui stil de programare paralel specific:
  - făcând astfel irelevantă topologia rețelei.
- Sincronizarea procesoarelor se face o dată la fiecare trepte de timp L, unde L este un parametru de periodicitate.
- Un calcul paralel constă dintr-o secvență de superpasi.
  - Într-un suprapas dat, fiecare procesor îndeplinește o sarcină constând din pași de calcul locali, transmisii de mesaje și recepții de mesaje de la alți procesatori.
  - Datele primite în mesaje nu vor fi utilizate în superpasul curent, ci mai degrabă începând cu următorul superpas.
  - După fiecare perioadă de unități de timp L, se face o verificare globală pentru a vedea dacă superpasul curent a fost finalizat.
    - Dacă da, atunci procesoarele trec la execuția următorului superpas.
    - În caz contrar, următoarea perioadă de unități de timp L este alocată superpasului neterminat.

Modelul LogP - 1996

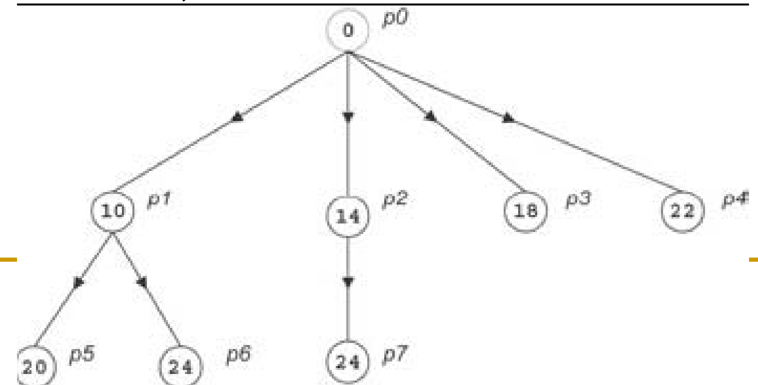
- Dezvoltarea algoritmilor paraleli eficienți suferă de proliferarea rețelelor de interconectare disponibile:
  - designul algoritmului trebuie făcut practic de la zero pentru fiecare nouă arhitectură.
  - ? extrage efectele topologiei de interconectare (ca PRAM pentru masinile cu memorie globala) pentru a elibera proiectantul algoritmului de mulțimea de detalii specifice mașinii.
- Există modele care înlocuiesc informațiile topologice reflectate în graful de interconectare cu un număr mic de parametri;
  - s-a arătat că surprinde efectul topologiei de interconexiune destul de precis.
- Un exemplu de astfel de modele abstracte: modelul LogP (1996).
- Modelul specifică caracteristicile de performanță ale rețelei de interconectare, dar nu descrie structura rețelei.
- Modelul are patru parametri de bază:
  1. *L*: o limită superioară a latenței sau întârzierii, efectuată la trimiterea unui mesaj de la procesorul sursă către procesorul țintă.
  2. *o*: durata în care un procesor este angajat în transmiterea sau recepția fiecărui mesaj - în acest timp proc. nu efectuează alte operațiuni.
  3. *g*: decalajul dintre mesaje: intervalul de timp minim între transmisii consecutive de mesaje sau recepțiile de mesaje consecutive la un procesor.
  4. *P*: numărul de procesoare.

Modelul LogP

- Procesoarele comunică prin mesaje scurte de la punct la punct.
- Presupune un timp unitar pentru operațiunile locale și îl numește ciclu de procesor.
- Parametrii L, o și g sunt măsurați ca multipli ai ciclului procesorului.
- Se presupune că rețeaua are o capacitate finită:
  - cel mult L / g mesaje pot fi în tranzit de la orice procesor sau la orice procesor în orice moment.
  - Dacă un procesor încearcă să transmită un mesaj care să depășească această limită, acesta se oprește până când mesajul poate fi trimis fără a depăși limita de capacitate.
- Modelul este asincron:
  - procesoarele funcționează în mod asincron, iar latența experimentată de orice mesaj este imprevizibilă, dar este delimitată mai sus de L în absența stalelor.
- Din cauza variațiilor de latență, este posibil ca mesajele direcționate către un anumit procesor țintă să nu ajungă în aceeași ordine în care sunt trimise.
- Parametrii nu sunt la fel de importanți în toate situațiile; de multe ori este posibil să fie ignorat unul sau mai mulți parametri și să se lucreze cu un model mai simplu.
  - Algoritmi care comunică rareori datele: ignoră lățimea de bandă și limitele de capacitate.
  - Dacă mesajele sunt trimise în fluxuri lungi canalizate prin rețea (timpul de transmisie este dominat de golurile dintre mesaje), latența poate fi ignorată.
  - În unele MPP, surplusul domină decalajul, deci g poate fi eliminat.

Exemplul 1 / LogP: difuzarea optimă a unei singure unităţi de date de la un procesor la altele P - 1

- Idee: toate procesoarele care au primit unitatea de date o transmit cât mai repede posibil, asigurând în acelaşi timp că niciun procesor nu primeşte mai mult de un mesaj.
  - Sursa începe să transmită unitatea de date la timpul 0.
  - Prima unitate de date intră în rețea la ora o, ia cicluri *L* pentru a ajunge la destinație și este primită de către procesator la timp $L + 2o$.
  - Între timp, sursa va iniția transmisia la altele g, $2g$,…,
  - Presupunând $g \geq o$, fiecare dintre ele acționând ca rădăcina unui arbore difuzat mai mic.
- Arborele de difuzare optim pentru procesoarele p este dezechilibrat cu ieșirea la fiecare nod determinata de valorile relative ale lui *L, o* și *g*.
- Figura: arbore de difuzare optim pentru $P = 8$, $L = 6$, $g = 4$, si $o = 2$.
  - No./node: timpul la care a primit unitatea de date și poate începe să o trimită.
  - Surpulusul procesarii a transmisiilor succesive se suprapune cu livrarea de mesaje anterioare.
  - Procesoarele pot experimenta cicluri de ralanti la sfârșitul algoritmului, în timp ce ultimele mesaje sunt în tranzit.



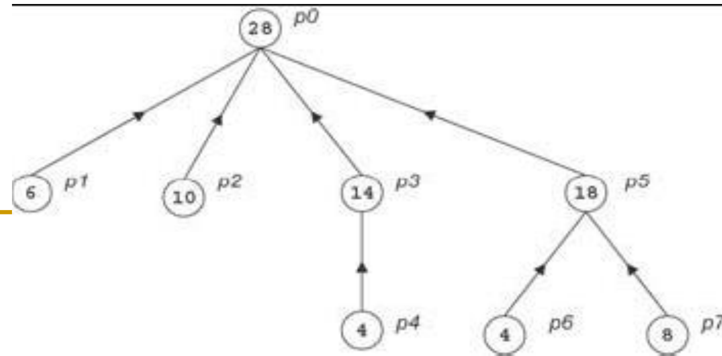
Ex. 2: suma cât mai multor valori posibil în timp T

- Modelul de comunicare între procs formează din nou un arbore(~Ex. 2).
- Fiecare procesor are sarcina de a însuma un set de elemente și apoi (cu excepția procesorului rădăcină) să transmită rezultatul părintelui său.
  - Elemente care trebuie rezumate de un proc. constau din intrări originale stocate în memoria sa, împreună cu rezultatele parțiale primite de la copiii săi în arborele de comunicare.

1. Determina programul optim al evenimentelor de comunicare;
2. Determina distribuția intrărilor inițiale.

- Daca $T \leq L + 2o$,
  - soluția optimă este însumarea a $T + 1$ valori pe un singur proc

  altfel
  - ultimul pas efectuat de procesorul rădăcină (la momentul $T - 1$) este adăugarea unei valori pe care a calculat-o local la o valoare pe care tocmai a primit-o de la un alt procesor.
  - Proc. la distanță trebuie să fi trimis valoarea la momentul T - 1 - L - 2o și presupunem recursiv că formează rădăcina unui arbore de însumare optim cu acesta limitare de timp.
  - … (vezi textbook)
- Fig: arbore de comunicare pentru o însumare optimă pentru $T = 28, P = 8, L = 5, g = 4, o = 2$.



Paradigma de transmitere a mesajelor

Principii

- Paradigma este una dintre cele mai vechi și utilizate pe scară largă pentru programarea computerelor paralele.
  - rădăcinile sale pot fi identificate în primele zile ale prelucrării paralele.
  - adoptarea sa este larg răspândită.
- Există două atribute cheie care caracterizează paradigma.
  1. Presupune un spațiu de adrese partiționat
  2. Nu acceptă decât o paralelizare explicită.
- Fiecare element de date trebuie să aparțină uneia dintre partițiile spațiului:
  - datele trebuie împărțite și plasate explicit;
  - încurajează localitatea in acces.
- Toate interacțiunile (numai citire, citire / scriere) necesită cooperarea a 2 procese:
  1. procesul care are datele,
  2. procesul care dorește să acceseze datele.
- Avantajul interacțiunilor explicite în două sensuri:
  - programatorul cunoaște pe deplin toate costurile interacțiunilor non-locale și este mai probabil să se gândească la algoritmi (și mapări) care reduc la minimum interacțiunile.
  - paradigma poate fi implementată eficient pe o mare varietate de arhitecturi.
- Dezavantaj:
  - Pentru interacțiuni dinamice și / sau nestructurate, complexitatea codului scris pentru acest tip de paradigmă poate fi foarte mare din acest motiv.

Probleme la programare

- Paralelismul este codat explicit de programator:
  - Programatorul este responsabil:
    - pentru analiza algoritmului / aplicației seriale de bază
    - Identificarea modalităților prin care el / ea poate descompune calculele și extrage concurența.
  - Programarea folosind paradigma tinde să fie dură și exigentă intelectual.
  - Pentru programele corespunzătoare de transmitere a mesajelor pot fi deseori obținute performanțe foarte mari la un număr foarte mare de procese.
- Codurile sunt scrise folosind paradigmele asincrone sau slab sincrone.
  - În paradigma asincronă:
    - toate sarcinile concomitente se execută asincron.
    - astfel de programe pot fi mai greu de motivat și pot avea un comportament nedeterminist.
  - Programe sincronizate:
    - sarcinile sau subseturile de sarcini se sincronizează pentru a efectua interacțiuni.
    - între aceste interacțiuni, sarcinile se execută complet asincron.
    - Deoarece interacțiunea se întâmplă sincron, este încă destul de ușor de motivat despre program.
- Paradigma acceptă executarea unui program diferit pe fiecare dintre cele p procesele.
  - oferă flexibilitatea finală în programarea paralelă,
  - face ca meseria de a scrie programe paralele să fie ne-scalabila în mod eficient.
  - ⇒ majoritatea programelor sunt scrise folosind abordarea mai multor date (SPMD) a unui singur program.
    - În programele SPMD, codul executat de diferite procese este identic, cu excepția unui număr mic. procese (de exemplu, procesul „rădăcină”).

Blocuri de construcții: operațiuni de trimitere și primire

- În forma lor simplă, prototipurile acestor operații sunt:
  - send(void *sendbuf, int nelems, int dest)
  - receive(void *recvbuf, int nelems, int source)
    - sendbuf indică un buffer care stochează datele care trebuie trimise,
    - recvbuf indică un buffer care stochează datele care urmează să fie primite,
    - nelems este numărul de unități de date care trebuie trimise și primite,
    - dest este identificatorul procesului care primește datele,
    - source este identificatorul procesului care trimite datele.
- Variante ale modului în care aceste funcții sunt puse în aplicare.
- Exemple:

P0	P1
a = 100;	receive(&a, 1, 0)
send(&a, 1, 1);	printf("%d\n", a);

- Majoritatea platformelor au suport suplimentar hardware pentru trimiterea /receptionarea mesajelor:
    - Transfer de mesaje asincrone folosind hardware-ul interfeței de rețea.
      - permite transferul din memoria tampon în locația dorită fără intervenția procesorului.
    - DMA (acces direct la memorie)
      - permite copierea datelor dintr-o locație de memorie în alta fără suport CPU;
      - Dacă se returnează trimiterea înainte de realizarea operațiunii de comunicare, P1 ar putea primi valoarea 0 în loc de 100!

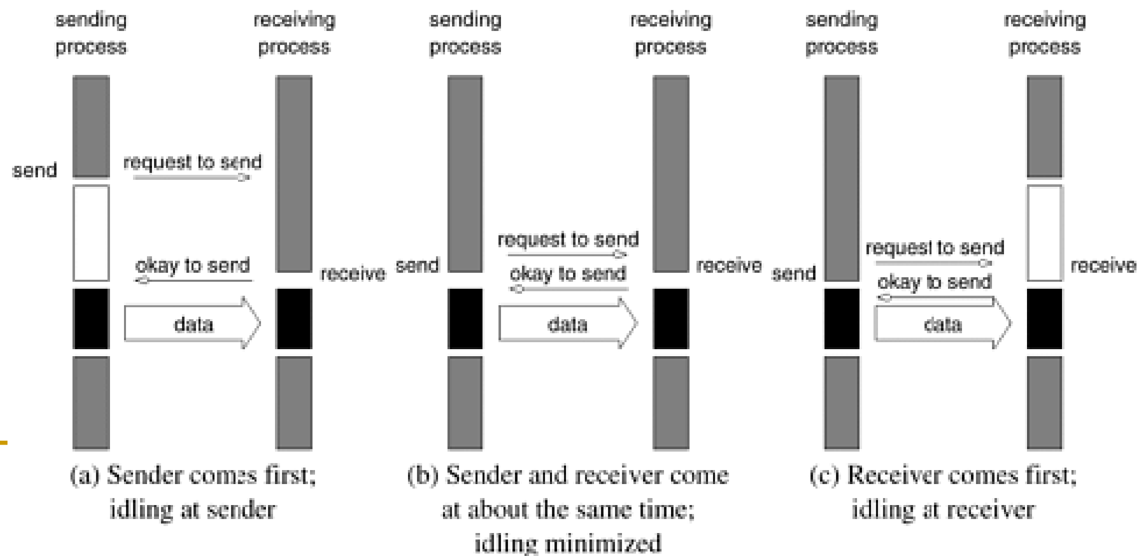
Operatii blocante de transmitere a mesajelor

- Operatia de trimitere blochează până nu poate garanta că semantica nu va fi încălcată la întoarcere indiferent de ceea ce se întâmplă în program ulterior.
- Există două mecanisme prin care se poate realiza acest lucru:
  1. Send/Receive blocant fara tampon.
  2. Send/Receive blocant cu tampon.

1. **Send/Receive blocant fara tampon**
   - Operația de trimitere nu se întoarce până când nu a fost acceptată primirea la procesul de primire.
   - Apoi mesajul este trimis și operațiunea de trimitere revine la finalizarea operațiunii de comunicare.
   - Implică o strângere de mână între procesele de trimitere și primire.
     - Procesul de trimitere trimite o solicitare pentru a comunica procesului de primire.
     - Când procesul de primire întâlnește receive, acesta răspunde la cerere.
     - Procesul de trimitere la primirea acestui răspuns inițiază o operație de transfer.
   - Deoarece nu există tampoane utilizate la capetele de trimitere sau primire, aceasta este, de asemenea, denumită o operație de blocare ne-tamponată.

Surplusuri in Send/Recv blocant fara tampon

- 3 scenarii:
  - (a) send apare inainte de receive,
  - (b) send si receive apar in acelasi timp,
  - (c) receive apare inainte de send.
  - În cazurile (a) și (c): la procesul de trimitere și primire există un surplus considerabil.
  - ⇒ Un protocol non-tampon cu blocare este potrivit atunci când trimiterea și primirea sunt postate aproximativ în același timp.
    - Într-un mediu asincron, acest lucru poate fi imposibil de prevăzut.



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Impas in Send/Recv blocant fara tampon

- Se considera următorul schimb simplu de mesaje care poate duce la un impas:

```
P0                  P1
send(&a, 1, 1); send(&a, 1, 0);
receive(&b, 1, 1);        receive(&b, 1, 0);
```

- Trimiterea la P0 așteaptă primirea potrivită la P1.
- Trimiterea la procesul P1 așteaptă recepția corespunzătoare la P0, rezultând o așteptare infinită.
- Impacturile sunt foarte ușore de realizat.
- În exemplul de mai sus, impasul poate fi corectat prin înlocuirea secvenței de operare a unuia dintre procese de o recepție și o trimitere, spre deosebire de invers;.
- Acest lucru face de multe ori codul mai greoi.

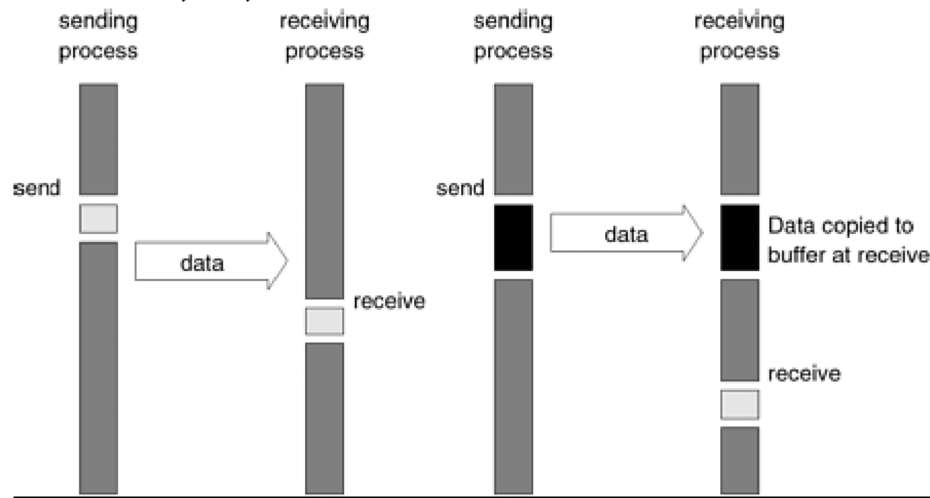
Send/Receive blocant cu tampon

- Expeditorul:
  - are un tampon prealocat pentru comunicarea mesajelor.
  - copiază datele în memoria tampon desemnată.
  - returnează controlul după finalizarea operațiunii de copiere.
  - continuă cu programul știind că orice modificare a datelor nu va afecta semantica programului.
- Comunicarea propriu-zisă se poate realiza în mai multe moduri, în funcție de resursele hardware disponibile.
  - Dacă hardware-ul acceptă comunicare asincronă. (independent de procesor), apoi un transfer de rețea poate fi inițiat după încurcătură. a fost copiat în buffer.
- La receptie:
  - datele sunt copiate și într-un tampon la receptor.
  - când procesul de primire întâlnește o operație de primire, verifică dacă mesajul este disponibil în bufferul său de recepție.
  - dacă da, datele sunt copiate în locația țintă.

Send/Receive blocant cu tampon

(a) în prezența hardware-ului de comunicare cu buffere la capetele de trimitere și primire;
(b) în absența hardware-ului de comunicare:

- expeditorul întrerupe receptorul și depune datele în tampon la capătul receptorului.
- ambele procese participă la o operațiune de comunicare, iar mesajul este depus într-un tampon la capătul receptorului.
- când receptorul întâlnește în cele din urmă o operație de primire, mesajul este copiat din buffer în locația țintă.



Impactul tampoanelor finite în transmiterea mesajelor

- Fie următorul fragment de cod:

```
P0                                  P1
for (i = 0; i < 1000; i++) {        for (i = 0; i < 1000; i++) {
        produce_data(&a);                   receive(&a, 1, 0);
        send(&a, 1, 1); }           consume_data(&a); }
```

- Dacă proc. P1 a ajuns lent la această buclă, proc. P0 ar fi putut trimite toate datele sale.
- Dacă există suficient spațiu tampon, ambele procese pot continua;
- Cu toate acestea, în cazul în care bufferul nu este suficient (adică, revărsarea tamponului), expeditorul ar trebui să fie blocat până când unele dintre operațiunile de primire corespunzătoare au fost postate, eliberând astfel spațiul tampon.
- Acest lucru poate duce adesea la cheltuieli generale neprevăzute și degradare a performanței.
- Este o idee bună să scrii programe care au cerințe de buffer limitate.
- Un fragment de cod simplu, cum ar fi următoarele blocaje, deoarece ambele procese așteaptă să primească date, dar nimeni nu le trimite.

```
P0                      P1
receive(&a, 1, 1);      receive(&a, 1, 0);
send(&b, 1, 1);         send(&b, 1, 0);
```

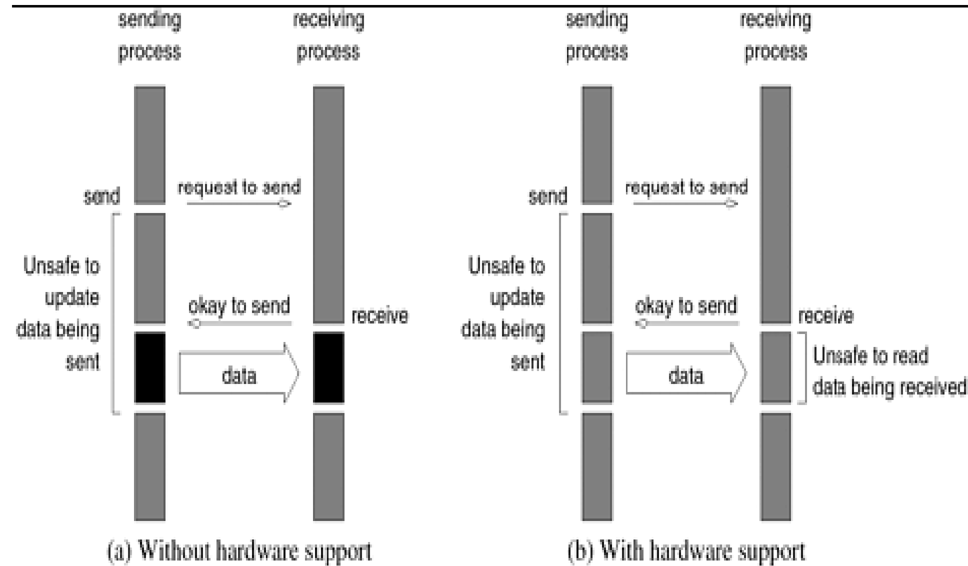
- În acest caz, blocajele sunt cauzate numai de așteptarea operațiunilor de primire.

Operațiuni de transmitere care nu blochează

- Adesea este posibil să se asigure corectitudinea semantică și să ofere o operațiune de trimitere / primire rapidă care duce la un surplus minim
- Se întoarce din opțiunea de trimitere / primire înainte să fie în siguranță semantic.
- Utilizatorul trebuie să fie atent să nu modifice datele care pot participa potențial la o operațiune de comunicare.
- Operațiunile care nu blochează sunt însoțite de o opțiune de check-status indicând dacă semantica transferului inițiat poate fi încălcată sau nu.
- La întoarcerea de la o opțiune de trimitere / înregistrare care nu blochează, se poate efectua orice calcul care nu depinde de finalizarea operației.
- Mai târziu în program, procesul poate verifica dacă operațiunea de neblocare s-a încheiat sau nu și, dacă este necesar, se aștepta finalizarea acesteia.
- Operațiunile care nu blochează pot fi tamponate sau ne-tamponate.
  - În cazul non-tamponat:
    - un proces care dorește să trimită date altuia pur și simplu postează un mesaj în așteptare și revine la programul de utilizator. Programul poate face apoi alte lucrări utile.
    - Când apare receive, operațiunea de comunicare este inițiată.
    - Când această operație este finalizată, operația de verificare a stării indică faptul că este sigur ca programatorul să atingă aceste date.

Spațiul posibilelor protocoale

	Blocking Operations	Non–Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non–Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

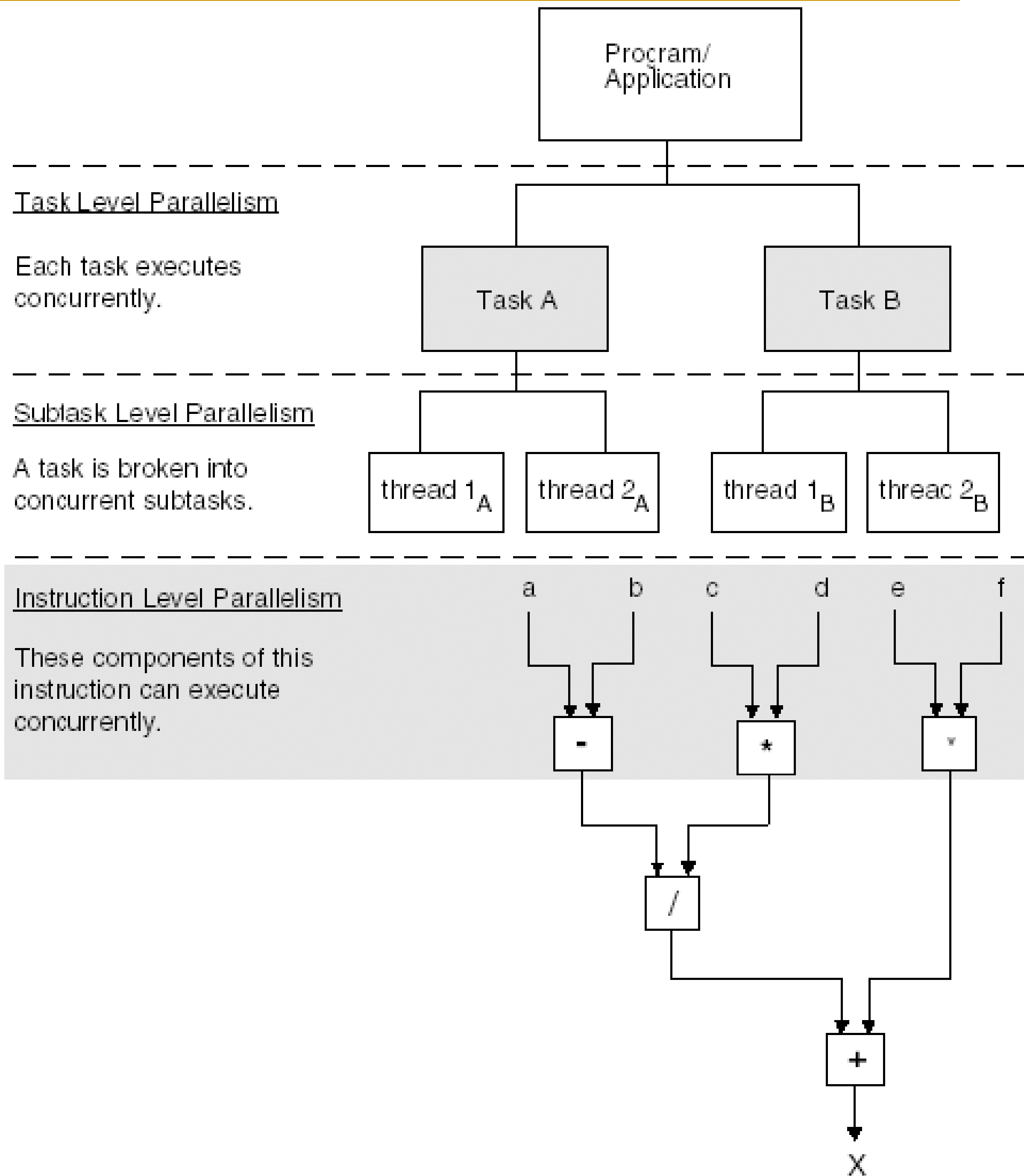


(a) Without hardware support
(b) With hardware support

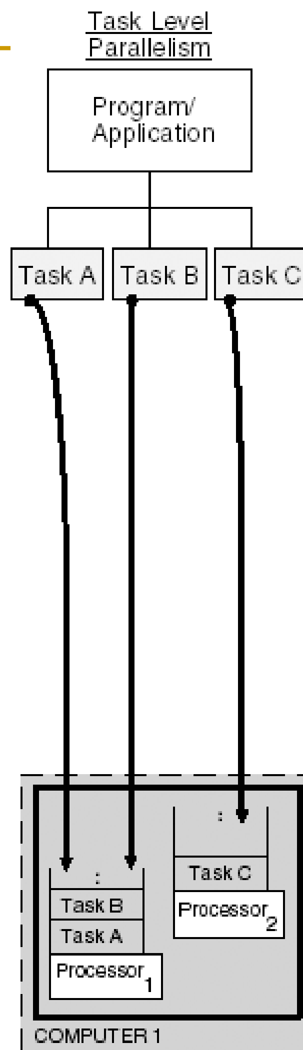
Non-blocking non-buffered send & recv ops
(a) în absența hardware-ului de comunicare;
(b) în prezența hardware-ului de comunicare.

Niveluri de paralelism

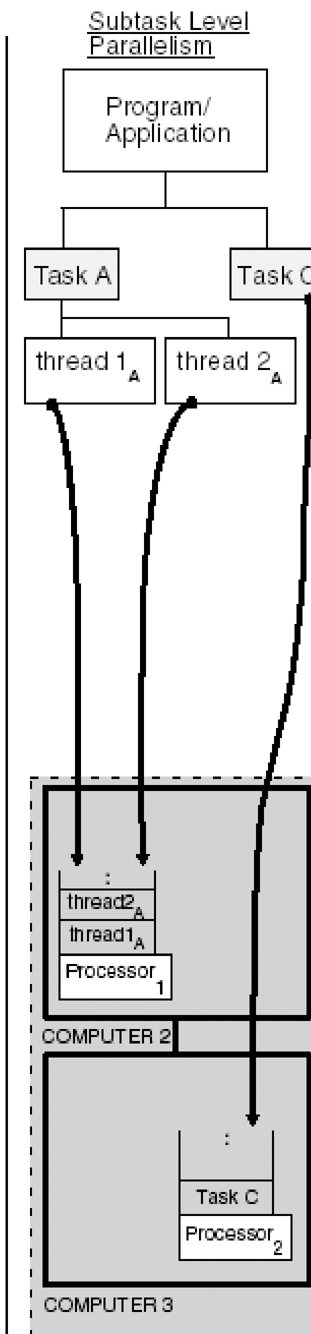
Niveluri de paralelism care sunt posibile în cadrul unui singur program de calculator



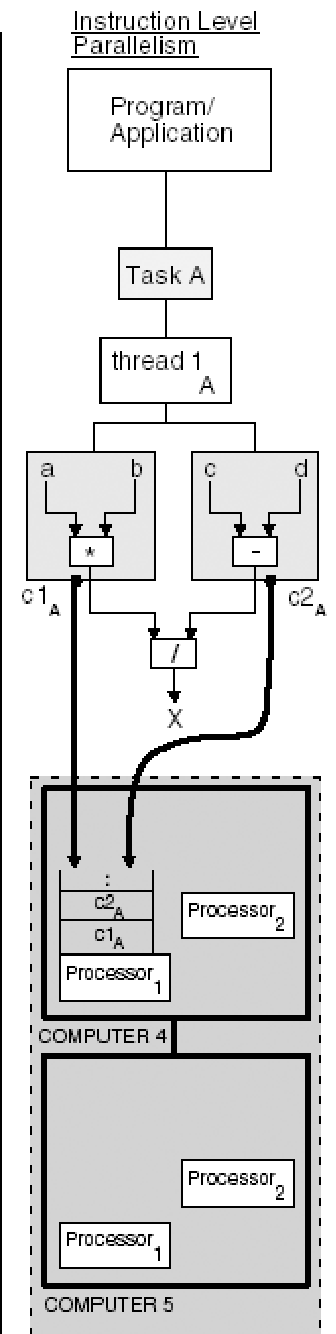
Niveluri de paralelism combinate cu configurațiile de bază ale procesorului paralel



Single multiprocessor computer



PVM with multiple single processor computers



PVM with multiple multiprocessor computers

Niveluri de paralelism

1. *Micro-paralelizare*
   - are loc în interiorul unui singur procesor.
   - nu necesită intervenția programatorului pentru implementare.
2. *Paralelizare de nivel granular mediu*
   - asociate cu limbajul acceptat sau paralelizarea ciclurilor.
   - Cu toate că s-au făcut unele progrese în automatizarea acestui nivel de paralelizare cu ajutorul compilatoarelor optimizate, rezultatele acestor încercări sunt doar satisfăcătoare moderat.
3. *Paralelizare de nivel granular grosier*
   - asociate cu computere paralele cu memorie distribuită
   - este introdus aproape exclusiv de programator.
4. *Paralelizare la nivel de grid*
   - model foarte promițător pentru rezolvarea problemelor mari,
   - aplicabilitatea sa este limitată la anumite clase de probleme de calcul, aparținând categoriei „la scară largă în mod jenant paralel”.

Micro-paralelism

- Procesoarele desktop moderne, cum ar fi cele dezvoltate de Intel, AMD, IBM etc. sunt deja paralizate.
  - Astfel de procesoare au mai multe conducte pentru operații întregi și în virgulă plutitoare, astfel încât pot fi luate în considerare două niveluri diferite de paralelizare:
  1. În primul rând, adâncimea conductei:
     - dacă se utilizează o conductă de adâncime k, atunci operațiile k pot fi executate în același timp.
  2. În al doilea rând, numărul de conducte:
     - presupunând că există l conducte întregi de adâncime k1 și m conducte în virgulă flotantă de adâncime k2 într-un procesor dat și că toate conductele funcționează la capacitate maximă la un moment dat, atunci operațiunile $k1 \times l + k2 \times m$ sunt executate de procesorul concomitent în fiecare ciclu.
- Disponibilitatea acestui nivel de paralelism este o funcție a dependențelor din cadrul unui flux de operații de limbaj mașină.
  - Aceste dependențe sunt analizate și microparalelismul este acceptat:
  1. de către unitatea logică din interiorul procesorului la nivel hardware
  2. de către compilator și optimizare suportata de compilator la nivel de software.

Paralelism la nivel granular mediu

- Să presupunem că mai multe procesoare sunt conectate la o memorie partajată globală (logic și fizic), modul tipic de introducere a paralelizării este de a efectua operațiuni similare pe subseturi de date.
- Nivel algoritmic natural de realizare a unei astfel de paralelizări: divizare între mai multe procesoare a unui ciclu.
  - Un ciclu dat este împărțit în părți câte procesoare există;
  - Presupunem că numărul de părți este egal cu numărul de procesoare;
  - Fiecare parte este executată independent de un procesor separat.
- Susținut fie printr-un set de directive speciale, fie prin extensii de limbaj la nivel înalt.
- Compilatoarele capabile să genereze automat acest nivel de paralelism.
  - Rezultatele au fost până acum dezamăgitoare.
  - Compilatoarele paralele sunt relativ reușite în generarea codului par. cu cicluri simple, adunarea a doi vectori, înmulțire matriceala etc..
  - În cazuri mai complicate, de exemplu, când funcțiile sunt apelate în interiorul ciclurilor, codul trebuie să fie împărțit manual în unități paralele.

Paralelism de nivel granular grosier

- Abordarea tipică a paralelizării memoriei distribuite este crearea de unități de programare independente care vor executa unități de lucru separate care comunică între ele prin trecerea mesajelor.
- Minimizarea numărului de mesaje transmise între componente devine un obiectiv important în proiectarea programului.
- Trebuie să căutăm să împărțim un program paralel distribuit în mari unități de calcul care să fie cât mai independente una de cealaltă și să comunice rar.
  - Cel mai adesea, fiecare unitate de lucru este un derivat al programului principal și realizează același subset de operații ca celelalte unități de lucru, dar pe seturi de date separate.
    - Acest tip de abordare se numește SPMD (program unic, date multiple) și este adesea denumită paralelizare la nivel grosier.
- Acest nivel de paralelizare trebuie implementat manual de programator, deoarece diviziunea sarcinilor se bazează pe o analiză semantică a alg. utilizate pentru rezolvarea problemei.
- Una dintre problemele importante: echilibrarea incarcarii.

Paralelism de nivel granular grosier

- Deoarece fiecare unitate de calcul este relativ independentă, poate fi nevoie de timpi diferiti pentru finalizare.
  - La rândul său, acest lucru poate duce la o situație în care toți mai putin unul dintre precesoare rămân inactive în timp ce așteaptă ca ultimul să își finalizeze activitatea.
  - Unitățile de lucru ar trebui să fie mari și independente și, de asemenea, ar trebui să își îndeplinească sarcinile într-un timp similar => abordarea SPMD ceva mai atractivă decât divizarea muncii în unități complet independente.
- Software-ul „să se potrivească” cu hardware-ul de bază.
  - tratarea unei mașini de memorie partajată ca un computer cu memorie distribuită și aplicarea abordării pe baza transmiterii mesajelor.
  - tratarea unui computer cu memorie distribuită ca un sistem de memorie comună (abordare impractica)
  - hardware hibrid, unde nodurile de memorie partajată sunt combinate într-o configurație de memorie distribuită, ceea ce duce la un computer distribuit cu memorie partajată.
    - Tratarea unei astfel de mașini ca un computer cu memorie distribuită și aplicarea tehnicilor de paralelizare adecvate este de obicei mai reușită decât tratarea acesteia ca un mediu de memorie partajată.

Paralelism la nivel de Grid

- O serie de computere, indiferent de arhitecturile lor individuale sunt conectate slab intr-o rețea.
- În cel mai general caz, fiecare mașină și conexiunile dintre ele se presupune a fi diferite.
  - Sistem extrem de eterogenheterogeneous system,
  - Necesită cel mai grosier nivel de paralelizare:
    - activitatea trebuie împărțită în unități independente care pot fi finalizate pe diferite calculatoare la viteză diferită și returnate coordonatorului soluției principale în orice moment și în orice ordine, fără a compromite integritatea soluției.
- Exemple de sarcini testate cu succes:
  - analiza seturilor foarte mari de blocuri de date independente, în care problema constă în dimensiunea totală a datelor care trebuie analizate
    - precum proiectul SETI@home