EDICIONS VIRTUALS

Josep Díaz
María Serna

Algorítmica per a paral·lelisme massiu.

Introducció

EDICIONS UPC

Josep Díaz i María Serna

Algorítmica per a paral·lelisme massiu. Introducció

Índex

Capítol 1

Models de computació paral·lela

Aquest llibre és una introducció al disseny dels algorismes paral·lels. El seu objectiu fonamental és ensenyar a pensar en paral·lel, és a dir, a descompondre els problemes de manera que se'n pugui aprofitar l'estructura paral·lela, quan aquesta existeix. Aquesta descomposició ens permetrà utilitzar el paral·lelisme massiu per resoldre, de manera *eficient*, alguns problemes en molt menys temps que amb solucions seqüencials.

Què vol dir *de manera eficient*? En principi, considerarem problemes que tenen una solució seqüencial polinòmica determinista, és a dir, problemes a la classe P. Volem dissenyar algorismes que siguin més ràpids que els seqüencials i que utilitzin un nombre *no gaire elevat* de processadors (un nombre constant o polinòmic a l'entrada del problema que, a més, depèn de la grandària del problema). Com és usual, els ordres de magnitud dels recursos són definits respecte a la talla de l'entrada del problema.

Evidentment, parlar d'un nombre polinòmic de processadors no és realista, ja que això equivaldria a tenir màquines amb un nombre molt elevat i variable de processadors, però el nostre propòsit és l'estudi de la naturalesa intrínseca dels problemes per veure el motiu que fa que aquests problemes es puguin paral·lelitzar de manera eficient. En aquest sentit, no ens preocuparem excessivament dels aspectes d'implementació.

La lectura del llibre pressuposa un coneixement del material bàsic d'algorísmica i complexitat seqüencial equivalent, per exemple, al contingut dels 15 primers capítols del llibre de Cormen, Leiserson i Rivest [CLR89].

1.1 Memòria i processadors

Un processador seqüencial es compon d'una unitat de processament amb alguns registres i una memòria a la qual pot accedir. Quan tenim diversos processadors que han de treballar en paral·lel, hem de decidir la manera com els processadors han d'accedir a la memòria. Primer cal decidir la disposició de la memòria com es pot tenir un bloc de memòria únic, que anomenarem memòria *compartida* o memòria *comuna*, o bé mantenir una partició en mòduls independents, que s'anomena memòria *distribuïda*. Després hem de decidir com accedeixin els processadors a la memòria. A la figura 1.1 se'n presenten els models

principals:

(a) La memòria és compartida i cada processador pot accedir a la memòria directament. En aquest model, la comunicació entre els processadors es fa a través de la memòria. Aquest és l'esquema corresponent a la RAM paral·lela o PRAM, que descriurem més endavant.

(b) La memòria es distribueix en diferents mòduls amb una xarxa d'interconnexió que modula l'acces dels processadors a les dades. La xarxa d'interconnexió habitualment té més processadors. Alguns d'aquests processadors addicionals poden ser processadors especialitzats en la transmissió de dades.

(c) És similar al cas (b), però cada processador té assignada una part de la memòria comuna amb una xarxa d'interconnexió per comunicar-se entre els processadors.

(d) La memòria es reparteix entre tots els processadors d'una xarxa d'interconnexió. La diferència amb el cas previ és que ara no hi ha cap processador addicional a la xarxa d'interconnexió.

En els casos (b) i (c), els processadors addicionals tenen una capacitat de càlcul suficient per transmetre missatges i mantenir algun tipus de cua.

Els sistemes de tipus (a) s'anomenen màquines amb *memòria compartida*, els dels tipus (b) i (c), màquines amb *memòria distribuïda*, i els del tipus (d) *xarxes* de processadors.

1.2 Models de computació

Un ordinador opera executant instruccions sobre dades. Un conjunt d'instruccions (algorisme) indica a l'ordinador el que ha de fer a cada pas. Aquestes instruccions actuen sobre el conjunt de dades d'entrada. Segons el tipus d'arquitectura, podem considerar quatre tipus de models de computació:

• Màquines SISD (Single Instruction Single Data). Aquest és el model seqüencial de Von Newman. En aquest model només hi ha un flux de dades i una seqüència d'instruccions.

• Màquines MISD (Multiple Instruction Single Data). Tenim n processadors, n instruccions i la mateixa entrada de dades. Les màquines MISD s'utilitzen per a aplicacions molt especialitzades com la classificació d'elements diferents d'un mateix compost químic (vegeu, per exemple, l'exercici **1:2**).

• Màquines SIMD (Single Instruction Multiple Data). Aquest tipus de màquines consten de n processadors, cadascun amb la seva memòria local. Tots els n processadors operen sota el control d'una **única** instrucció, donada per una unitat central de control (un rellotge). Cada processador pot accedir a dades diferents. La màquina SIMD és un model síncron. A cada pas, tots els processadors executen la mateixa instrucció sobre dades diferents i el cicle d'instrucció no finalitza fins que no ho fa el procesador, que triga més temps. Aquesta instrucció pot ser simple (sumar o comparar dues quantitats) o complexa (fer una fusió o una classificació). A vegades pot ser necessari que un subconjunt de processadors

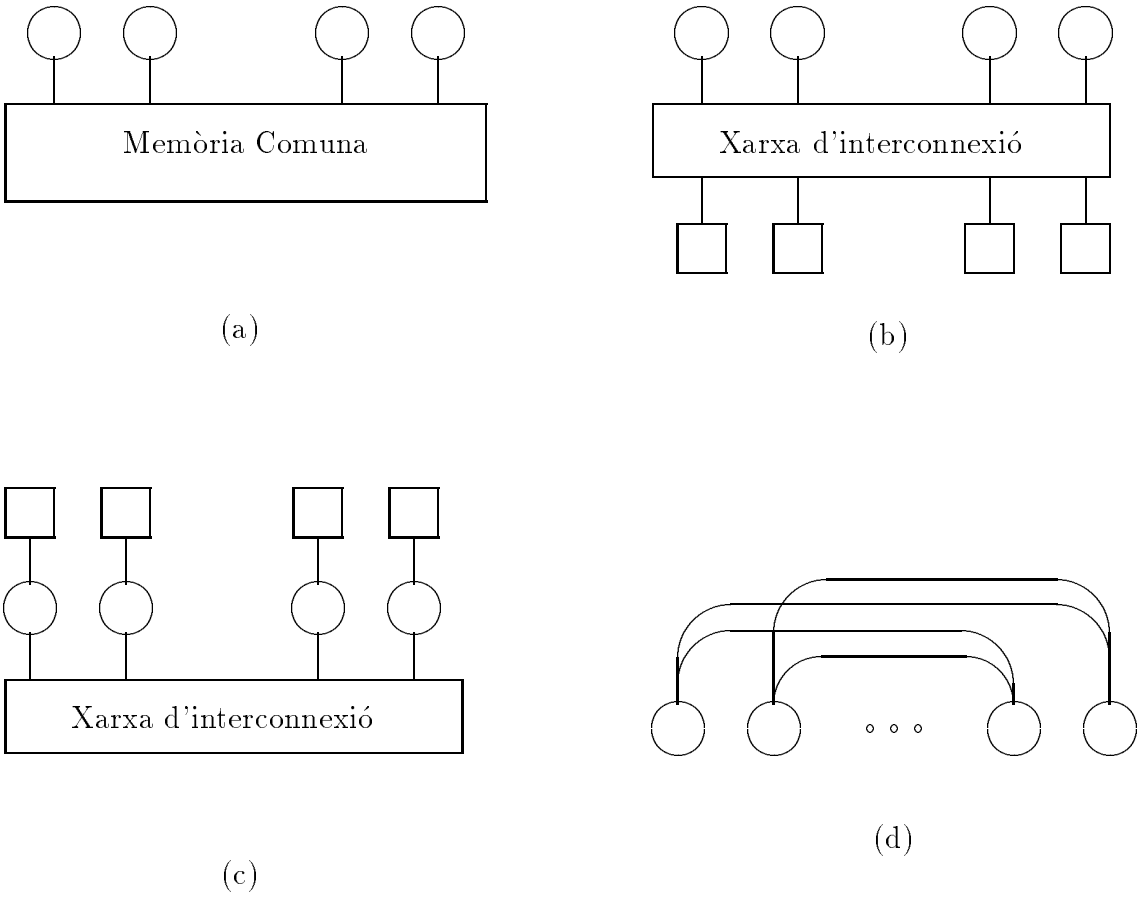


Figura 1.1: Memòria compartida i distribuïda.

no executin una instrucció. Aquesta informació es pot codificar a la instrucció mateixa, indicant a cada processador quan ha de estar actiu o passiu.

En aquest model de màquina, també es requereix que els processadors puguin comunicar-se entre ells. Hi ha dues maneres de realitzar aquesta comunicació: per via de xarxa d'interconnexió i per via de memòria compartida.

A causa de la simplicitat i sincronia del model, les màquines SIMD són particularment efectives a l'hora d'estudiar les propietats que fan que un problema de la classe P sigui paral·lelitzable, i aquest és el model que utilitzarem al llarg del llibre.

• Màquines MIMD (Multiple Instruction Multiple Data). Aquest tipus de màquina és el més general i potent. Consta de n processadors, n instruccions i n fluxos de dades.

En una màquina MIMD, cada processador, a més de la memòria local, també té la seva pròpia unitat de control. Són processadors més potents que els utilitzats a les SIMD. Cada processador opera sota el control d'una instrucció donada per la seva unitat de control. Per tant, els processadors executen programes diferents sobre dades diferentes mentre resolen subproblemes del mateix problema. Consegüentment, els processadors actuen asíncronament. La comunicació entre els processadors es pot realitzar com a la SIMD; per via de xarxa d'interconnexió o per via de memòria compartida.

Les MIMD amb memòria compartida s'anomenen *multiprocessadors* (tightly coupled machines), mentre que les MIMD que es comuniquen utilitzant una xarxa d'interconnexió s'anomenen *multicomputadors* o *sistemes distribuïts* (loosely coupled machines). El model MIMD presenta dificultats de programar, ja que s'hi han d'introduir semàfors o sistemes de sincronització equivalents. En general s'utilitzen per resoldre problemes d'estructura complexa, en que on les SIMD, per rao de la seva senzillesa, no poden ajudar. Mentre que, com ja hem dit, les SIMD s'utilitzen en problemes P, les MIMD s'utilitzen per paral·lelitzar problemes de més complexitat, inclosos els problemes PSPACE-complets. En l'estudi teòric de les propietats que fan que un problema sigui paral·lelitzable o no, utilitzarem la SIMD. Com a punt de referència, veurem un exemple d'aplicació de la MIMD, per entendre la seva potència i dificultat d'aplicació.

Exemple 1.1 (Cerca heurística). *Una aplicació de les* MIMD *és la paral·lelització de programes de cerca heurística per a jocs d'estratègia, com els escacs.*

El que fan aquests programes és generar i explorar l'anomenat *arbre de joc*. L'arrel de l'arbre és la configuració actual. Els fills de l'arrel representen totes les posicions que es poden obtenir en una jugada a partir de l'arrel. Els fills dels fills representen les configuracions obtingudes per la resposta del contrari, etc. A cada fulla de l'arrel s'assigna un valor que representa "quant de bona" és, des del punt de vista d'obtenir la victòria. Aleshores, el que fa l'algorisme és explorar tot l'arbre i cercar quin és el millor camí (el camí que portarà cap a la victòria). En general aquests arbres són molt grans, perquè exploren un nombre exponencial de possibilitats (Aquest tipus de jocs són PSPACE complets). Amb vista a rebaixar el temps de cerca, els algorismes generen l'arbre al mateix temps que fan l'exploració, utilitzant una cerca amb profunditat prioritària. Començant per l'arrel, es creen i exploren els camins, un per un. En alguns nusos pot succeir que, sobre la base d'informacions ja obtingudes, podem conèixer els descendents d'aquest nus que ens portaran

a camins pitjors (estratègicament parlant) que els camins ja explorats. En aquest moment ja no cal continuar per aquesta branca i tornem a generar el proper camí.

Aquest procediment s'executa seqüencialment. Una manera d'implementar-ho amb una MIMD és distribuir els subarbres entre els processadors i procurar que el màxim nombre de subarbres siguin explorats en paral·lel. En el transcurs de la computació, els diferents processadors poden intercanviar informació. Per exemple, un processador pot obtenir d'un altre processador la millor jugada fins a aquest moment, i utilitzar-la per esporgar l'arbre que li resta per explorar. Altra informació que es pot intercanviar és el moment en què els diferents processadors finalitzen la tasca.

Observem que aquest tipus d'algorismes no són eficientment implementats amb una SIMD, ja que en qualsevol instant la instrucció executada per un processador pot ser diferent de les instruccions executades pels altres processadors; un processador pot estar generant una nova posició, mentre que un altre processador està avaluant una fulla o comunicant la millor funció.

1.3 Complexitat dels algorismes paral·lels

Com és habitual, per parlar de la complexitat utilitzarem les notacions asimptòtiques següents. Donades dues funcions, f i g, direm que

> $g(n)$ és *d'ordre almenys* $f(n)$, i ho denotarem per $g(n) = \Omega(f(n))$, si existeix una constant $c \geq 1$ i un natural n_o tals que per a tot $n \geq n_o$ tenim $g(n) \geq c \cdot f(n)$.
>
> $g(n)$ és *d'ordre com a màxim* $f(n)$, notat $g(n) = O(f(n))$, si existeix una constant $c \geq 1$ i un natural n_o tals que per a tot $n \geq n_o$ tenim $g(n) \leq c \cdot f(n)$.
>
> $g(n)$ és *asimptòticament equivalent* a $f(n)$, notat $g(n) = \Theta(f(n))$, si $g(n) = O(f(n))$ i $g(n) = \Omega(f(n))$.

Quan s'estudia la complexitat temporal d'un algorisme, es mesura el temps que tarda un algorisme a finalitzar. Per avaluar la qualitat d'un algorisme seqüencial hi han dos paràmetres a considerar:

1. Si l'algorisme és el més rapid possible.

2. En cas negatiu, com es pot comparar amb altres algorismes existents.

Per contestar (1), s'ha d'obtenir una fita inferior al problema i comparar amb la complexitat de l'algorisme obtingut. Recordem, per exemple, la multiplicació de matrius.

Exemple 1.2 (Multiplicació de matrius). *Siguin A i B dues matrius amb dimensions $m \times n$ i $n \times k$, respectivament. Volem calcular el seu producte.*

Recordem que la matriu producte $C = A \times B$ ve donada per

$$c_{ij} = \sum_{s=1}^{n} a_{is} b_{sj}.$$

Una fita inferior per multiplicar dues matrius $n \times n$ és $\Omega(n^2)$. D'altra banda, l'algorisme seqüencial canònic, que es dóna com a algorisme 1, té una complexitat de $O(n^3)$ i, per tant, no és òptim. De fet, es coneixen algorismes més ràpids que l'algorisme 1, com l'algorisme de Strassen. Però no es coneix cap algorisme òptim.

Algorisme 1 Multiplicació de matrius.

```
      MULTIPLICAR(a[n,k], b[k,m])
1     per i = 1 fins n fer
2         per j = 1 fins k fer
3             c[i,j] := 0
4             per s = 1 fins k fer
5                 c[i,j] := c[i,j] + a[i,s] * b[s,j]
              fper
          fper
      fper
```

Considerem la complexitat seqüencial d'un altre problema, el problema de la cerca.

Exemple 1.3 (Cerca no ordenada). *Donat un subconjunt $S = \{s_1, \cdots, s_n\}$ d'un conjunt A i un element $x \in A$, hem de determinar si $x \in S$.*

Si el conjunt S no està ordenat, el problema de la cerca té una fita inferior $\Omega(n)$. Una implementació de l'algorisme de cerca es dóna com a algorisme 2, on el conjunt S es representa mitjançant un vector no ordenat que conté els elements de S. L'algorisme 2 té

Algorisme 2 Cerca seqüencial d'un conjunt.

```
      CERCA(s[n], x, k)
1     i := 1; k := 0
2     mentre i ≤ n i k = 0 fer
3         si s[i] = x
4             llavors k := 1
5             altrament i := i + 1
          fsi
      fmentre
```

complexitat $\Theta(n)$, així, doncs, és una solució seqüencial òptima al problema de la cerca.

Quan s'analitzen algorismes paral·lels, especialment amb el model SIMD, s'apliquen les mateixes idees que amb el seqüencial. Però, en aquest cas, a més del temps hi ha un altre paràmetre que ens interessa; el nombre de processadors. Per poder comparar dos algorismes quan un (o els dos) són algorismes paral·lels, hem de tenir en compte quin dels dos paràmetres volem controlar.

Donat un algorisme paral·lel A, representarem per $T(A)$ el temps en el cas pitjor i per $P(A)$ el màxim del nombre de processadors utilitzats a cada pas. Suposem ara que tenim dos algorismes, A i B, per resoldre el mateix problema.

En primer lloc, podem comparar els dos algorismes mirant només el temps en el cas pitjor. L'**acceleració** d'A respecte de B es defineix com

$$Acc(A,B) = \frac{T(A)}{T(B)}.$$

Aquesta mesura té l'inconvenient de no considerar el nombre de processadors, que habitualment és una quantitat important. Definim el **cost** o **treball** d'un algorisme paral·lel com el producte del temps que triga pel nombre de processadors que utilitza,

$$W(A) = T(A)P(A).$$

Observem que, per a un algorisme seqüencial el cost i el temps coincideixen. A partir de la definició prèvia de cost, podem comparar els algorismes paral·lels trobats amb els algorismes seqüencials que existeixen i amb altres algorismes paral·lels. Per això s'introdueix un nou paràmetre. Definirem **l'eficiència** d'A respecte de B com

$$Ef(A,B) = \frac{C(A)}{C(B)}.$$

Diem que un algorisme paral·lel és **òptim** si el seu cost és igual a la fita inferior del problema. En el disseny d'algorismes paral·lels, volem obtenir algorismes amb eficiència 1, o tan a prop com sigui possible de l'algorisme òptim.

1.4 Referències bibliogràfiques

Textos bàsics sobre paral·lelisme són [Qui87] and [JGD87], entre altres. Una bona referència sobre les màquines MIMD i jocs és [MC82].

Exercicis

1:1 Si voleu paral·lelitzar un problema que sabeu que pertany a la classe PSPACE-complet, quin model de màquina paral·lela escollireu?

1:2 Considerem que volem analitzar una mostra de l'aigua del port de Barcelona per obtenir les quantitats que conté de cadascuna de les 15 substàncies cancerígenes de la llista de la NIH. Per fer-ho de manera senzilla i ràpida, quin tipus de màquina paral·lela utilitzaríeu?

1:3 Problema del faraó i la torre de Lego: Un faraó, amb vista a perpetuar el seu ego, vol construir al mig del desert una torre amb una base quadrada de 100 m^2 i amb una alçària d'1 Km. Per construir-la, una fàbrica li ofereix uns blocs de mida 100 m$\times$100 m$\times$1 m i uns elevadors que poden carregar una pila de fins a 5000 blocs i col·locar-la a sobre d'una altra pila de fins a 5000 blocs. Per col·locar un bloc a sobre d'un altre bloc cal una setmana, i per col·locar una pila a sobre d'una altra pila cal una setmana si la pila té menys de 100 blocs i dues si en té més. El faraó disposa de tants diners com calgui, però vol construir la torre en el mínim temps possible. Dissenyeu un pla de treball per construir la torre amb el mínim temps.

1:4 Una indústria química guarda els productes tòxics en vuit magatzems. A cada magatzem hi ha vuit barrils i cada un és ple d'un sols tipus de substància tòxica. Cap producte és tòxic per si mateix, però esdevé tòxic en combinació amb els altres. Es vol fer una redistribució dels barrils emmagatzemats de manera que cada magatzem tingui només productes d'un tipus. El trasllat s'ha de fer per carretera, cada camió pot fer un viatge d'anada i tornada entre dos magatzems en un dia, però carregant només quatre barrils. Per raons de seguretat es vol fer el trasllat de la manera més discreta possible. Determineu el mètode més ràpid de fer-ho i quants dies es trigaria si:

(a) Volem que un sol camió faci el trasllat.

(b) Es poden contractar més camions, però cada dia només un camió pot visitar un magatzem donat.

1:5 Una empresa de missatgeria utilitza un grup de setze avions per establir connexió entre dos grups de vuit ciutats. Fins ara fa servir un sistema radial i cada ciutat té assignat un avió que diàriament fa un viatge a Central, la ciutat més equidistant dels dos grups. A Central es reparteixen els paquets d'acord amb la seva destinació, de manera que al viatge de tornada cada avió porta la correspondència dirigida cap a la seva ciutat d'origen. Malauradament, per raons de seguretat, cal que cada dia coincideixin no més de dos avions a la mateixa ciutat i, per tant, el sistema s'ha de canviar.

Es vol mantenir encara la velocitat del correu: si ara cada avió ha d'aterrar dues vegades, l'empresa pot permetre alguna escala més. Per això l'empresa ha negociat

amb diferents aeroports a distàncies intermèdies i n'hi ha més de 30 que podrien ser acceptables.

La idea és distribuir primer el correu d'un grup cap a l'altre, començant amb dos avions a cada ciutat, i després fer el mateix recorregut a la inversa. Es vol dissenyar una estratègia tenint en compte que

1. Hi ha dos avions que surten de cada ciutat.
2. Els avions poden utilitzar destinacions intermèdies per intercanviar paquets, encara que mai no aterrin més de dos avions en un aeroport.
3. Cada paquet ha de ser en un avió que finalment aterri a la seva ciutat de destinació.
4. Cada avió pot fer com a màxim, dues escales.

1:6 Quina és la raó que als enunciats anteriors, els números siguin vuit i setze?

Capítol 2

Algorismes sistòlics bàsics per la Mesh

A fi d'extreure les característiques que fan que un problema sigui fàcilment paral·lelitzable, utilitzarem un model simple i síncron, la SIMD. En aquest capítol i els següents estudiarem les màquines SIMD que es comuniquen per xarxa d'interconnexió. Al capítol 6 presentarem el model SIMD amb memòria compartida.

Els processadors de la SIMD, cadascú amb la seva memòria local, estan interconnectats directament entre ells. Cada processador a cada pas pot enviar o rebre un únic missatge per cada aresta de sortida. En aquestes condicions, seria desitjable connectar cada processador amb tots els altres, però això pot ser massa costós. Amb n processadors, el nombre total de connexions és de $n(n-1)/2 = O(n^2)$; això fa que la xarxa presenti problemes espacials de comunicació i, a més, és massa costós mantenir totes les connexions. Per tant, s'utilitzen topologies d'interconnexió que no siguin tan costoses i que mantinguin baix el nombre de passos necessaris per comunicar qualsevol parell de processadors a la xarxa.

Les diferents maneres d'interconnectar els processadors, com també el disseny d'algorismes per passar missatges entre processadors, ha esdevingut un camp de recerca molt interessant. En aquest llibre, únicament considerarem un subconjunt petit de topologies que, d'altra banda, són suficients per obtenir bons resultats per a una gran part de les aplicacions. Les xarxes que considerarem poden ser modelades com a grafs on els vèrtexs representen processadors i les arestes connexions directes entre processadors. La manera en la qual els processadors obtenen simultàniament les dades, es transmeten-se missatges entre ells, però com que no tots els processadors tenen connexió directa, els missatges hauran de travessar alguns processadors abans d'arribar a la seva destinació. El *hardware* responsable de dirigir els missatges s'anomena el *router*, que implementa els algorismes de transmissió, per què cada missatge arribe a la seva destinació de la manera més eficient.

Hi ha diferents paràmetres en els problemes de transmissió de dades:

1. **Topologia de la xarxa** Com estan connectats els processadors. Poden estar com vector lineal, com una mesh, com un hipercub, etc ...

2. **Control de fluxe** Com es transmeten els missatges entre processadors. Els missatges

es poden trencar en paquets i cada paquet s'envia tot sencer individualment. Aquest tipus s'anomena *store and forward*. Una alternativa es el *worm routing*. Els missatges es divideixen es seqüències curtes d'un o dos bits d'informació. Totes les seqüències que provenen del mateix missatge segueixen el mateix camíi, una derrera l'altra, de manera que seqüències adjacents s'emmagatzemen al mateix o en processadors consecutius.

3. **Gestió d'espera** El fet que per cada aresta únicament pugui circular un paquet en la mateixa direcció a cada instant fa que s'hagi d'establir un mecanisme d'espera i prioritats a cada nus. Bé pot ser una cua de prioritat tipo FIFO, o que els paquets que van més lluny tinguin prioritat, o la prioritat sigui per als que van a processadors de més a prop. Un tipus particularment interessant de mecanisme és el *Hot-potato* o deflecció, on un paquet no s'atura mai en un processador. Si quan arriba, la sortida més adient està ocupada, surt per qualsevol altra sortida, pero no hi ha cap tipus de *buffer* als processadors.

4. **Tipus d'encaminament** Es pot demanar que els missatges vagin pel camí mínim (evidentment en aquest cas no podem utilitzar deflecció), o que el camí depend únicament del punt de partida i del punt d'arribada. En aquest cas, es diu que la xarxa és *oblivious*. Altrament, o la xarxa pot ser *adaptativa*, és a dir, a cada moment de la transmissió, l'ecaminament del paquet depèn de l'estat de la xarxa.

5. **Tipus de problemes** Bàsicament hi ha dos tipus de problemes: Problemes estàtics, on abans de començar a enviar coneixem quins missatges volem enviar i qualsevol missatge pot ser enviat des del primer pas, i els problemes dinàmics (també anomenats on-line) on els processadors poden enviar missatges a qualsevol moment, sense que al començament tinguem coneixement dels missatges que s'enviaran als propers passos.

En general, per al cas de problemes estàtics, ens interessa minimitzar el temps que tots els missatges triguen a arribar al seu destí, i en el cas de problemes dinàmics, el temps màxim d'espera per missatge. Però també pot ser important el considerar altres paràmetres, com la grandaria màxima de la cua a cada processador, o en lloc de fer estimacions sobre el temps pitjor, calcular el temps mitjà. Els algorismes per resoldre problemes d'encaminament d'informació en xarxes, representen una de les contribucions més importants cap al disseny de màquines de paral·lelisme massiu de propòsit general, ja que aquestes màquines utilitzen gran part dels seus recurssos encaminant les dades als processadors corresponents.

2.1 Vector lineal

La forma bàsica d'un **vector lineal** (Linear Array) està formada per n processadors connectats en línia. Cada processador té com a màxim dos veïns; un a la dreta i un altre a l'esquerra. Cada processador està connectat de manera bidireccional amb els seus veïns, com a la figura 2.1. A cada pas de la computació cada processador pot:

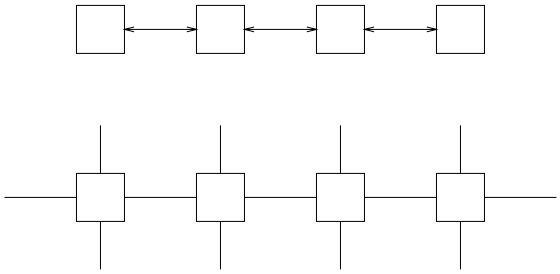


Figura 2.1: Un vector lineal.

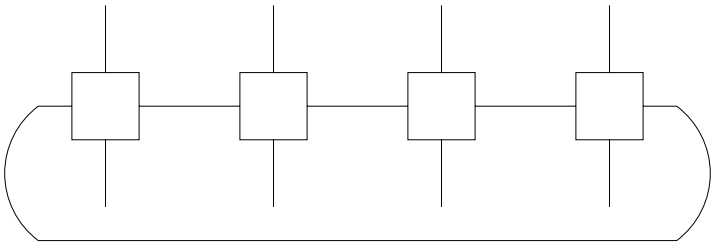


Figura 2.2: Un anell amb comunicació amb l'exterior.

1. rebre dades dels veïns,
2. realitzar una computació local utilitzant la memòria local,
3. passar dades als seus veïns,
4. actualitzar la memòria local.

El temps es divideix en passos. Per a això s'utilitza un rellotge global, de manera que tota la xarxa opera síncronament. Per analogia amb el moviment sistòlic del cor, els models sota consideració s'anomenen *sistòlics*.

Cal remarcar que només hem descrit la comunicació entre processadors, però que també podrem disposar de connexions per introduir les dades directament als processadors. Per exemple, en una xarxa com la de la figura 2.1, les línies d'entrada/sortida que no connecten cap processador es poden utilitzar per passar dades d'entrada o sortida. Així podem assumir que en un únic pas la xarxa es pot carregar o descarregar amb dades o bé que es pot anar carregant/descarregant per la dreta o per l'esquerra.

Una variació en el model bàsic consisteix en el que anomenarem el **tancament** de la xarxa; això vol dir que connectem el primer processador amb l'últim, com s'ha fet a la figura 2.2, de manera que tots els processadors tinguin el mateix nombre de connexions, és a dir, el mateix grau. Aquesta xarxa s'anomena **anell** (ring).

Exemple 2.1 (Classificació). *Donats n elements $x_1, \ldots x_n$, els volem classificar.*

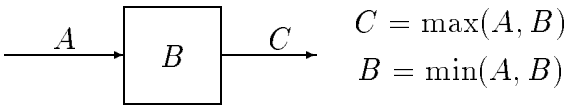


Figura 2.3: Operació realitzada per un processador a la primera fase de l'algorisme de classificació amb un vector lineal.

L'algorisme paral·lel que descriurem es basa en l'algorisme seqüencial d'*inserció*. Partint d'una taula inicialitzada amb un valor més gran que tots els elements per classificar, l'algorisme seqüencial processa els elements un per un i manté una llista classificada dels elements tractats. Quan arriba un nou element, troba la posició que li toca, desplaça els elements més grans i el col·loca al seu lloc. L'algorisme paral·lel manté la mateixa estructura, la diferència és que ara, tan bon punt comencem a tractar l'element x_i al pas següent, començarem a tractar x_{i+1} sense esperar que acabin els anteriors.

Utilitzem un vector lineal amb n processadors. Les dades entraran a cada processador des de la dreta cap a l'esquerra. L'algorisme sistòlic s'implementa en dues fases.

En la primera fase, cada processador fa el següent (vegueu la figura 2.3):

1. accepta l'entrada per l'esquerra,
2. compara el valor que entra amb el valor emmagatzemat a la memòria local,
3. emmagatzema el valor més petit a la memòria local,
4. envia cap a la dreta el valor més gran.

Quan la primera fase finalitza, el processador i conté el valor i-èsim a la seqüència ordenada.

En la segona fase, l'algorisme extreu la seqüència ja classificada, desde la dreta cap a l'esquerra. Una manera de fer-ho és que cada processador memoritzi la seva posició al vector. A més, cada processador té a la seva memòria local un comptador, de manera que després de $n - i + 1$ passos, el processador i comença a passar informació cap a l'esquerra. Una altra manera de fer-ho és que al final de la cadena d'entrada, quan un processador no rep més informació per la dreta (o rep un símbol especial), comença a passar informació cap al processador a l'esquerra (vegeu el problema **2:1**).

Respecte a la complexitat, el nombre de processadors és n i el temps és $\Theta(n)$; per tant, el cost és $C(n) = \Theta(n^2)$. Recordem que la classificació en seqüencial es pot implementar amb un cost de $\Theta(n \log n)$ passos; aleshores l'eficiència de l'algorisme és $\Theta(n \log n/n^2) = \Theta(\log n/n)$ i, per tant, podem concloure que per a valors grans de n l'algorisme paral·lel presentat és molt ineficient respecte als algorismes seqüencials més ràpids.

Observem que en el model presentat, a cada processador comparem mots sencers. És el que s'anomena el *model de mot* (word model), on cada operació de comparació o transmissió de mots sencers es realitza en un pas. Aquest és el model que utilitzarem quan parlem de SIMD per xarxa d'interconnexió. Però hem de ser conscients que per a mots molt llargs

o quan es vol entrar en detalls d'implementació física de la xarxa, el model de mot no és exacte i s'ha d'utilitzar el *model de bit* (bit model), on totes les operacions sobre mots es trenquen en operacions sobre bits i un pas de la computació correspon a la transmissió d'un únic bit o a la comparació de dos bits. Remetem el lector interessat en aquest model al llibre de Leighton [Lei93] .

Si en un vector lineal el nombre de processadors P és més petit que el nombre n d'elements per classificar, es poden dividir els n elements en P grups de manera que cada processador s'encarregui de n/P elements. Però per això s'ha d'assumir que cada processador pugui emmagatzemar, com a mínim, n/P elements. És necessari considerar la *granularitat* dels processadors. Un processador amb *granularitat molt fina* (fine grain processor) és el que té molt pocs registres de memòria i les mínimes operacions aritmètiques. Un processador amb *granularitat gruixuda* (coarse grain processor) és aquell que té una memòria gran i força potència de comunicació. En aquest llibre, assumirem que els processadors tindran la granularitat tan gruixuda com calgui.

De la mateixa manera, donades dues SIMD de xarxa d'interconnexió, X_1, X_2, amb processadors P_1 i P_2 respectivament, sigui $P_1 > P_2$. Qualsevol algorisme dissenyat per a X_1 es pot convertir en un algorisme per a X_2 a condició que els processadors de X_2 tinguin granularitat més gruixuda que els de X_1, és a dir, suficient granularitat perquè cada processador de X_2 simuli $\lceil P_1/P_2 \rceil$ processadors de X_1, i que la xarxa X_1 sigui suficientment regular i recursiva per tal de poder-la dividir en blocs que tinguin el mateix (o gairebé el mateix) nombre de processadors, de manera que si col·lapsem cada bloc fins a un node, obtenim una xarxa que es pot *encaixar* (embedded) a les connexions de X_2.

L'eficiència de l'algorisme resultant és equivalent a l'eficiència de l'algorisme original encara que el temps de càlcul no millori.

Utilitzant el mateix argument, es pot simular un vector lineal L_1, amb processadors P_1, fent servir un vector lineal L_2 amb processadors P_2. A cada processador p_i de L_2 se li assignen les tasques dels processadors amb subíndex entre $[\lceil P_1/P_2 \rceil (i-1)+1, \lceil P_1/P_2 \rceil]$. Aplicant-ho a l'exemple 4, si tenim un vector lineal amb $P < n$ processadors, podem ordenar les operacions en n passos, a cada pas cadascun dels P processadors executa els passos paral·lels dels $O(n/P)$ processadors que simula.

Exemple 2.2 (Classificació, mètode de la bombolla). *Utilitzarem ara un vector lineal amb n processadors per classificar n elements. L'algorisme que farem servir és una paral·lelització del mètode de la bombolla (bubble o odd-even transposition).*

Recordem que el mètode de la bombolla parteix d'una taula inicialitzada amb els valors per classificar. L'algorisme repeteix n fases. A la fase i es comparen dos a dos tots els elements a les posicions n fins a i, i s'hi fan els intercanvis necessaris. L'algorisme paral·lel fa en un pas totes les comparacions que no involucren elements comuns.

Suposem que comencem amb els n elements carregats al vector i que cada processador p_i conté un element. Volem finalitzar amb els elements classificats d'esquerra a dreta. L'algorisme és molt simple: A cada pas senar comparem els continguts dels processadors p_1 i p_2; p_3 i p_4; etc.; i a cada pas parell comparem els continguts dels processadors p_2 i

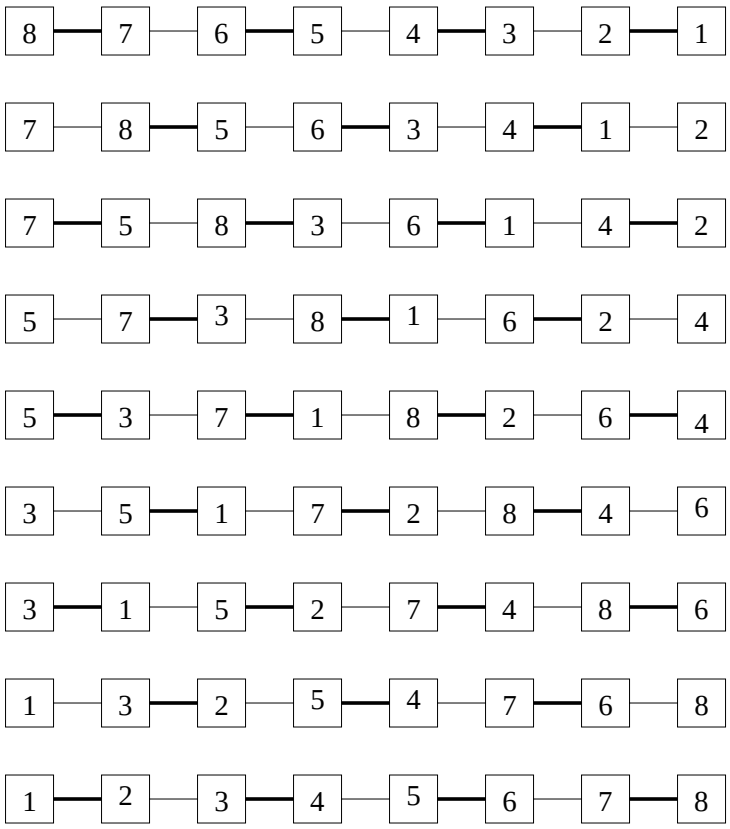


Figura 2.4: Un exemple d'aplicació de l'algorisme de la bombolla implementat amb un vector lineal.

p_3; p_4 i p_5; etc. Després de cada comparació, fem els intercanvis adients. Un exemple de l'aplicació de l'algorisme es dóna a la figura 2.4.

Ara demostrarem que aquest algorisme classifica n elements en n passos. Per fer-ho, primer demostrarem un lema general: el "lema de classificació de cadenes 0–1", també anomenat *"principi 0-1"*. Aquest lema es pot aplicar a algorismes de classificació que únicament utilitzen intercanvis. A més, les operacions d'intercanvi han de ser *preespecificades*, és a dir, les operacions de comparació i intercanvi han de ser independents dels resultats d'intercanvis previs. Els algorismes que compleixen aquesta propietat s'anomenen *algorismes preespecificats* (oblivious algorithm). Primer establirem la propietat bàsica dels algorismes preespecificats i en deixem la demostració com a exercici.

Lema 2.1. *Siguin* $\{x_1, \ldots, x_n\}$ *i* $\{y_1, \ldots, y_n\}$ *dues seqüències d'enters tals que si* $x_i \leq x_j$ *aleshores* $y_i \leq y_j$. *Si* A *és un algorisme de classificació preespecificat, la permutació calculada* $A(x_1, \ldots, x_n) = x_{\sigma(1)} \cdots x_{\sigma(n)}$ *és la mateixa que* $A(y_1, \ldots, y_n) = y_{\sigma(1)} \cdots y_{\sigma(n)}$, *on* σ *és una permutació del grup simètric* S_n.

El lema següent estableix el fet que en el cas d'algorismes preespecificats és suficient establir la correctesa de l'algorisme per a cadenes formades únicament per 0 i 1.

Lema 2.2 (Principi 0–1). *Si un algorisme de classificació preespecificat, que únicament utilitza operacions de comparació i intercanvi, classifica correctament qualsevol de les 2^n cadenes de $\{0,1\}^n$, aleshores el mateix algorisme classifica correctament qualsevol cadena amb n elements arbitraris.*

Demostració. Sigui A un algorisme preespecificat d'intercanvi que classifica correctament qualsevol cadena de $\{0,1\}^n$, però que no classifica correctament la cadena $x_1,\ldots,x_n$. Arribarem a una contradicció. Sigui $\pi \in S_n$ la permutació que classifica la cadena, és a dir, $x_{\pi(1)} \leq x_{\pi(2)} \leq \cdots \leq x_{\pi(n)}$. Sigui $\sigma \in S_n$ tal que $A(x_1,\ldots,x_n) = x_{\sigma(1)},\ldots,x_{\sigma(n)}$ és la sortida (no ordenada) del nostre algorisme d'ordenació, sigui k el menor enter tal que $x_{\sigma(k)} \neq x_{\pi(k)}$; sabem que per a tot $i<k$, $x_{\sigma(i)} = x_{\pi(i)}$, la qual cosa implica que $x_{\sigma(k)} > x_{\pi(k)}$ i, per tant, existeix un $r>k$ tal que $x_{\sigma(r)} = x_{\pi(k)}$. Definim la cadena

$$x_i^* = \begin{cases} 0 & \text{si } x_i \leq x_{\pi(k)} \\ 1 & \text{altrament} \end{cases}$$

Si $\{x_i^*\}_{i=1}^n$ és l'entrada, utilitzant el lema previ, l'algorisme A produeix com a sortida

$$x^*_{\sigma(1)},\ldots,x^*_{\sigma(k)},\ldots,x^*_{\sigma(r)},\ldots = 0,\ldots,0,1,\ldots,0,\ldots,$$

la qual cosa és una contradicció a la hipòtesi. □

Per demostrar que l'algorisme de la bombolla classifica n elements en n passos, hem de demostrar que funciona per a qualsevol seqüència de $\{0,1\}^n$. Considerem un vector amb n processadors, on cada processador té emmagatzemat un 0 o un 1. Sigui k el nombre total d'1. Hem de demostrar que en n passos, l'algorisme de la bombolla mou els k uns als processadors $n-k+1,\ldots,n$.

Sigui j_1 la posició on inicialment hi ha l'1 més a la dreta. Si j_1 és parell, l'1 no es comença a moure fins al segon pas. Altrament, al pas 1, el j_1 es mou cap a la dreta i a cada un dels passos següents continuarà anant cap a la dreta fins a arribar a la posició n. De la mateixa manera, podem considerar l'evolució del segon 1 més cap a la dreta, i succesivament arribarem fins que el k-èsim 1 més cap a la dreta comença a moure's al pas $k+1$-èsim, i no s'atura fins a arribar al processador $n-k+1$. Per tant, en n passos tindrem correctament classificada la cadena de $\{0,1\}^n$. Utilitzant el principi 0-1, l'algorisme de la bombolla classifica correctament en n passos qualsevol entrada amb n elements.

Exemple 2.3 (Producte matriu per vector). *Volem multiplicar una matriu A, de dimensió $m \times n$, i un vector b, amb n elements, utilitzant un vector lineal amb n processadors.*

Recordem que el vector producte $c = A\,b$ ve donat per

$$c_i = \sum_{j=1}^{n} a_{ij} b_j.$$

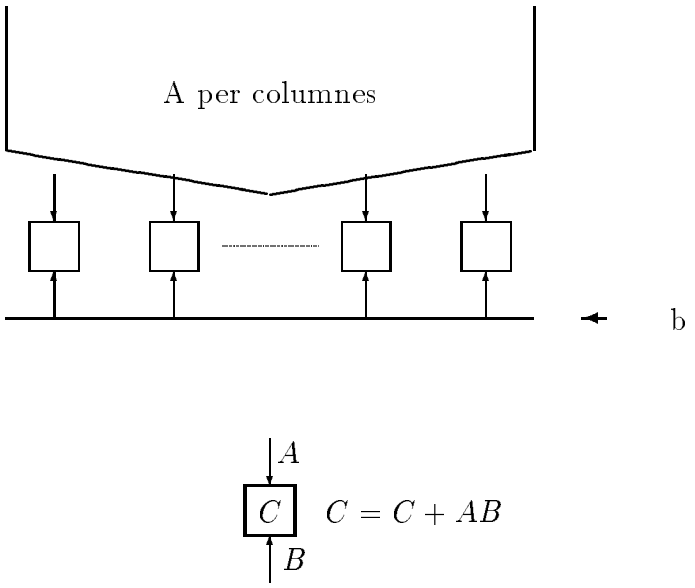


Figura 2.5: Producte d'una matriu per un vector amb transmissió simultània.

Aquesta fórmula es pot reescriure com una recurrència,

$$\begin{aligned} c_i^{(0)} &= 0, \\ c_i^{(j)} &= c_i^{(j-1)} + a_{ij}b_j. \end{aligned}$$

Considerem que cada processador té quatre entrades/sortides; una a dalt, una a l'esquerra, una a baix i una a la dreta. Si els valors de b es transmeten a tots els processadors en temps constant utilitzant l'entrada de baix (a través d'un bus de dades), cada processador a cada pas de la computació fa el següent:

1. accepta l'entrada per baix,
2. accepta l'entrada per dalt,
3. multiplica els dos valors que rep i els emmagatzema a la memòria local.

Quan entrem la fila i-èsima de la matriu per l'entrada de dalt del processador i, a cada pas entra una columna de A a la mesh, i aquest algorisme calcula a cada pas j els elements $c_i^{(j)}$ de la recurrència (vegeu la figura 2.5).

Per resoldre el problema de la transmissió simultània de dades, entrem el vector b per l'esquerra i endarrerim l'entrada de les files de la matriu, de manera que la fila i entri i avanci una unitat de temps després de la fila $i-1$ (vegeu la figura 2.6). Amb aquest decalatge aconseguim que des de l'instant en què s'activa cada processador, a cada pas rebi les dades que necessita. Inicialment tots els processadors i estan inicialitzats a 0. Quan el processador i rep entrades a i b, realitza les accions següents a la seva memòria local:

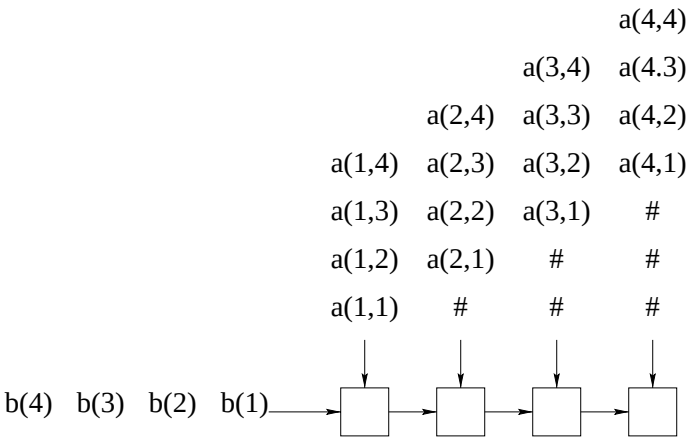


Figura 2.6: Producte d'una matriu per un vector.

1. les multiplica,
2. afegeix el resultat a c_i,
3. si $j \neq n$, envia b cap a $p(i+1)$.

Observem que totes aquestes operacions tenen un cost constant de 3. Per tant, el nombre de passos necessaris per multiplicar una matriu $m \times n$ per un vector amb n elements és $O(n)$ i el nombre de processadors és $O(n)$, la qual cosa dóna un cost $O(n^2)$.

2.2 La mesh

Aquesta topologia consisteix en una taula bidimensional en forma de rectangle amb $m \times k$ processadors que es comuniquen de forma bidireccional amb els seus veïns. Denotarem per (i, j) el processador corresponent a la fila i-èsima i la columna j-èsima. Els processadors poden comunicar-se de diferents maneres. La més simple és la comunicació directa del processador (i, j) amb els processadors veïns $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ i $(i, j+1)$. Tanmateix, també es poden comunicar per connexions diagonals o bé es poden considerar els models **tancats**, en què cada fila i/o columna constitueix un anell. Quan tanquem només les files (o només columnes), obtenim un *cilindre*; quan tanquem files i columnes, tenim la xarxa anomenada *torus*. A la figura 2.7 s'il·lustren algunes de les topologies possibles. Observem que per $k = 1$ tenim un vector lineal.

A la mesh també podem considerar que tenim tantes connexions d'entrada o sortida com calguin; així assumirem que cada processador té connexions a qualsevol dels seus quatre veïns i una altra connexió per introduir o treure dades directament a cada processador.

Exemple 2.4 (Multiplicació de matrius). *Vegem ara com multiplicar dues matrius, A i B, de dimensions $m \times n$ i $n \times k$, respectivament, utilitzant una mesh $m \times k$.*

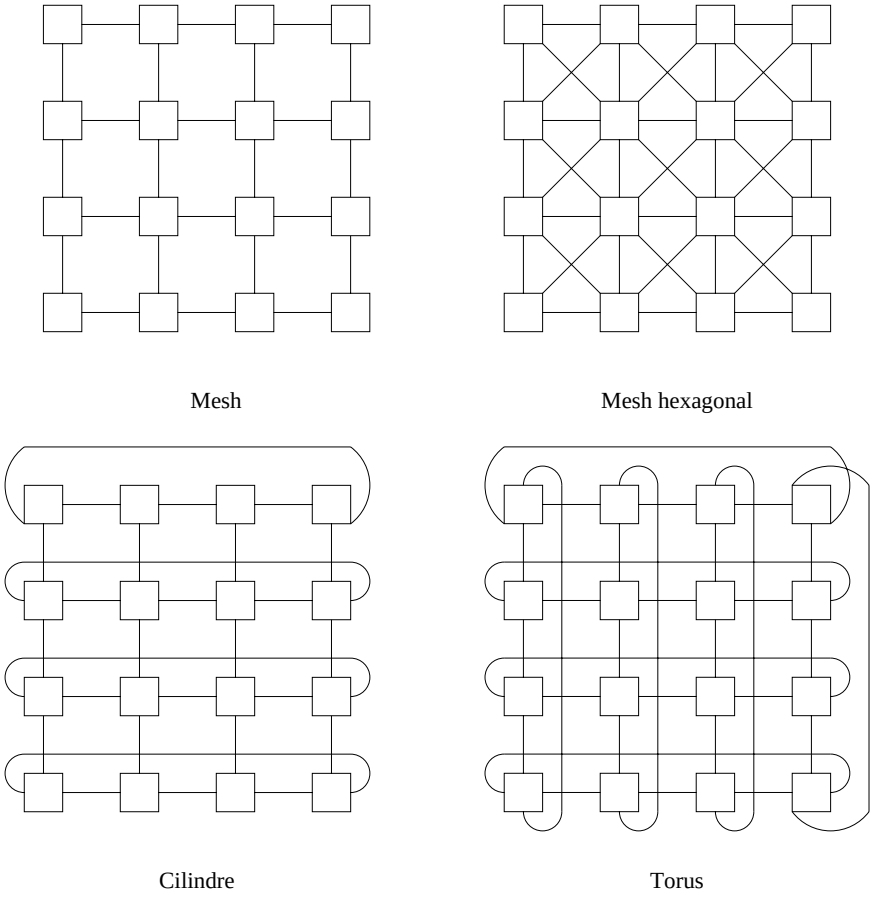


Figura 2.7: Diferents maneres de connectar els processadors d'una mesh.

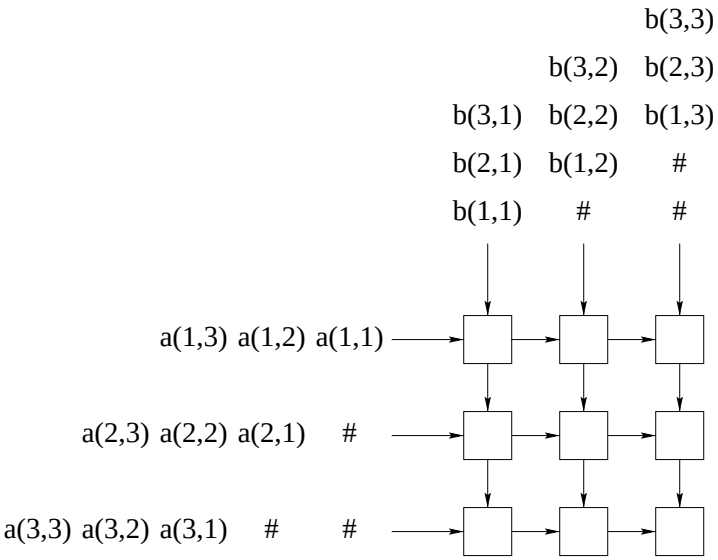


Figura 2.8: Multiplicació de matrius 3×3.

La matriu producte $C = A \times B$ ve donada per

$$c_{ij} = \sum_{s=1}^{n} a_{is} b_{sj}.$$

Recordem que al capítol 1 vam presentar l'algorisme seqüencial més simple per multiplicar matrius. L'algorisme per a la mesh és una generalització de l'algorisme presentat a l'exemple 6.

Les matrius A i B entren simultàniament a la mesh, A per files i B per columnes, de manera que la fila i-èsima de A entra i avança una unitat de temps després que la fila $i-1$-èsima. De la mateixa manera, la columna j-èsima de B va endarrerida una unitat de temps respecte de la columna $j-1$-èsima. D'aquesta manera, s'assegura que a_{is} es troba amb b_{sj} al processador (i, j). En finalitzar l'algorisme, el resultat c_{ij} és emmagatzemat al processador (i, j).

Inicialment tots els processadors (i, j) estan inicialitzats a 0. Cada cop que el processador (i, j) rep entrades a i b, realitza les accions següents a la seva memòria local:

1. les multiplica,
2. afegeix el resultat a c_{ij},
3. si $j \neq k$, envia a cap al processador $(i, j+1)$,
4. si $i \neq m$, envia b cap al processador $(i+1, j)$.

Observem que totes aquestes 4 operacions tenen un cost constant de 4. Per tant, el nombre de passos necessaris per multiplicar dues matrius $n \times n$ és $O(n)$ i el nombre de

processadors és $O(n^2)$, la qual cosa dóna un treball $O(n^3)$. Per tant, l'algorisme no és òptim.

La mesh és una topologia molt adient per a la manipulació de matrius. La representació d'un graf per la seva matriu d'adjacència permet implementar amb una mesh molts dels algorismes per problemes sobre grafs.

Exemple 2.5 (Clausura transitiva). *Vegem com computar amb una mesh $n \times n$ la clausura transitiva d'un graf $G = (V, E), V = \{1, 2, \ldots, n\}$.*

Recordem que, donat un graf $G = (V, E)$, $|V| = n$, la seva **matriu d'adjacència** $A = (a_{ij})$ ve definida per

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{altrament} \end{cases}$$

La **clausura transitiva** $G^* = (V, E^*)$ de G ve definida pel conjunt d'arestes

$$E^* = \{(i,j) | \text{ existeix un cami de } i \text{ fins a } j\}$$

La matriu d'adjacència de la clausura transitiva de G^* és la matriu $A^* = (a^*_{ij})$.

És ben conegut que $A^* = \sum_{i=1}^{n} A^n$, on A denota la matriu d'adjacència de G i, com que A és una matriu binària, la $+$ representa la disjunció i el producte la conjunció. Una manera de computar G^* és implementar el càlcul de A^* amb una mesh tancada $n \times n$, la qual cosa ens dóna un algorisme amb $O(n \log n)$ passos (vegeu l'exercici **2:6**).

De fet, al capítol següent veurem un algorisme més eficient per calcular la clausura transitiva.

Exemple 2.6 (Classificació). *Classifiquem n enters col·locats a la memòria dels processadors d'una mesh de $\sqrt{n} \times \sqrt{n}$.*

Sense perduda de generalitat, podem assumir que n és un quadrat perfecte. En finalitzar l'algorisme, els n enters estaran classificats de la manera següent: començant pel processador $(1,1)$ fins al $(1, \sqrt{n})$, els elements següents vindran als processadors $(2, \sqrt{n})$ fins al $(2,1)$, després el $(3,1)$, etc. Els enters ordenats formen una "serp" dintre de la mesh, com a la figura 2.9.

L'algorisme ve estructurat en fases. Una fase és una mena de "macropas".

1. En les fases senars classifica cada fila de la mesh, de manera que en les files parelles els nombres més petits van cap a l'esquerra, i en les files senars els més petits van cap a la dreta.

2. En les fases parelles, classifica cada columna de manera que els nombres més petits vagin cap a dalt de la mesh.

Per realitzar les classificacions a cada fase de l'algorisme, considerem cada fila o cada columna de la mesh com un vector lineal independent amb $\sqrt{n}$ processadors i cadascun utilitza l'algorisme de la bombolla descrit anteriorment. Així, doncs, el temps total serà el nombre de fases per $\sqrt{n}$. A la figura 2.10 és dona un exemple de l'execució d'aquest algorisme.

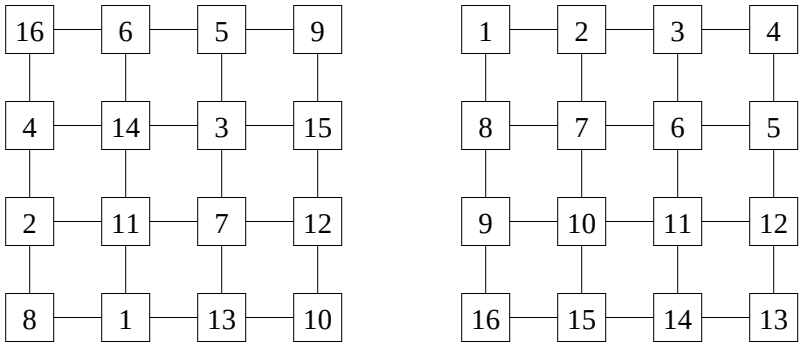


Figura 2.9: La mesh abans i després de la classificació.

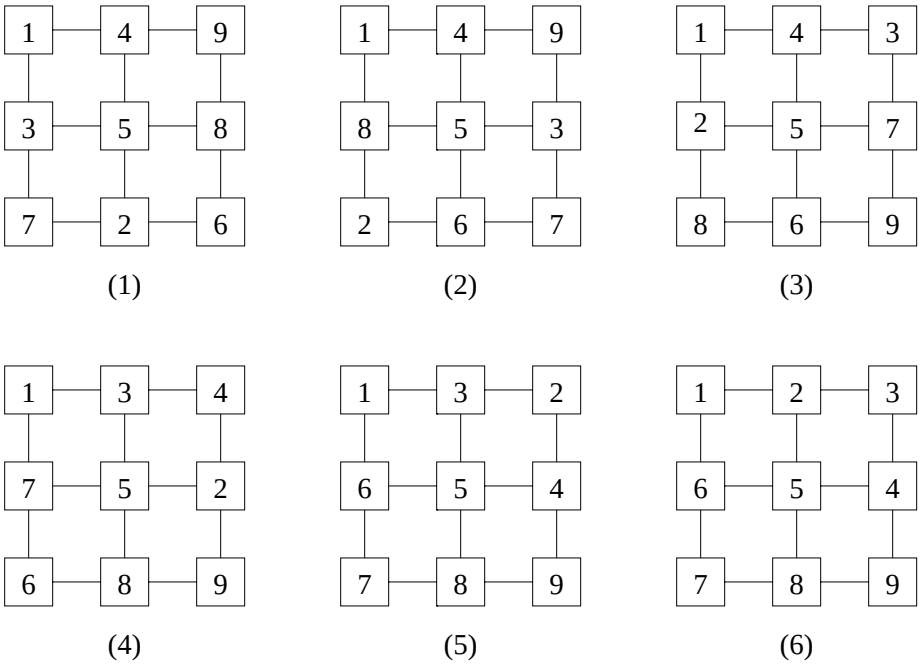


Figura 2.10: Un exemple de classificació a la mesh.

Teorema 2.1. *L'algorisme descrit classifica correctament n enters en $\log n + 1$ fases. Per tant, el nombre total de passos és $O(\sqrt{n}\log n)$*

Demostració. Com que es tracta d'un algorisme preespecificat, podem utilitzar el principi 0-1. Pera això cal demostrar que l'algorisme classifica correctament qualsevol seqüència de $\{0,1\}^n$, en $O(\sqrt{n}\log n)$.

Direm que una fila o columna és *homogènia* si tots els processadors a la fila contenen 0 (o contenen 1). Demostrarem que, a cada fase, com a mínim la meitat de les files no homogènies es converteixen en homogènies. Dividim les files de la mesh en tres regions: la regió superior de files homogènies amb només zeros, la regió inferior amb tot uns, i la regió del mig amb files no homogènies. Agrupem les files no homogènies en parells consecutius i estudiem l'efecte de classificar les dues files: la primera fila tindrà els zeros a l'esquerra i la segona tindrà els zeros a la dreta. Per a cada parell de files homogènies consecutives, hi ha tres possibles sortides de la classificació:

- Si el parell té més 0s que 1s, després de la classificació hi haurà unes posicions al mig en què les dues files seran 0.
- Si hi ha més 1s que 0s, després de la classificació hi haurà unes posicions al mig en què les dues files seran 1 .
- Si hi ha igual nombre de 0s que d'1, després de la classificació, on a la primera fila hi hagi un 0 a l'altra hi haurà tindrem un 1, i viceversa.

Si classifiquem cada columna amb dos elements, col·loquem el més petit a dalt. En el primer cas, la primera fila només tindrà zeros; en el segon cas, la segona fila serà tot 1, i en el tercer cas la primera fila serà tot 0 i la segona tot 1. És a dir, cada dues files es produeix una fila homogènia i, després de les dues fases, com a mínim la meitat de les files passen a ser homogènies.

Com que a cada fase reduïm la grandària de la regió del mig com a mínim a la meitat, després de $2\log\sqrt{n}$ fases haurem classificat totes les files excepte una. Fem una classificació d'aquesta darrera fila i ja tenim tots els elements classificats. El nombre total de fases és $\log n + 1$, i el nombre total de passos és $O(\sqrt{n}\log n)$, amb la qual cosa tenim un treball de $O(n\sqrt{n}\log n)$ i una eficiència de $O(\log n/(\sqrt{n}\log n))$. □

Per finalitzar, hem de considerar el protocol perquè cada processador conegui quan ha d'aturar una fase i començar la propera. Ho deixem com a exercici **2:5**.

2.3 Paràmetres d'una topologia

Els models de SIMD que hem analitzat fins ara són simples i eficients per a alguns tipus d'algorismes. En general, amb vista a obtenir temps molt ràpids per a qualsevol algorisme, es necessiten dues característiques de la topologia utilitzada. La primera característica és que el *diàmetre* de la xarxa sigui petit respecte al nombre total de processadors. El

diàmetre es defineix com el màxim de la distància entre parell de processadors qualssevol a la xarxa. El diàmetre d'una xarxa és una fita inferior al temps de computació, ja que el resultat de la computació realitzada per un processador s'ha de transmetre als altres processadors. En els models estudiats aquest paràmetre és gran. Per exemple, el diàmetre d'una mesh de $n \times m$ és $m+n-2$. La segona característica que ens interessa és l'*amplada de bisecció* (bisection width). **L'amplada de bisecció** d'una xarxa es defineix com el mínim nombre d'arestes que hem d'eliminar per particionar la xarxa en dues xarxes de la mateixa grandària. La relació entre l'amplada de bisecció i la velocitat amb què una xarxa pot implementar un algorisme es deu al fet que les dades contingudes i computades per una meitat de la xarxa s'han de distribuir a l'altra meitat, i l'amplada de bisecció indica el "coll d'ampolla" del pas de dades entre les dues meitats: a amplada de bisecció més gran, menys problemes de pas de dades. Les topologies estudiades fins ara tenen una amplada de bisecció petita. Per exemple, una mesh $n \times m$ amb $m \geq n$ té una amplada de bisecció de m si m és parell, o $m+1$ altrament, i un diàmetre gran.

La majoria dels algorismes descrits tenen la propietat que no ens hem de que preocupar gaire de problemes de transmissió de dades. En aquesta secció introduirem el problema de l'*encaminament de paquets* (packet routing), per a les topologies estudiades. Donada una topologia específica, per exemple, una mesh de $\sqrt{n} \times \sqrt{n}$, considerarem m paquests d'informació. Cada paquet conté una dada i la direcció del processador on va destinada, i els m paquets estan distribuïts entre els n processadors. El problema de l'encaminament consisteix a fer arribar cada paquet a la seva destinació amb el mínim nombre de passos. Assumirem que cada paquet va destinat a un processador diferent. A més, assumirem que dos paquets poden recórrer al mateix temps la mateixa aresta únicament si van en direccions contràries. En qualsevol pas intermedi, un processador pot contenir més d'un paquet.

2.4 Encaminament voraç

En aquest llibre analitzem un cas particular d'encaminament, anomenat el *problema de l'encaminament d'una permutació* (full permutation routing). Cada processador conté un paquet destinat a un altre processador i, a més, hi ha un paquet destinat a cada processador. Per tant al final, cada processador haurà rebut un paquet.

Veguem com resoldre aquest problema amb un vector lineal amb n processadors. Al començament cada processador conté un paquet destinat a un altre processador i, per tant, al final també cada processador contindrà un paquet. Una estratègia bona és que a cada pas, cada paquet que necessita desplaçar-se cap a l'esquerra (o cap a la dreta) ho faci. Aquesta és la tipica estratègia "voraç" (greedy) i, per al cas de vectors lineals, està ben definida, ja que en cap moment dos paquets intentaran travessar la mateixa aresta en la mateixa direcció. Cada paquet arriba a la seva destinació en d passos, on d és la distància que hi ha del processador on el paquet era al començament, fins al processador de destinació. Per tant, la complexitat de l'encaminament voraç a un vector lineal amb n processadors és $O(n)$.

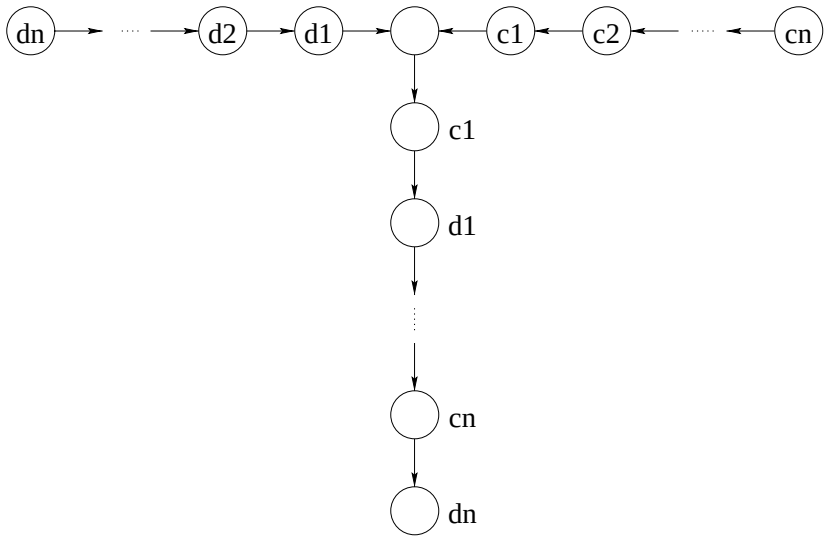


Figura 2.11: Un cas de col·lisió en una mesh, amb una cua de prioritat amb n elements.

Quan considerem una mesh $\sqrt{n} \times \sqrt{n}$, l'algorisme voraç no funciona tan bé. La raó d'això és que, a causa de la topologia, és possible que dos o més paquets siguin forçats a passar per la mateixa aresta, en la mateixa direcció al mateix temps (vegeu la Figura 2.11). En aquest cas, un dels paquets s'haurà d'esperar fins que passi l'altre, és a dir, s'haurà de posar a la cua.

En el cas de la figura 2.11, la situació és encara pitjor ja que en lloc de fer arribar dos paquets al mateix processador per continuar en la mateixa direcció, hi fem arribar dues cadenes de paquets. En aquest cas, la cua pot ser de fins a $\sqrt{n}/2$ elements i tenint en compte que, a cada pas, el processador del mig rep dos paquets a cada instant i només en pot transmetre un, la cua pot arribar a ser de fins a $\sqrt{n}/2$ elements. Clarament és necessari articular un mecanisme que reguli l'arribada de paquets als nusos de manera que no es formin grans cues, i un protocol per tractar les cues que es formin. Un del protocols més utilitzats és l'anomenat *prioritat al que va més lluny* (farthest-first), que dóna prioritat als paquets que viatgen més lluny. Amb aquest protocol considerem l'algorisme voraç d'encaminament que primer encamina cada paquet cap a la columna correcta i després fins a la seva destinació final a aquesta columna. Aquest algorisme s'anomena *algorisme voraç bàsic per a la mesh*. Vegem-ne la complexitat en el cas pitjor, quan apliquem l'algorisme a una mesh $\sqrt{n} \times \sqrt{n}$.

El lema següent analitza el comportament, en el cas pitjor, de l'algorisme voraç en un vector lineal quan considerem que cada processador pot tenir més d'un paquet per enviar i utilitza el protocol de prioritat per al que va més lluny.

Lema 2.3. *Donat un vector lineal amb n processadors, on cada processador pot contenir un nombre arbitrari de paquets, però al final cada processador conté un únic paquet, tots els paquets arribaran a la seva destinació en $n-1$ passos, utilitzant l'algorisme voraç amb*

protocol de prioritat per al que va més lluny.

Demostració. En el cas pitjor tindrem $n-1$ paquets en el primer processador, destinats a cada un dels altres processadors. En aquest cas, el paquet destinat al processador n trigarà $n-1$ passos, però en aquest temps tota la resta de paquets hauran arribat a la seva destinació. □

Aquest resultat és bàsic per analitzar el comportament de l'algorisme voraç per a la mesh.

Teorema 2.2. *El nombre màxim de passos necessaris per encaminar correctament n paquets a una mesh $\sqrt{n} \times \sqrt{n}$, utilitzant l'algorisme voraç bàsic, és de $2\sqrt{n}-2$ passos, i utilitza com a màxim una cua de $2/3\sqrt{n}-1$.*

Demostració. Per demostrar el teorema, observem que, segons el lema anterior, cada paquet arriba a la columna que li correspon en, com a màxim $\sqrt{n}-1$ passos, i coneixem que l'encaminament dintre d'una columna pot trigar fins a $\sqrt{n}-1$ passos. Per tant, tots els paquets arriben a la seva destinació final en $2\sqrt{n}-2$ passos, com a màxim.

Per veure la grandària màxima de pila, vegem que el cas pitjor és dóna quan tots (o gairebé tots) els paquets amb destinació a una mateixa columna arriben a través d'un únic processador i s'utilitzen les tres connexions d'entrada. En aquest cas, de cada tres paquets que hi arribin només un es pot transmetre. A la figura 2.12 considerem que els $\sqrt{n}-2$ paquets als processadors $(1,2),(1,3),\ldots,(1,\sqrt{n}/3)$ i $(2,1),(2,2),\ldots,(2,(2\sqrt{n}/3)-1)$ tenen com a destinació els processadors $(3,\sqrt{n}/3),(4,\sqrt{n}/3),\ldots,(\sqrt{n},\sqrt{n}/3)$. Tots aquests paquets arriben en $\sqrt{n}/3-1$ passos al processador $(2,\sqrt{n}/3)$, però en aquest nombre de passos únicament $\sqrt{n}/3-1$ paquets poden passar per l'aresta que va del processador $(2,\sqrt{n}/3)$ al processador $(3,\sqrt{n}/3)$ i, eventualment, la cua de paquets esperant creuar aquesta aresta arribarà a ser de fins a $2/3\sqrt{n}-1$, amb la qual cosa el teorema queda demostrat. □

2.5 Hot potato routing

L'algorisme voraç es comporta bé, però pot arribar a utilitzar piles de fins $\sqrt{n}/2$ elements. Anem a presentar un altre tipus d'algorisme que triga més passos però utilitza una cua de grandària constant. Considerem una *mesh* $n \times n$ que conté x paquets, $(x \leq n^2)$ cadascun direccionat a un processador diferent. L'algorisme consta de tres fases

1. Classificar utilitzant l'agorisme , els paquets fent servir com a clau la columna on van adreçats, i col·locant-los classificats a la columna 1, després a la 2, etc..

2. Encaminar cada paquet per la fila on està, fins arribar a la seva columna.

3. Encaminar cada paquet per la seva columna, fins arribar al seu desti.

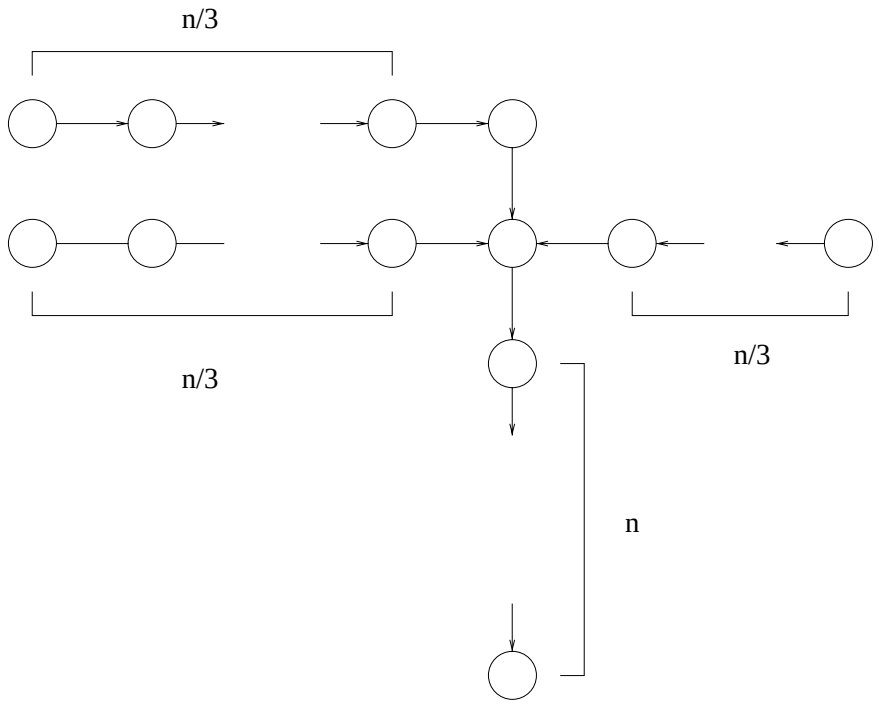


Figura 2.12: El cas pitjor per la formació de cues en l'algorisme voraç.

Quan finalitzem la fase 1, per a qualsevol i, j, existeix com a màxim 1 paquet a la fila i adreçat cap a la columna j, altrament hi haurien $n+1$ paquets direccionats a processadors situats a la fila j, cosa que va contra el nostre model de xarxa. Les fases 2 i 3 necessiten $2n-2$ passos. Per tant el temps total necessari és de $6n + o(n)$ passos.

Canviarem el tipus de transmissió a *deflecció* o Hot Potato. Quan un paquet arriba a un nus, ha de sortir immediatament; si no pot sortir per l'aresta òptima ha de fer-ho per qualsevol altra, però els processadors a la mesh no tenen buffer.

Un primer algorisme de deflecció per a qualsevol mesh $n \times m$ està basat en el fet que una mesh $n \times n$ amb n parell, conté un circuit Hamiltonià (un recorreegut que partint d'un nus passa per cada nus de la *mesh* un sol cop, fins arribar al nus de partida). Per tant. si tenim una *mesh* $n \times n$, fixem un circuit Hamiltonià i encaminem tots els paquets seguint aquest circuit, fins arribar a la seva destinació, on es queden mentre que la resta continuen. El màxim nombre de passos que un paquet realitzarà serà $n^2 - 1$. Per tant, aquest algorisme utilitza un nombre de passos $O(n^2)$. Per millorar aquest algorisme, considerem que cada paquet es mou de cap a la dreta i despres cap a l'esquerra a la fila on ès, fins arribar a la seva columna; si quan passa per la seva columna no pot entrar-hi (per congestió) aleshores continua bellugant-se per la fila. Un cop pot començar a moure's per la seva columna, es belluga per aquesta fins arribar al nus de destinació. Notem que pot començar a moure's per la columna en la direcció contrària a la que vol anar.

Teorema 2.3. *Donada una mesh $n \times n$ qualsevol permutació de paquets, serà encaminada*

correctament amb l'algorisme previ en $O(n^{1.5})$ passos.

Demostració. Partim l'execució de l'algorisme en intervals de $4n$ passos cada interval. Per a una columna j, sigui N_i el nombre de paquets destinats a un nus a la columna j, que al començament del i-ésim interval encara no han arribat a la seva destinació. Tenim que $N_1 \leq n$. Direm que un paquet està bloquejat si quan intenta girar per la seva columna no pot (degut al fet que dos paquets a la mateixa columna l'impedeixen pujar o baixar). Notem que dos paquets poden bloquejar com a màxim un paquet.

També per qualsevol parell de paquets viatjant per la mateixa columna, es trobaran com a màxim 2 cops abans d'arribar a la seva destinació,

$$2\binom{l}{2} \geq N_1 - l \Rightarrow l^2 - l \geq N_i - l$$
$$\Rightarrow l \geq \left\lceil \sqrt{N_i} \right\rceil .$$

Com que cada un dels l paquets arriba abans o durant els $2n$ passos $N_{i+1} \leq N_i - \lceil \sqrt{N_i} \rceil$. Però si $x \in \mathbb{N}$ tenim

$$N_i \leq x^2 \Rightarrow N_{i+1} \leq (x-1)^2 \tag{2.1}$$

ja que altrament tindríem

$$\begin{aligned}
N_{i+1} &> (x-1)^2 \\
&\Rightarrow (x-1)^2 < N_{i+1} \leq N_i < x^2 \\
&\Rightarrow x - 1 < \sqrt{N_{i+1}} \leq \sqrt{N_i} \leq x \\
&\Rightarrow \left\lceil \sqrt{N_{i+1}} \right\rceil = \left\lceil \sqrt{N_i} \right\rceil = x \\
&\Rightarrow N_{i+1} \leq N_i - \left\lceil \sqrt{N_i} \right\rceil \leq x^2 - x \\
&\Rightarrow N_{i+2} \leq N_{i+1} - \left\lceil \sqrt{N_{i+1}} \right\rceil \leq (x^2 - x) - x \\
&< (x-1)^2.
\end{aligned}$$

Per tant de l'equació 2.1 podem obtenir una fita al nombre d'intervals. Com que $N_1 \leq n \leq \lceil \sqrt{n} \rceil^2$, utilitzant inducció sobre k tenim $N_{2k+1} \leq (\lceil \sqrt{n} \rceil - k)^2$ és a dir $N_{2\lceil \sqrt{n} \rceil + 1} \leq 0$ el que implica que el nombre d'intervals és com a màxim $2\lceil \sqrt{n} \rceil$. Com que cada interval pren $4n$ passos, el temps total de l'algorisme és com a màxim $8n\lceil \sqrt{n} \rceil \leq 8n^{1.5} + 8n$. □

2.6 Referències bibliogràfiques

La millor referència sobre SIMD de xarxes d'interconnexió és el llibre de Leighton [Lei93]. Altres referències generals sobre sistòlics i pipeline per SIMD amb topologia de xarxa d'interconnexió són els llibres [Akl89], [Kun89] i l'article clàssic [FK80]. Existeixen algorismes molt més eficients per al problema de l'encaminament en mesh, vegeu, per exemple, la secció 1.7.2 del [Lei93]).

Exercicis

2:1 Expliciteu amb cura l'algorisme que resol el problema de la classificació sobre un vector lineal, explicat al text.

2:2 Expliqueu com fer la transposició d'una matriu $n \times n$, utilitzant una mesh. Demostreu que la complexitat és $O(n)$.

2:3 Demostreu el lema 1.

2:4 En l'algorisme de l'ordenació de n elements amb una mesh de $\sqrt{n} \times \sqrt{n}$ processadors, doneu un mecanisme perque cada processador sàpiga quan s'ha d'aturar una fase i començar la propera. (Ajuda: utilitzeu 2 rellotges).

2:5 Finalitzeu la demostració de l'exemple 2.6. És a dir, doneu un protocol de com cada processador sap quan ha de començar cada fase de l'algorisme.

2:6 Dissenyeu un algorisme sistòlic per a la mesh tal que, donada una matriu A, ens calculi la matriu $B = A^2 + A$. Suposeu que l'element a_{ij} és al processador P_{ij} al començament i que al final tindrà el valor de Bij.

2:7 Digueu com encaminarà un vector lineal amb n processadors qualsevol parell de permutacions, utilitzant com a màxim n passos.

2:8 Una mesh tridimensional és un cub $n \times n \times n$ de processadors. Dissenyeu un algorisme sistòlic per calcular el producte de dues matrius amb dimensió $n \times n$.

2:9 Dissenyeu un algorisme per a la mesh tal que, donat un conjunt de n^2 valors diferents emmagatzemats als processadors de la mesh, al final de l'algorisme cada processador conegui el nombre i del processador que conté el valor màxim del conjunt.

2:10 Modifiqueu l'algorisme del problema anterior per al cas que s'admetin valors repetits. En aquest cas, dos o més processadors poden tenir el mateix identificador.

2:11 Demostreu que qualsevol torus amb $n \times n$ processadors no conté un circuit Hamiltonia. (Per tant existeix un algorisme de defleccio per a transmissió que funciona en $O(n^2)$ passos.) Com implementarieu l'algorisme?

Capítol 3

El model semisistòlic de computació

Una xarxa SIMD, que permet una utilització limitada d'algunes operacions globals com transmissió de dades o acumulació, s'anomena *model semisistòlic*. Cada processador en una xarxa SIMD es pot considerar com una unitat lògica connectada a uns registres de memòria. La unitat lògica realitza les computacions i els registres emmagatzemen les dades i tenen el comptador de passos. En una xarxa sistòlica, les connexions entre processadors es realitzen de manera que a cada processador les dades li entren per la unitat lògica, després van als registres, d'on surten cap al proper processador. En una xarxa semisistòlica la sortida de cada processador pot fer-se directament de la unitat lògica o des de la sortida dels registres. Quan unim més d'un processador directament per les unitats lògiques, podem passar missatges en un pas de computació a tots els processadors connectats directament, ja que el temps es comptabilitza en passar per un registre. La flexibilitat que es guanya amb el model semisistòlic té un preu; el temps "real" d'implementar cada pas d'un model semisistòlic és, com a mínim, el temps de transmetre totes les dades a totes les unitats lògiques connectades directament.

El que més ens interessa de les xarxes semisistòliques és que habitualment el disseny d'un algorisme per a una xarxa semisistòlica és molt més senzill que per a una xarxa sistòlica i, en determinades condicions, a partir d'un algorisme semisistòlic podem obtenir un de sistòlic que resol el mateix problema. Al capítol 2 hem dissenyat un algorisme per implementar el producte d'una matriu per un vector pensant en la transmissió simultània a través d'un bus de dades. Una cop ja tenim un algorisme semisistòlic, el flux de dades es pot representar mitjançant un graf dirigit (digraf) amb pesos, on el pes de cada aresta representa el nombre de registres que hi ha a la línia de comunicació i els nusos representen els processadors. Amb aquest digraf associat veurem quines condicions hem d'imposar per poder obtenir un algoritme sistòlic que resolgui el mateix problema.

3.1 Sistolització

Per estudiar la conversió d'una xarxa semisistòlica en una de sistòlica, comencem per representar cada xarxa per un digraf amb pesos, on les unitats lògiques es representen

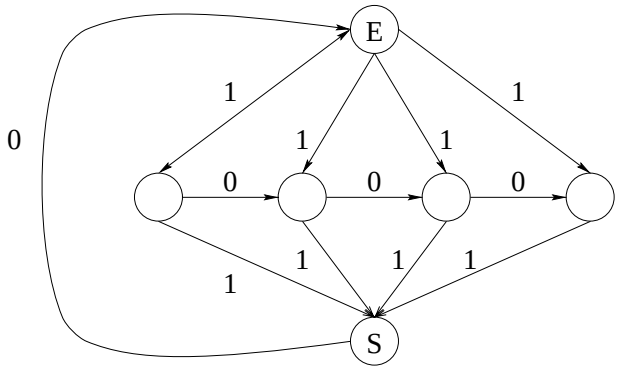


Figura 3.1: El graf X associat a la xarxa semisistòlica per al càlcul del producte d'una matriu per un vector.

com els nusos, les arestes indiquen unitats lògiques adjacents i els pesos sobre les arestes representen el nombre de registres en la connexió entre les unitats lògiques. Aleshores, una xarxa sistòlica ens dóna un graf associat, on les arestes tenen pesos positius, i una xarxa semisistòlica té com a graf associat un graf on les arestes tenen pesos no negatius i cicles positius.

Cada xarxa té un **hoste**, que és el processador a través del qual entra o surt el flux de dades de la xarxa. De fet, tindrem dos hostes, un d'entrada i un altre de sortida, i assumirem que hi són connectats per una aresta amb pes 0 que va de l'hoste de sortida a l'hoste d'entrada amb pes 0. A la figura 3.1 s'ilustra el graf associat a la xarxa semisistòlica per al càlcul del producte matriu per vector de la figura 2.5. Hem suposat que a cada pas el valor calculat es transmet per la connexió de baix, encara que aquest valor només té sentit quan el càlcul finalitza. La transformació d'una xarxa semisistòlica en una de sistòlica equivalent consisteix a transformar el digraf de la primera en un digraf tal que tot nus computa el mateix i totes les arestes tenen pes positiu.

Existeix un mètode anomenat **retemporització** (retiming) que, sota certes condicions, transforma una xarxa semisistòlica en una de sistòlica. L'operació de retemporitzar un nus consisteix a eliminar (o afegir) un registre de les arestes que arriben al nus, i afegir (o eliminar) un registre de les arestes que en surten. A la figura 3.2 es dóna un exemple de retemporització positiva i negativa. Observem que perquè la xarxa resultant després d'una retemporització positiva continuï essent semisistòlica, cal que les arestes de sortida tinguin pes positiu, i també les arestes d'entrada en el cas d'una retemporització negativa. L'operació de retemporització és estrictament local: cada nus incrementa o decrementa el temps amb què una dada arriba a ell i, a continuació, endarrereix o avança amb la mateixa quantitat el temps que triga a sortir. Per tant, el temps de computació global no canvia.

Definició 3.1. *Dues xarxes semisistòliques són equivalents si la topologia dels seus digrafs són les mateixes, els nusos dels dos digrafs realitzen la mateixa computació, les dues xarxes tenen les mateixes entrades/sortides i hi ha una seqüència de retemporitzacions locals que*

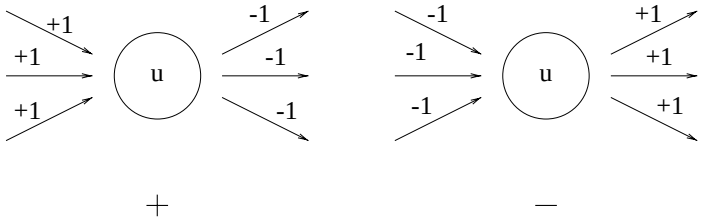


Figura 3.2: Retemporització positiva i negativa.

permeten passar del graf associat a l'una al graf associat a l'altra, passant sempre per xarxes semisistòliques.

Ens interessa veure en quines condicions, partint d'una funció global, que anomenarem *funció lag*, que assigna a cada nus un valor global enter de retemporització, podem descriure una seqüència de retemporitzacions que ens permeti obtenir una xarxa semisistòlica equivalent.

És evident que si retemporitzem de cop, d'acord amb la funció *lag*, el nombre de registres en una aresta $a = (u, v)$ en el graf obtingut després de retemporitzar tots els nusos es podria calcular directament com:

$$\text{noupes}(a) = \text{pes}(a) + \text{lag}(v) - \text{lag}(u).$$

El resultat següent ens dóna les condicions que ha de complir una funció *lag* per tal que la transformació directa d'una xarxa semisistòlica S en una semisistòlica S' es pugui dur a terme mitjancant una seqüència vàlida de retemporitzacions locals.

Lema 3.1 (Retemporització). *Sigui S una xarxa semisistòlica amb graf G. Si existeix una funció* lag *definida sobre els enters tal que*

1. lag *(hoste de sortida)*$= 0$,

2. *per a tota aresta $a = (u, v)$ tenim* $\text{pes}(a) + \text{lag}\ (v) - \text{lag}\ (u) \geq 0$,

aleshores la xarxa S' amb graph G i pesos obtinguts per retemporització directa amb la funció lag *és una xarxa semisistòlica equivalent a S.*

Demostració. Sigui S' la xarxa amb el mateix graf G, obtinguda després de retemporitzar, d'acord amb la funció global *lag* que existeix per hipòtesi. Aleshores, per a tota aresta $a = (u, v)$ tenim que el seu pes final s'obté com a $\text{pes}'(a) = \text{pes}(a) + \text{lag}(v) - \text{lag}(u)$.

Observem que el pes de cada cicle a S' és idèntic al pes del cicle corresponent a S. Per hipòtesi, tots els cicles de S han de tenir pes positiu; aleshores tots els cicles de S' també tenen pes positiu. A més, tota aresta a S' té un pes més gran o igual que 0; per tant, S' és semisistòlica.

Per demostrar que S' és equivalent a S, hem de demostrar que S' es pot obtenir a partir de S utilitzant una seqüència de retemporitzacions. Sigui $L = \sum_{v \in V} |\text{lag}(v)|$. Farem una demostració per inducció sobre el valor de L.

Si $L = 0$ aleshores S és S'. Per a cada enter k, assumim que el lema és cert per a $L < k$, i volem demostrar que també és cert per a $L = k$.

Cas 1: Hi ha un nus u en què la funció lag és positiva. Llavors, si tenim almenys un nus v, pel qual tota aresta de sortida de v té pes més gran que 0, v es podria retemporitzar positivament. Després d'aquesta retemporització la nova $L' = L - 1 < k$ i, per la hipòtesi inductiva, el lema és cert.

Vegem ara que aquesta condició és certa. Suposem que tot nus té almenys una aresta de sortida amb pes 0; així u verificara la condició. Sigui $e = (u, v)$ l'aresta sortint de u amb pes 0. Com que s'ha de complir que $\text{pes}(u, v) + \text{lag}(v) - \text{lag}(u) \geq 0$ tenim que $\text{lag}(v) \geq \text{lag}(u) > 0$, d'on es pot deduir que $\text{lag}(v) > 0$. Per tant, si tenim una aresta amb pes 0, al cap de l'aresta la funció *lag* és positiva. Podem continuar aquesta construcció fins que arribem a generar un cicle o bé arribem a l'hoste de sortida. En el primer cas, podem trobar arestes amb pes 0 fins a formar un cicle a S; per tant, tenim un cicle amb pes total 0, la qual cosa és una contradicció. En el segon cas, el valor de la funció *lag* a l'hoste de sortida és major que zero, cosa que també és una contradicció. Per tant, hem contradit la hipòtesi, i existeix almenys un nus que té totes les arestes de sortida amb pes positiu i, per tant, es pot retemporitzar positivament.

Cas 2: El cas en què totes els nusos tinguin *lag* negatiu és semblant al cas anterior, però ara es necessitarà demostrar que hi ha un nus que té totes les arestes d'entrada amb pes positiu. Això es deixa com a exercici **3:6**. □

El teorema següent ens indica les condicions en les quals una xarxa semisistòlica es pot convertir en una xarxa sistòlica equivalent. La demostració del teorema és constructiva i dóna un procediment algorísmic per fer la conversió. La notació $G - 1$ representa el graf format a partir de G restant 1 als pesos de totes les arestes.

Teorema 3.1 (Teorema de conversió sistòlica). *Donada una xarxa semisistòlica S amb graf G, existeix una funció global* lag *tal que la xarxa resultant és sistòlica si i només si el graf $G - 1$ no té cicles amb pesos negatius.*

Demostració. Si $G - 1$ no té cicles amb pesos negatius, definim la funció $\text{lag}(v)$ com el pes del camí amb pes mínim de v a l'hoste de sortida. Perquè la funció *lag* estigui ben definida, necessitem que no hi hagi cicles negatius, altrament podem recórrer aquests cicles i reduir el pes tant com ho vulguem i no hi haurà camins mínims. Queda per demostrar que la funció *lag* compleix les condicions del lema de retemporització. Si h és l'hoste de sortida, tenim $\text{lag}\ (h) = 0$. Per a qualsevol aresta $a = (u, v)$, $\text{pes}'(a) = \text{pes}(a) + \text{lag}(v) - \text{lag}(u)$. Però a $G - 1$, el camí de u fins a l'hoste de sortida que passa per v no és més curt que el camí més curt de u fins a l'hoste. Per tant, $\text{lag}(u) \leq \text{pes}(a) + \text{lag}(v)$ i, substituint a dalt ens dóna, $\text{pes}'(a) = \text{pes}(a) + \text{lag}(v) - \text{lag}(u) \geq 0$. Aleshores podem aplicar el lema de retemporització.

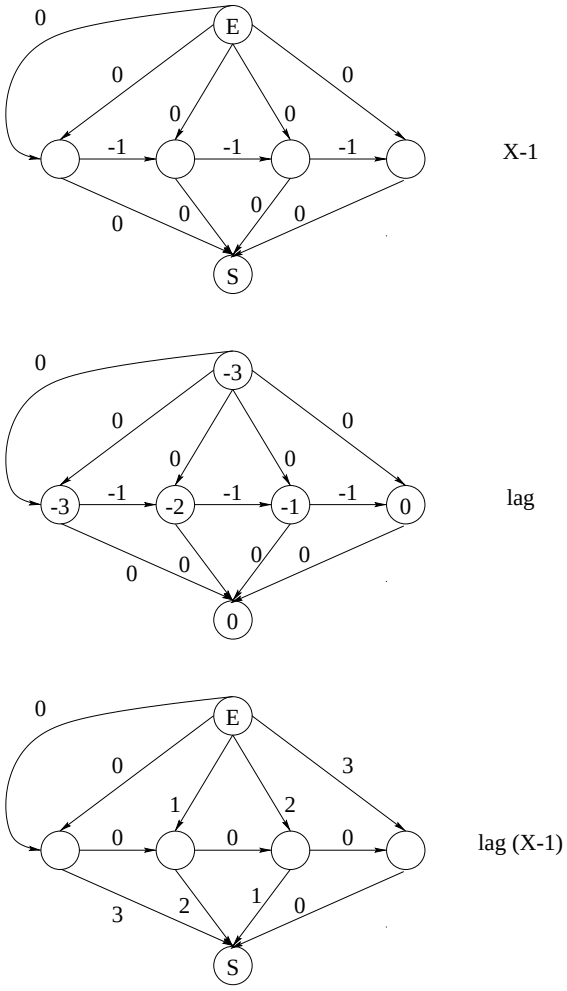


Figura 3.3: El graf $X-1$, la funció *lag* i el graf resultant per retemporització.

Per demostrar l'altra implicació, observem que una funció *lag* mai no canvia el pes total d'un cicle; aleshores, si hi ha un cicle negatiu a $G-1$, això implica que G té un cicle amb més arestes que pes i, per tant, no existeix una xarxa sistòlica equivalent a S. □

A la figura 3.3 tenim el digraf $X-1$ corresponent al digraf de la figura 3.1. La funció *lag* és l'obtinguda per l'aplicació del teorema precedent i el graf obtingut després de la retemporització.

Per acabar, cal sumar 1 a totes les arestes i recuperar el significat de les arestes que connecten els hostes. En aquest cas, la xarxa original no té cap sortida i l'entrada es fa a través de diferents arestes; això dóna la xarxa de la figura 3.4, on el pes a les arestes d'entrada significa que hem d'endarrerir l'arribada de les dades corresponents tantes unitats de temps com s'especifica. Observem que això ens porta a obtenir com a resultat l'algorisme sistòlic, que hem presentat al capítol anterior.

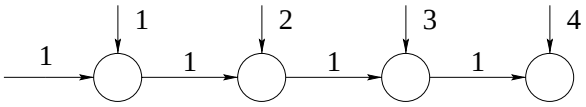


Figura 3.4: La xarxa semisistòlica corresponent al graf X.

3.2 Xarxes no sistolitzables

Si es dóna el cas que $G-1$ encara té cicles negatius, la xarxa no es pot sistolitzar. Malgrat això, es pot trobar un algorisme sistòlic que, encara que no sigui equivalent, resol el mateix problema. El procediment consisteix a trobar la mínima constant k tal que $kG-1$ no tingui cicles negatius (kG representa el graf al qual hem multiplicat tots els pesos per k). La xarxa representada per kG no és idèntica a la xarxa representada per G, però bàsicament les dues realitzen la mateixa computació, però una fa la computació amb un retard de k passos i en la xarxa sistòlica corresponent a kG podrem realitzar k computacions (del mateix tipus de problema) al mateix temps.

Exemple 3.1. *Considerem el problema de multiplicar una matriu A, amb banda k i dimensió $m \times n$, i un vector b de n elements, utilitzant un vector lineal.*

Recordem que una matriu de banda $k = 2r + 1$ és una matriu en què tots els elments fora de la diagonal principal i de les $2r$ diagonals secundàries són zero, és a dir,

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & 0 & 0 \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} & 0 \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & 0 & a_{76} & a_{77} \end{pmatrix}$$

En aquest cas, la fórmula habitual per calcular el producte es pot escriure com

$$c_i = \sum_{j=1}^{n} a_{ij} b_j = \sum_{i-r \le j \le i+r, j \ge 0} a_{ij} b_j .$$

Observem que, com a màxim, haurem de computar la suma de k elements; això vol dir que només cal utilitzar un vector lineal amb k processadors.

Si expandim aquesta fórmula, és fàcil veure que únicament els primers $r + 1$ elements de c necessiten el valor de b_1, i només els primers $r+2$ necessiten el valor de b_2. En general, com a màxim k elements necessiten el valor d'un determinat element de b. En particular,

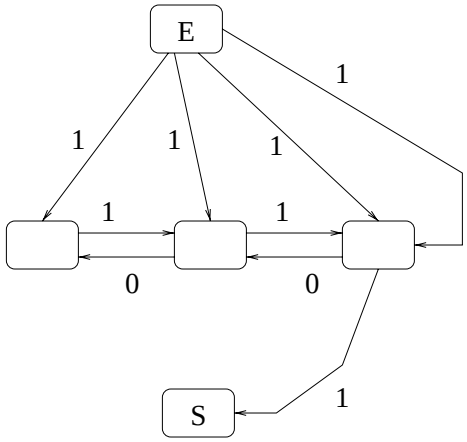


Figura 3.5: El graf G associat a la xarxa semisistòlica pel càlcul del producte d'una matriu de banda 3 per un vector.

tenim que

$$\begin{array}{lllll} c_1 = & a_{11}b_1 & +a_{12}b_2 & & \\ c_2 = & a_{21}b_1 & +a_{22}b_2 & +a_{23}b_3 & \\ c_3 = & & a_{32}b_2 & +a_{33}b_3 & +a_{34}b_4 \\ c_4 = & & & a_{43}b_3 & +a_{44}b_4 \quad +a_{45}b_5 \\ \cdots & & & \cdots & \cdots \end{array}$$

L'algorisme semisistòlic utilitza k processadors. Per la part de dalt entrarem la matriu de manera que cada processador rebi una diagonal i el processador central la diagonal principal. Per la dreta entrarem els elements de b amb cost zero a tots els processadors, i el vector c sortirà també per la dreta. Cada processador, excepte el primer (el que és més a l'esquerra), realitza en un pas les operacions següents:

- rep una dada per sobre (S), una per l'esquerra (E) i una per la dreta (D),
- calcula $B = S * D + E$,
- transmet B per la dreta.

El primer processador només rep S i D, calcula $B = S * D$ i transmet B per la dreta. El vector c anirà sortint per la dreta del processador més a la dreta.

A la figura 3.5 és dóna el graf G associat a la xarxa semisistòlica corresponent a l'algorisme precedent. Observem que el graf $G - 1$ té cicles negatius; això vol dir que per retemporització mai no trobarem una xarxa sistòlica equivalent. Però el graf $2G - 1$ no té cap cicle negatiu, i aquesta xarxa sí que es pot sistolitzar. A la figura 3.6 tenim el graf $2G - 1$ i els valors corresponents a la funció *lag* obtinguda a partir del teorema de sistolització. La xarxa sistòlica obtinguda es dóna a la figura 3.6. Cal remarcar que, per

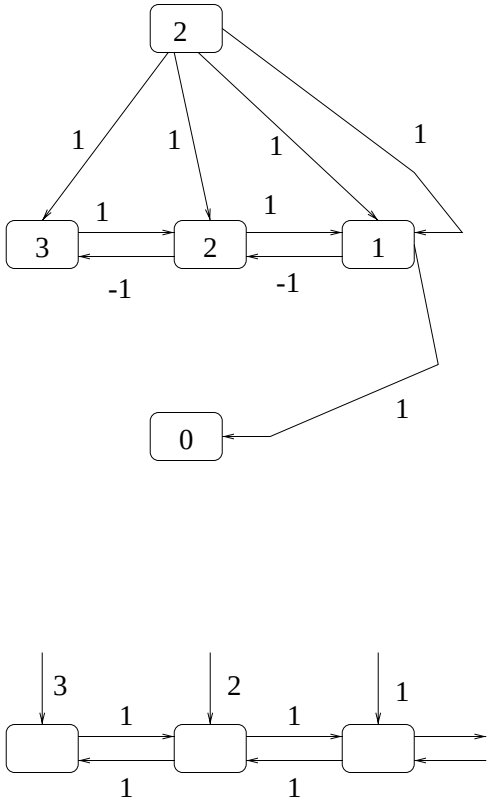


Figura 3.6: Graf $2G - 1$ i la seva funció *lag*.

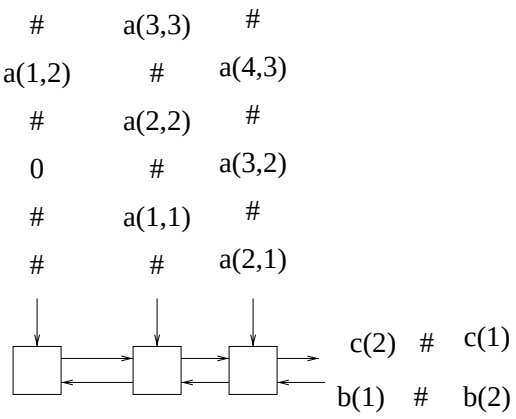


Figura 3.7: Una xarxa sistòlica per al càlcul del producte d'una matriu de banda 3 per un vector.

obtenir un algorisme sistòlic per al problema, hem multiplicat el graf per dos; això vol dir que la xarxa funciona més lentament, fa només una operació en dues unitats de temps. Per resoldre el problema, les dades han d'entrar a velocitat més lenta. A més, existeix una mena de temps mort a l'entrada de dades i de càlcul, que es pot aprofitar per resoldre altres instàncies del mateix problema.

3.3 Pipeline

Un avantatge que tenen els sistòlics és poder utilitzar l'efecte **pipeline**. És a dir, anar concatenant o solapant diferents instàncies del mateix problema per aconseguir que en cap moment no hi hagi processadors sense feina.

En l'example de la multiplicació de dues matrius $n \times n$ sobre una mesh, la utilització de pipeline ens permet computar el producte de k parells de matrius en $(k+2)n$ passos, mentre que si ho resolguéssim per separat necessitaríem $3kn$ passos.

En l'algorisme per a la multiplicació d'una matriu $n \times n$ amb banda k i un vector es necessiten $2n + k - 1$ passos, però utilitzant pipeline, multiplicar dues matrius es pot fer en $2n + k$ passos.

3.4 Disseny d'algorismes sistòlics per conversió

Resumim ara els passos que cal seguir per dissenyar un algorisme sistòlic per conversió, utilitzant els resultats presentats en aquest capítol.

1. Formalitzar el problema.

2. Dissenyar un algorisme per resoldre'l i una xarxa que ens en permeti la implementació. Cal que aquesta xarxa sigui semisistòlica, i hem de verificar que no hagi cicles amb

pes negatiu al digraf associat G. A més, hem de separar, si és posible, el flux de dades d'entrada i sortida.

3. Verificar si $G-1$ té cicles amb pes negatiu. Si trobem un cicle negatiu, cerquem el primer valor k tal que $kG-1$ no té cicles negatius.

4. Calcular la funció de retemporització i retemporitzar el graf kG. Hem de tenir cura a retemporitzar adequadament l'hoste d'entrada.

5. Maximitzar l'eficiència de la xarxa sistòlica resultant utilitzant pipeline, per resoldre k problemes.

3.5 Càlcul de la clausura transitiva

A continuació dissenyarem un algorisme més eficient per al càlcul de la clausura transitiva d'un graf. Per fer-ho primer, recordarem un algorisme seqüencial per resoldre aquest problema. Després dissenyarem una xarxa semisistòlica adient, veurem que es pot sistolitzar i trobarem un algorisme sistòlic. Com a xarxa base utilitzarem una mesh $n \times n$, on n és el nombre de nusos del graf.

3.5.1 Un algorisme seqüencial

Sigui $G = (V, A)$, amb $V = \{1, \cdots, n\}$, un graf no dirigit amb n vèrtexs i m arestes. L'algorisme seqüencial consta de n fases. Començant amb $G^0 = G$, a cada fase k, $1 \leq k \leq n$, es calcula el graf $G^k = (V, E^k)$ afegint arestes al graf G^{k-1}. A la fase 1, $E^1 = E^0$ i afegim l'aresta (i, j) a E^1 si i només si es pot connectar i amb j passant per 1 al graf G^0, és a dir, si $(i, 1) \in E$ i $(1, i) \in E$. A la fase 2, $E^2 = E^1$ i afegim (i, j) a E^2 si i només si $(i, 2) \in E^1$ i $(2, i) \in E^1$. A la fase k, $E^k = E^{k-1}$ i afegim l'aresta (i, j) a E^k si i només si $(i, k) \in E^{k-1}$ i $(k, i) \in E^{k-1}$. Representarem per A^i la matriu d'adjacència del graf G^i.

A cada fase k tenim la fórmula següent per calcular la matriu d'adjacència del graf G^k a partir de la matriu d'adjacència del graf G^{k-1}:

$$a_{ij}^k = a_{ij}^{k-1} \vee (a_{ik}^{k-1} \wedge a_{kj}^{k-1}). \tag{3.1}$$

L'algorisme corresponent CLAUSURA es dóna com a algorisme 3.

El lema següent que estableix la correcció de l'algorisme 3 es deixa com a exercici **3:7**.

Lema 3.2. *L'algorisme* CLAUSURA *calcula correctament la clausura transitiva d'un graf amb n vèrtex en $O(n^3)$ passos.*

3.5.2 Algorisme semisistòlic per la mesh

L'algorisme seqüencial sempre realitza la mateixa operació; així tindrem una xarxa on cada processador executarà la instrucció 7. Hem de decidir quines dades arriben a cada processador, fixar-ne les connexions i el flux de dades.

Algorisme 3 Càlcul de la clausura transitiva.

```
      CLAUSURA(a[1:n, 1:n])
1     per i = 1 fins n fer
2           per j = 1 fins  n fer
3                 b[i, j] := a[i, j]
            fper
      fper
4     per k = 1 fins  n fer
5           per i = 1 fins n fer
6                 per j = 1 fins  n fer
7                       b[i, j] := b[i, j] ∨ (b[i, k] ∧ b[k, j])
                  fper
            fper
      fper
```

La matriu d'adjacència entra per l'esquerra. A més, hem de tenir en compte que a cada pas del càlcul l'operació és

$$b[i,j] := b[i,j] \vee (b[i,k] \wedge b[k,j]).$$

El valor de k està fixat per a tots els valors i, j. Si, a més, fixem la fila i, cal transmetre el valor $b[i,k]$ cap a tots els processadors que continguin la i-èsima fila. Com que la matriu entra per l'esquerra, una fila de la matriu estarà emmagatzemada a una columna de la mesh. Per tant, necessitem una connexió simultània per columnes, com a la figura 3.8. Cal determinar quina dada es transmet a tota la columna, ja que no podem canviar la connexió depenent del pas de càlcul.

Analitzem la situació de la figura 3.9, si tenim la fila k d'A^{k-1} a la columna k de la mesh, i per l'esquerra arriba la fila i d'A^{k-1}, transmetent amb cost zero, l'element que arriba per la diagonal a tots els processadors de la columna. Els processadors calculen la fila i d'A^k, que es pot transmetre per l'esquerra.

L'algorisme es divideix en dues fases, encara que les dues acabaran solapant-se, ja que la segona fase no espera que la primera finalitzi.

A la primera fase, per a tot i, $1 \leq i \leq n$, la fila i-èsima de la matriu A^0 entra per l'esquerra i avança fins a trobar la columna i-èsima de la mesh, a on es queda. Observem que a mesura que la fase 1 s'executa, a la columna 1 de la mesh tindrem la fila 1 de A^0; a la segona columna en la fila 2 d'A^1; per tant, a mesura que la fila travessa la mesh es va reconvertint primer en fila i d'A^1, en fila i d'A^2 fins a ser la fila i d'A^{i-1}, quan es queda a la columna i-èsima de la mesh. En acabar la fase 1 es compleix que, per a tot $1 \leq i \leq n$, la columna i-èsima de la mesh conté la fila i-èsima de G^{i-1}.

La segona fase comença quan la darrera fila d'A^0 ha entrat a la mesh; en aquest moment, la fila 1 d'A^0 emmagatzemada a la columna 1 de la mesh comença a viatjar cap a la dreta,

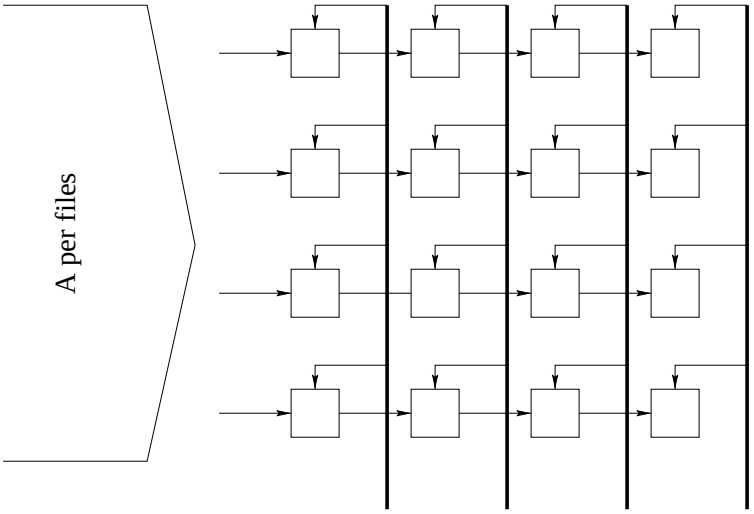


Figura 3.8: Connexions inicials de la xarxa semisistòlica per al càlcul de la clausura transitiva.

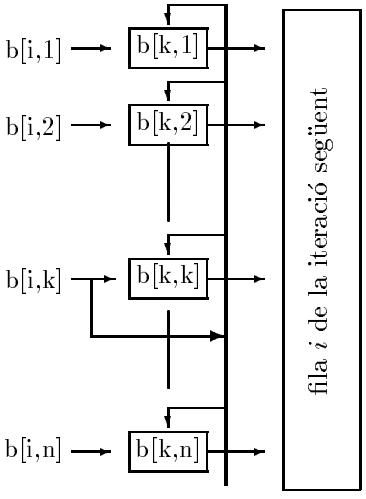
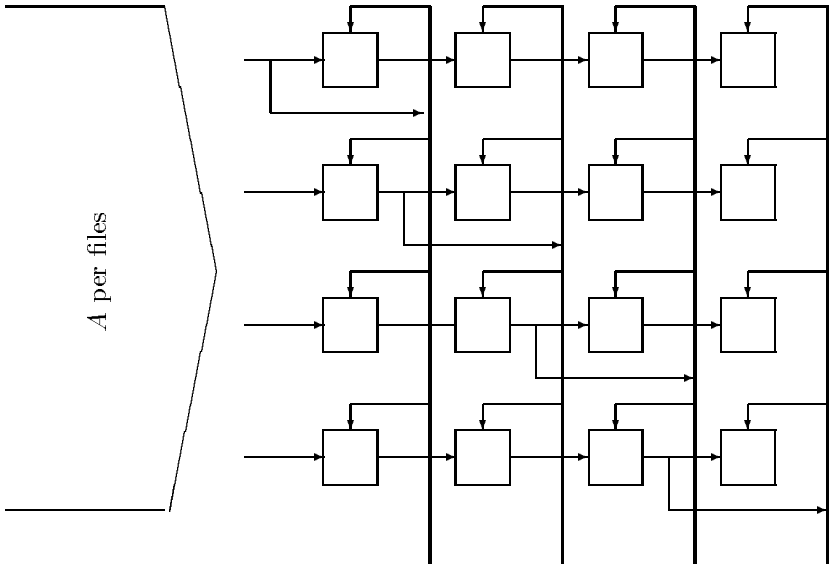
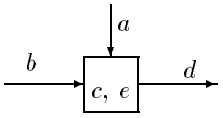


Figura 3.9: Anàlisi del comportament de l'operació bàsica.





inicialment c=&
$e = 0$

si c=&, $c = b$ i no es trasmet res per d

si no

si $e = 1$, $d = \#$
i el processador es desactiva

si $e = 0$
si $b = \#$, $d = c$ i $e = 1$
sino $d = c + (a.b)$

Figura 3.10: Xarxa semisistòlica per al càlcul de la clausura transitiva.

fins a sortir per la dreta de la mesh, convertida en fila 1 d'A^n. Totes les files fan el mateix, després que la fila $i-1$ d'A^{i-2} passi per sobre de la fila i d'A^{i-1}, la fila i d'A^{i-1} es transmet cap a la dreta fins a sortir per la dreta de la mesh convertida en fila i d'A^n.

A l'algorisme semisistòlic de la figura 3.10 hem suposat que la mesh està inicialitzada amb #, que controla el moment que la fila arriba a la columna, on es queda durant la primera fase. El processador (i, j) que rep les dades a i b i té emmagatzemat c calcula el valor d i actualitza c en les condicions especificades a la figura 3.10.

A la figura 3.11 es dóna un exemple d'aplicació de l'algorisme. Observem que en aquest algorisme les dues fases s'implementen simultàniament i, a més, hem assenyalat el fet que no es transmet cap missatge per una línia amb el símbol #.

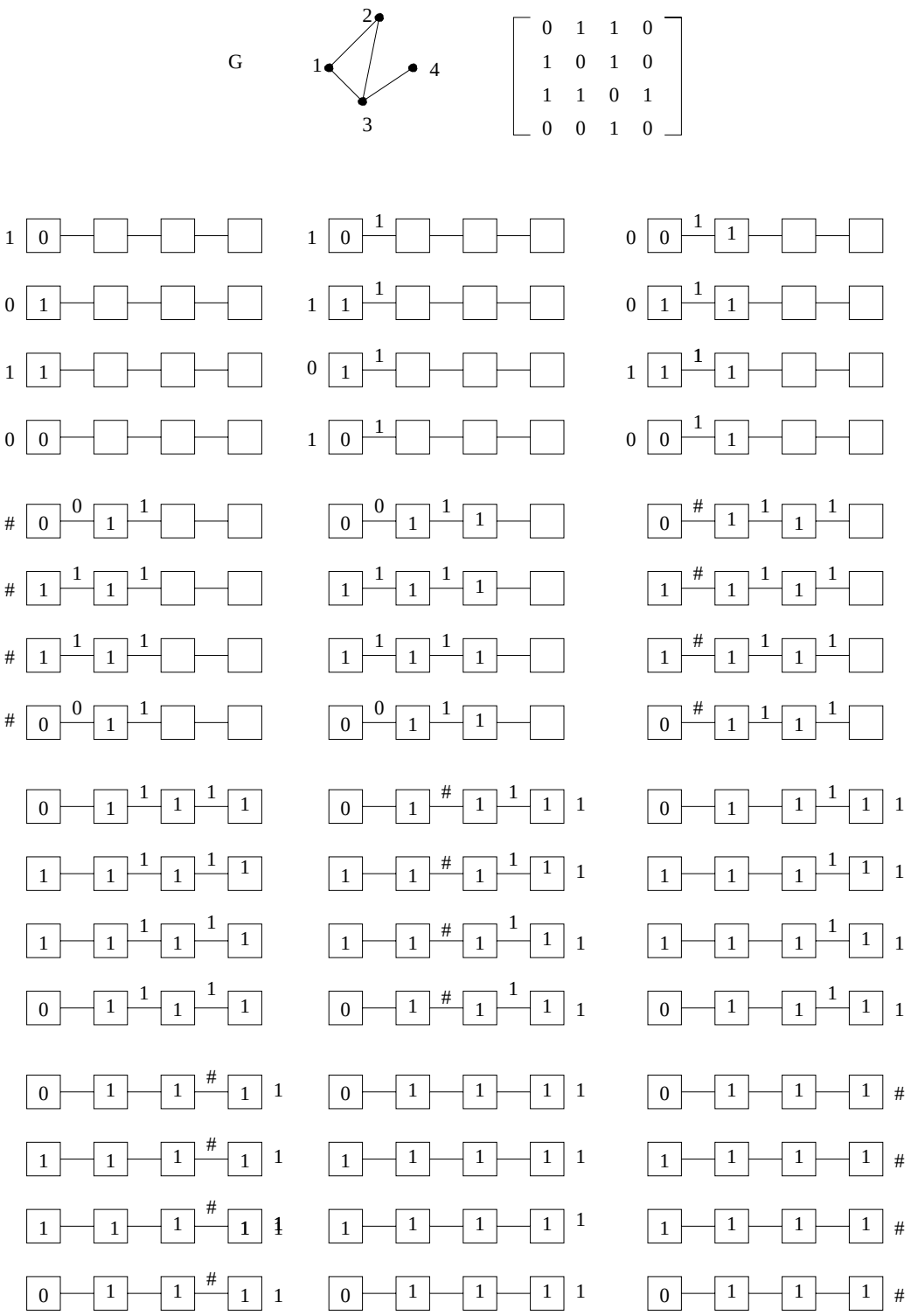


Figura 3.11: Exemple d'aplicació de l'algorisme semisistòlic per la clausura transitiva.

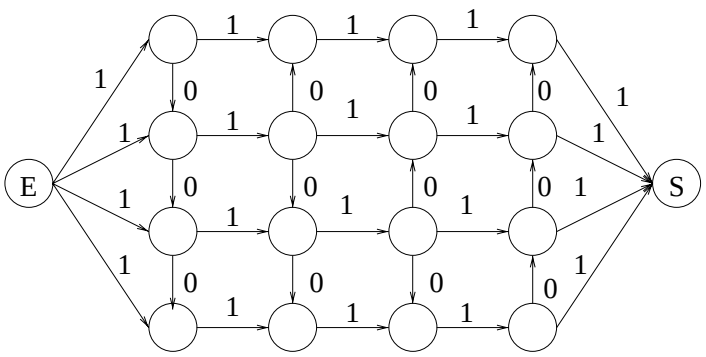


Figura 3.12: El graf $X(G)$ associat a la xarxa semisistòlica per calcular la clausura transitiva.

3.5.3 Algorisme sistòlic

A la figura 3.12 tenim el graf $X(G)$ associat a la xarxa semisistòlica representada a la figura 3.10. El graf $X(G) - 1$ es dóna a la figura 3.13. Com podem observar, $X(G) - 1$ no té cicles i, per tant, no té cicles negatius. Podem aplicar el mètode descrit al teorema de conversió sistòlica per trobar la funció *lag*, retemporitzar la xarxa i convertir-la en una de sistòlica. A la figura 3.13, dintre dels cercles, tenim el valor de la funció *lag* calculada. Retemporitzant d'acord amb la funció *lag*, obtenim el segon graf de la figura 3.13, que és el graf $X(G') - 1$ corresponent a una xarxa sistòlica.

Finalment, a la figura 3.14, tenim la disposició del flux de dades i els retards en la transmissió de dades. Observem que el temps total per computar la clausura transitiva d'un graf amb n nusos utilitzant una mesh com la descrita és de $3n - 1$.

3.6 Referències bibliogràfiques

El disseny d'algorismes sistòlics utilitzant semisistolicitat és explicat a les seccions 1.4–1.7 del llibre de Leighton [Lei93].

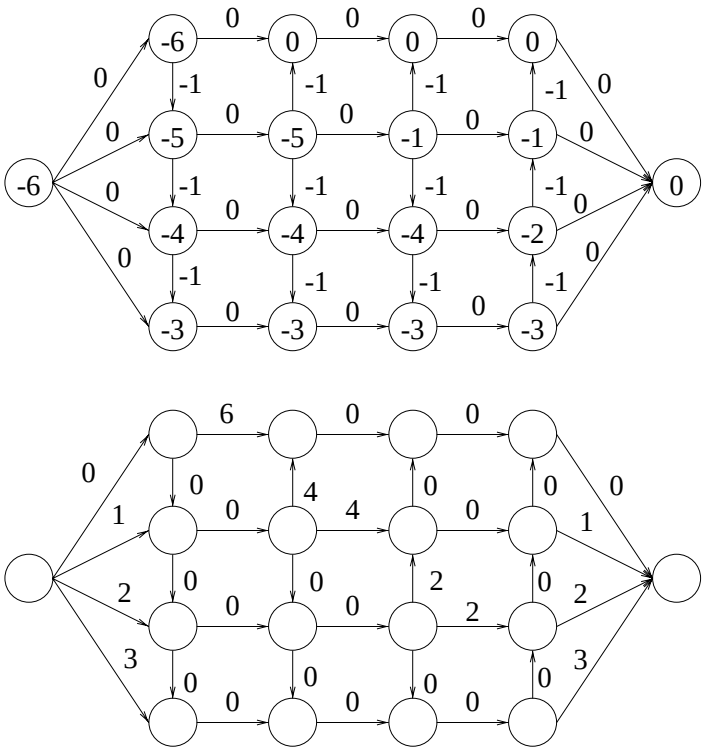


Figura 3.13: $X(G) - 1$ i els valors de la funció *lag*.

Figura 3.14: Graf associat a la xarxa sistòlica per al càlcul de la clausura transitiva.

Exercicis

3:1 Dissenyeu un algorisme sistòlic per a la mesh per resoldre un sistema triangular inferior, és a dir, un sistema $Ax = b$, on A és una matriu que té zeros per sobre de la diagonal.

3:2 Dissenyeu un algorisme sistòlic per al vector lineal per avaluar un polinomi.

3:3 Dissenyeu un algorisme sistòlic per a la mesh per calcular el producte de dos polinomis.

3:4 Dissenyeu un algorisme sistòlic per a la mesh, per trobar els components connexos d'un graf no dirigit.

3:5 Dissenyeu un algorisme sistòlic per a la mesh que resolgui el problema de trobar el camí amb pes mínim en un digraf amb pesos.

3:6 Completeu la demostració del lema 3.1.

3:7 Demostreu el lema 3.2.

3:8 Modifiqueu l'algorisme de multiplicació de matrius amb una mesh, de manera que admeti pipeline. (És a dir, quan el processador $(1,1)$ a la mesh estigui buit, es pot comencar a rebre més informació d'una altra entrada.) Quina seria la complexitat de multiplicar m matrius $n \times n$?

Capítol 4

Xarxes en arbre

En aquest capítol introduirem un nova topologia per a xarxes d'interconnexió, la topologia d'arbre. De fet, gairebé totes les xarxes amb aquesta topologia utilitzen com a base un arbre binari complet. Aquesta topologia té un diàmetre petit però la seva amplada de bisecció també és petita, això fa que sigui una xarxa adient per resoldre problemes sense gaire necessitat de comunicació.

Com que un simple arbre binari pot no ser prou versàtil, considerarem d'altres xarxes que tenen com a topologia base la d'arbre binari, en particular estudiarem l'anomenada "*mesh d'arbres*", que és una xarxa amb diàmetre petit i amplada de bisecció gran.

4.1 Arbre binari

En aquesta topologia, la xarxa forma un arbre binari complet amb $n = 2^d$ fulles. L'arbre té d nivells, a cada nivell i té 2^i processadors. En total una xarxa d'arbre binari amb alçada d té $2^{d+1} - 1$ processadors. L'arbre binari és una topologia amb diàmetre i amplada de bisecció petits.

Lema 4.1. *Una xarxa amb topologia d'arbre binari complet amb n fulles té diàmetre* $2 \log n$ *i amplada de bisecció 1.*

La demostració es deixa com a exercici **4:1**. Observem que les propietats de la topologia fan que sigui una xarxa ràpida, que es comporte bé en problemes que no requereixin gaire comunicació.

Vegem alguns exemples d'algorismes sistòlics per a la topologia d'arbre binari. En tots ells assumirem que el nombre de dades d'entrada és una potència de 2.

Exemple 4.1 (Màxim). *Com trobar el màxim de n nombres amb una xarxa en forma d'arbre binari amb n fulles i alçada* $\log n$.

Les dades entren per les fulles, com a la figura 4.1. Cada nus a la xarxa és un processador que compara els dos nombres que li entren, emmagatzema a la seva memòria local el més

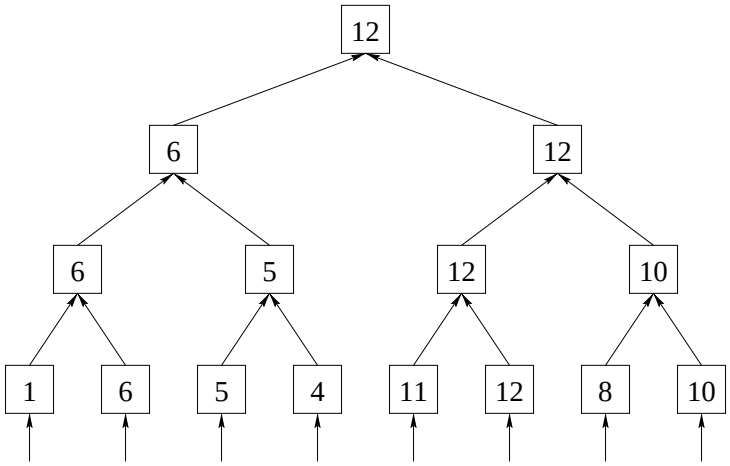


Figura 4.1: Xarxa en arbre per trobar el màxim de 8 elements.

petit i envia cap a dalt el més gran. El processador a l'arrel traurà el màxim. El temps és $\Theta(\log n)$ i el nombre de processadors $2n-1$.

De manera similar, es pot computar la suma o el producte de n nombres amb una xarxa en forma d'arbre binari, d'alçada $\log n$.

Exemple 4.2 (Sumes prefixades). *Donada una seqüència de n elements $x_1, \ldots, x_n$, volem calcular les sumes $y_i = x_1 + \cdots + x_i$ per a tot $1 \leq i \leq n$.*

Utilitzem un arbre binari complet amb n fulles, on $n = 2^k$ per a algun k. L'algorisme té dues fases, que s'implementen a mesura que es pot. En la primera, les dades traverssen l'arbre des de les fulles a l'arrel, i en la segona passen en l'ordre contrari. A les figures 4.2 i 4.3 es dóna un exemple del càlcul de les sumes prefixades per a un conjunt de 8 elements.

Les dades inicials entren a l'arbre a través de les fulles, és a dir, x_i entra a la xarxa per la fulla i-èsima. Al llarg del còmput els processadors reben dues dades dels fills (si no són fulles) o una del pare. Al primer cas les sumen emmagatzemen el resultat a la memòria local i després transmeten el resultat al pare. La dada que reben del fill esquerra la transmeten al fill dret. Quan reben una dada del pare, els processadors interns la passen als dos fills, mentre que els processadors fulles acumulen el valor al que ja tenen emmagatzemat. El resultat de les sumes prefixades queda emmagatzemat a les fulles. Aquest algorisme necessita dues passades per l'arbre i el seu cost temporal és de $2\log n$ passos.

4.2 Taula d'arbres

Estudiem ara una nova topologia, que és una barreja d'una mesh i d'arbres binaris, que s'anomena **taula d'arbres** (mesh of trees), que té un diàmetre petit i una amplada de bisecció gran. Aquestes dues propietats converteixen les taules d'arbres en unes xarxes adients per dissenyar algorismes ràpids.

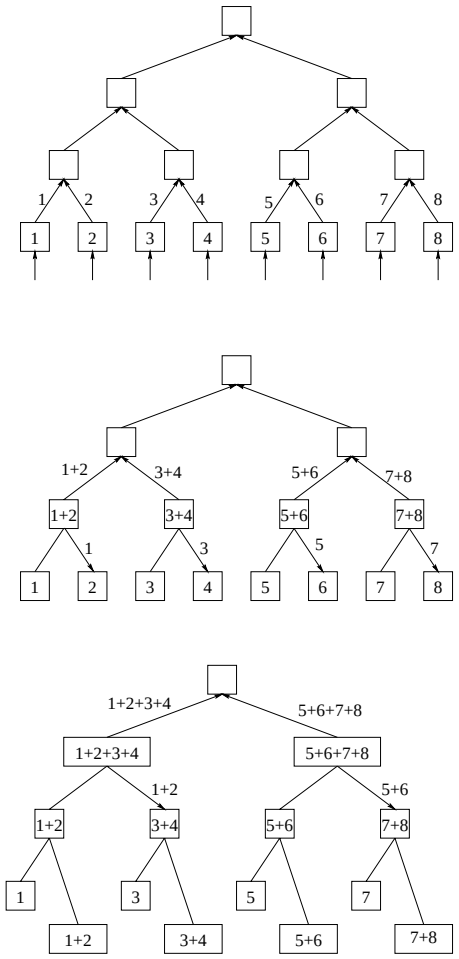


Figura 4.2: Càlcul de les sumes prefixades de 8 elements (I).

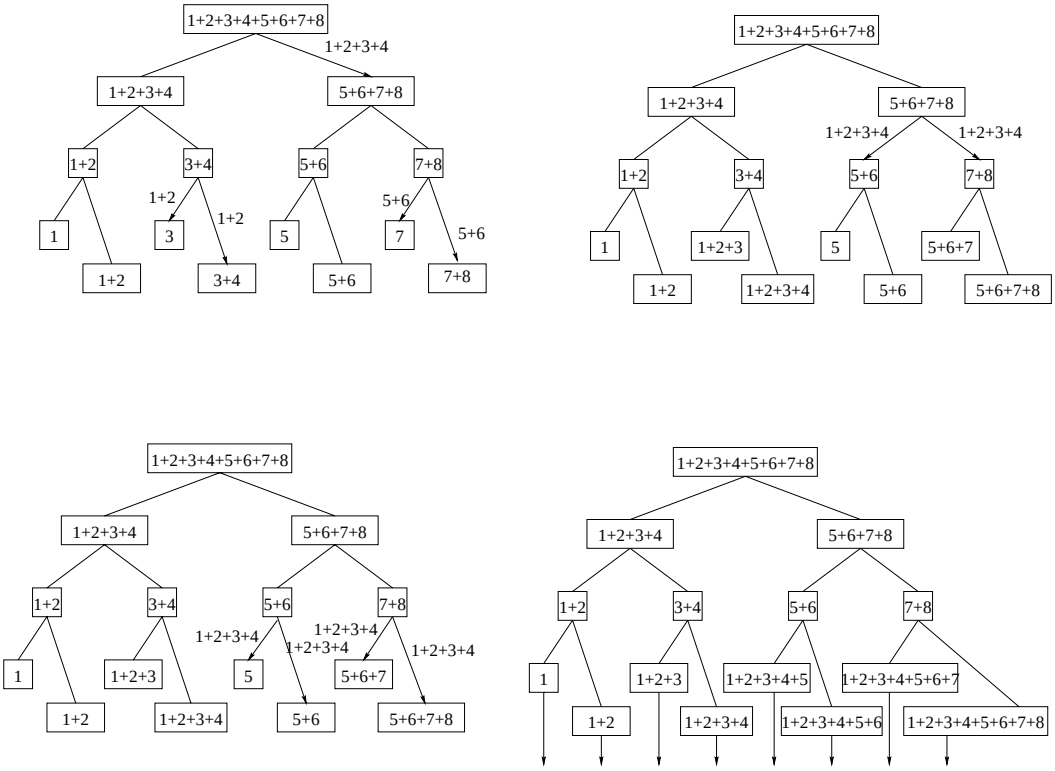


Figura 4.3: Càlcul de les sumes prefixades de 8 elements (II).

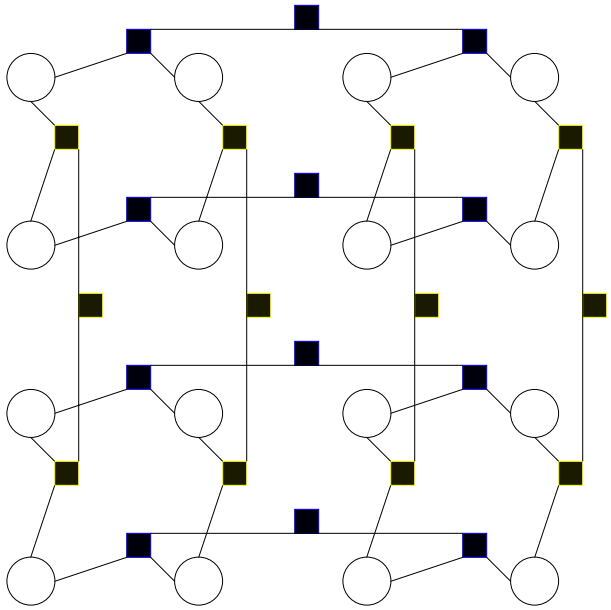


Figura 4.4: Taula d'arbres 4×4.

Una *taula d'arbres* amb dimensió $n \times n$ es construeix a partir d'un engraellat de $n \times n$ processadors etiquetats $\{(i,j) \mid 1 \leq i,j \leq n\}$, que no estan connectats entre ells. Per a cada i, $1 \leq i \leq n$, afegim nusos i arestes fins a formar un arbre binari complet, les fulles del qual són els nusos $\{(i,j) \mid 1 \leq j \leq n\}$. Aquests arbres s'anomenen **arbres fila**. De la mateixa manera, per a cada j, $1 \leq j \leq n$, afegim nusos i arestes fins a formar un arbre binari complet, les fulles del qual són els nusos $\{(i,j) \mid 1 \leq i \leq n\}$. Aquests arbres s'anomenen **arbres columna**. A la figura 4.4 els processadors de la base es representen amb cercles i els dels arbres amb quadrats. El nombre total de nusos és $3n^2 - 2n$; d'aquests processadors els n^2 processadors corresponents a les fulles són els de l'engraellat original.

El lema següent es deixa com a exercici **8:2**,

Lema 4.2. *Una taula d'arbres de dimensió $n \times n$ té un diàmetre de $4 \log n$ i una amplada de bisecció de n.*

Exemple 4.3 (Classificació). *Classifiquem ara els n elements del conjunt $\{x_0, x_1, \ldots, x_{n-1}\}$.*

En el pas 1, entrem cada element x_i a l'arrel del i-èsim arbre columna. Per a tot i, en $\log n$ passos, el valor x_i a l'arrel es transmet fins a les fulles del i-èsim arbre fila. En $2 \log n$ passos, el valor x_i es transmet des de les fulles del i-èsim arbre fila fins a les fulles del i-èsim arbre columna. Per tant, en el pas $3 \log n + 1$, el processador corresponent a la fulla (i,j) ha rebut els valors x_i i x_j, i els pot comparar. Si $x_j > x_i$ o $x_j = x_i$, amb $j > i$, aleshores (i,j) emmagatzema 1, altrament (i,j) emmagatzema 0. Al pas següen, se sumen en $\log n$ passos, els 0 i 1 emmagatzemats a les fulles de cada arbre columna. El resultat k_i de la suma a l'arrel del i-èsim arbre columna serà el rang de l'element x_i emmagatzemat

inicialment a l'arrel. Finalment cal enviar l'element x_i a l'arrel del k_i-èsim arbre fila, el qual trigarà com a molt $3 \log n$ passos. Així, la classificació de n elements es pot realitzar en $7 \log n + 1$ passos sobre una mesh d'arbres amb dimensió $n \times n$.

Exemple 4.4 (Producte matriu per vector). *Computem el producte d'una matriu* $A = (a_{ij})$ *de dimensió* $n \times n$ *per un vector* n *dimensional* $X = (x_i)$.

Sigui $Y = AX$ el vector resultant, $Y = (y_i)$. Per computar Y amb una taula d'arbres $n \times n$, comencem per entrar cada x_i a l'arrel del i-èsim arbre columna ($1 \leq i \leq n$). Els x_i es passen cap avall pels arbres columna, de manera que cada fulla del i-èsim arbre columna rep x_i en el pas n. En el pas $\log n$ de la computació, comencem a entrar a_{ij} a la fulla (i, j), de manera que aquesta fulla (i, j) computa el producte $a_{ij}x_j$. Aquests valors se sumen utilitzant els arbres fila, la qual cosa en $2 \log n$ passos l'arrel del i-èsim arbre fila conté $y_i = \sum_{j=1}^{n} a_{ij}x_j$.

És fàcil adonar-se que la topologia descrita no és la més adient de am vista a la multiplicació de matrius, sobretot per la seva amplada de bisecció, ja que tindrem n^2 paquets a la xarxa. Per realitzar la multiplicació de matrius de manera eficient introduirem a la secció 4.4 una generalització de la topologia prèvia. Abans, vegem l'encaminament a la taula d'arbres.

4.3 Encaminament en taules d'arbres

El fet que una taula d'arbres amb dimensió $n \times n$ tingui una amplada de bisecció de n fa que per encaminar correctament n^2 paquets, en què inicialment $n^2/2$ estan a cada costat de la bisecció, en el cas pitjor tots han d'anar a l'altre costat, cosa que requerirà com a mínim $n/2$ passos. Per tant qualsevol algorisme d'encaminament necessitara com a mínim $\Omega(n)$ passos. Però el fet que aquesta topologia tingui diàmetre petit respecte a la *mesh*, donarà un avantatge en resoldre el problema d'encaminar n paquets.

Considerem que tenim $m < n$ paquets emmagatzemats a les arrels dels arbres files i volem fer-los arribar a les seves destinacions, que són a les arrels dels arbres columnes. Sigui p_i la destinació del paquet enmagatzemat a l'arrel del i-èsim arbre fila. Per encaminar aquest paquet a l'arrel del p_i-èsim arbre columna, primer encaminem en $\log n$ passos el paquet cap a la fulla p_i del i-èsim arbre fila. Després encaminem el paquet directament cap a l'arrel del seu arbre columna, i arribarà en $\log n$ passos. Si les destinacions dels paquets són diferents, els camins seguits pels paquets mai no es creuen. Per tant, l'algorisme té una complexitat de $2 \log n$ passos.

Si més d'un paquet vol anar a la mateixa arrel, aleshores l'arbre corresponent actua com a cua de prioritat fent que els paquets arribin a l'arrel, un darrere de l'altre.

4.4 Taula d'arbres tridimensional

Estenem ara la topologia de taula d'arbres bidimensional al cas tridimensional. Una taula d'arbres amb dimensió $n \times n \times n$ es construeix a partir d'un cub $n \times n \times n$, amb els nusos

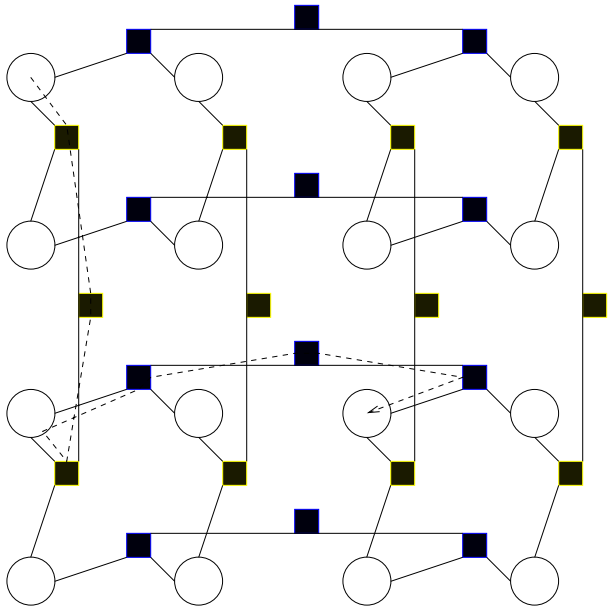


Figura 4.5: Encaminament en una taula d'arbres.

etiquetats $\{(i,j,k)|1 \leq i,j,k \leq n\}$. Per a cada $j,k : 1 \leq j,k \leq n$ afegim nusos i arestes fins a formar un arbre binari complet les fulles del qual són els nusos $\{(i,j,k)|1 \leq j \leq n\}$. Aquest arbre s'anomena l'arbre (j,k) amb dimensió 1. De manera similar, per cada $i,k : 1 \leq i,k \leq n$ afegim nusos i arestes fins a formar un arbre binari complet les fulles del qual són els nusos $\{(i,j,k)|1 \leq j \leq n\}$. Aquest arbre s'anomena l'arbre (i,k) amb dimensió 2. De la mateixa manera, es pot definir l'arbre (i,j) amb dimensió 3. Cadascun dels nusos del cub de partida és una fulla d'un dels arbres a cada dimensió.

Una taula d'arbres $n \times n \times n$ té $n^3+3n^2(n-1) = 4n^3-3n^2$ nusos i $3n^2(2n-2) = 6n^3-6n^2$ arestes. Cada nus té tres veïns excepte les $3n^2$ arrels, que en tenen dos. És fàcil demostrar el resultat següent:

Lema 4.3. *Una taula d'arbres $n \times n \times n$ té diàmetre $6 \log n$ i amplada de bisecció n^2.*

L'estructura de la taula d'arbres $n \times n \times n$ no és més que n taules d'arbres $n \times n$ amb les fulles connectades per n^2 arbres binaris complets.

Exemple 4.5 (Multiplicació de matrius). *Vegem ara com multiplicar dues matrius amb dimensions $n \times n$, $A = (a_{ij})$ i $B = (b_{ij})$ sobre una taula d'arbres $n \times n \times n$.*

Al començament, per a cada $i,j; 1 \leq i,j \leq n$, el valor a_{ij} entra a l'arrel de l'arbre (i,j) amb dimensió 3. Al mateix temps, per a cada $j,k; 1 \leq j,k \leq n$, b_{jk} entra a l'arrel de l'arbre (j,k) amb dimensió 1. En els propers $\log n$ passos, els valors d'A i B es distribueixen pels arbres. En el pas $\log n + 1$, per a cada valor de $i,j,k; 1 \leq i,j,k \leq n$, la fulla (i,j,k) rep al mateix temps $a_{i,j}$ i $b_{j,k}$, i els multiplica en un pas. A continuació, cada fulla passa el resultat

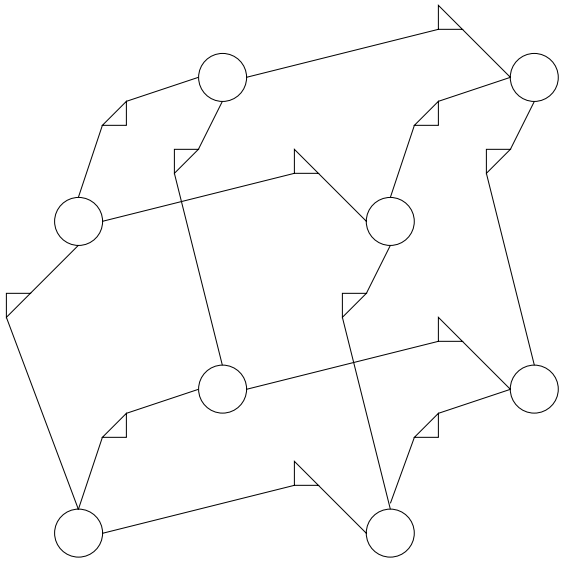


Figura 4.6: Una taula d'arbres $2 \times 2 \times 2$.

de la multiplicació al seu pare en l'arbre de dimensió 2. En $\log n$ passos més, cada arbre amb dimensió 2 suma els productes obtinguts a les seves fulles, i el resultat $c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$ queda a l'arrel del corresponent arbre (i, k) amb dimensió 2. Però aquesta quantitat és l'entrada c_{ik} a la matriu $C = AB$.

Aquest algorisme triga $2logn + 1$ passos per multiplicar dues matrius de $n \times n$, però evidentment el nombre de processadors és gran, cosa que fa que l'algorisme no sigui massa eficient.

4.5 Referències bibliogràfiques

Les taules d'arbres són una xarxa molt adient per a la resolució de problemes de grafs. Un cop més, la referència global és el capítol 2 del [CLR89].

Exercicis

4:1 Demostreu el lema 4.1.

4:2 Donada una llista de n enters positius (no ordenada), dissenyeu un algorisme per la topologia d'arbre binari per trobar, al mateix temps, el màxim i el mínim.

4:3 Donat un polinomi p de grau n i un valor a, dissenyeu un algorisme per la topologia d'arbre binari per calcular $p(a)$. Cal que l'algorisme només computi directament els valors a^k per valors de k que siguin potències de 2.

4:4 Dissenyeu un algorisme per a la topologia d'arbre binari per resoldre recurrències lineals, és a dir, equacions de la forma

$$z_i = a_i\, z_{i-1} + b_i\, z_{i-2}$$

per $2 \leq i \leq N$, donats $a_2, \ldots, a_N$, $b_2, \ldots, b_N$, z_0 i z_1.

4:5 Demostreu que una taula d'arbres de dimensió $n \times n$ té diàmetre $4 \log n$ i amplada de bisecció n.

4:6 Demostreu el ema 4.2.

4:7 Donats dos polinomis de grau n, volem calcular el seu producte en temps $O(\log n)$; per a això disposem d'una taula d'arbres de dimensió $n + 1 \times 2n + 1$, amb arbres diagonals addicionals.

Capítol 5

L'hipercub i xarxes similars

En aquest capítol introduirem un dels models més potents de SIMD amb xarxa d'interconnexió: l'hipercub. També esmentarem altres variacions de l'hipercub que, sense perdre potència, tenen alguns avantatges sobre el model bàsic de l'hipercub.

L'hipercub és una xarxa molt versàtil, per la facilitat i el poc cost amb què pot simular de manera eficient una computació sobre un vector, un arbre binari, una mesh o una taula d'arbres. És a dir, qualsevol algorisme sobre una de les xarxes descrites pot ser implementat sobre un hipercub amb un retard constant en el nombre de passos. Com a conseqüència, un algorisme per a les topologies vistes pot ser implementat directament sobre l'hipercub sense pèrdua d'eficiència. Aquest fet fa que els models del tipus hipercub siguin "populars" en el disseny de màquines paral·lleles reals.

5.1 L'hipercub

Donats $n = 2^q$ processadors, $p_0, p_1, \cdots, p_{n-1}$, una xarxa amb topologia d'un **hipercub** amb dimensió q s'obté connectant cada processador amb q veïns de la manera següent: el processador p_i es connecta amb tots els processadors p_j tals que la representació binària de j difereix només en 1 bit de la representació binària de i. Una altra manera equivalent de definir-ho és dir que dos processadors estan connectats si la seva *distància de Hamming* és 1.

Per tant, com a graf, un hipercub q-dimensional té $n = 2^q$ nusos (cada nus serà un processador) i $q\,2^{q-1}$ arestes. A més, cada nus té $q = \log n$ veïns, un per a cada posició de la representació binària del nus. Una aresta d'un hipercub amb n nusos es diu que té *dimensió* k si connecta dos nusos que difereixen exactament en la posició k-èsima. Donat un nus u, u^k representa el veí que té la posició k complementada i la resta són iguals. De la mateixa manera $u^{k_1,k_2,\ldots,k_r}$ és el nus que difereix de u en les posicions $k_1, k_2, \ldots, k_r$. Existeix un camí de u fins a $u^{k_1,k_2,\ldots,k_r}$ seguint successivament les arestes de dimensió k_1, k_2, etc. (vegeu la figura 5.1).

Si un hipercub amb n nusos eliminem totes les arestes de dimensió k per a algun $k \leq q$, ens queden dos hipercubs, cadascun amb $n/2$ nodes. Aquesta observació ens permet

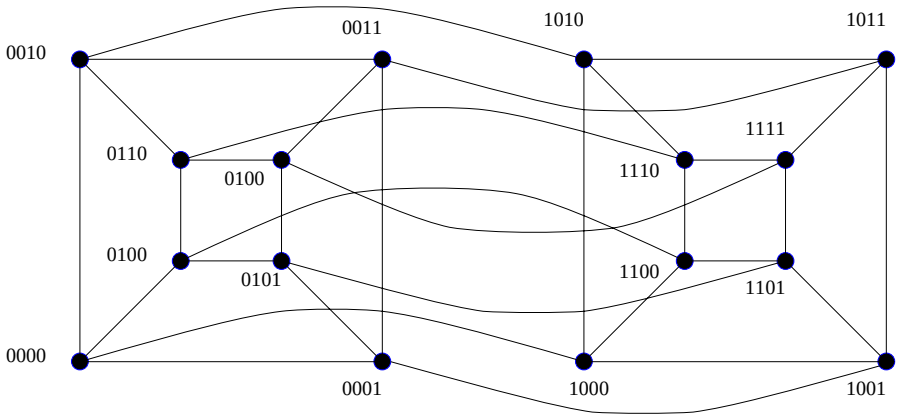


Figura 5.1: Un hipercub 4-dimensional amb els nusos etiquetats en binari.

redefinir un hipercub de manera recursiva. Un hipercub amb dimensió 0 és exactament un nus, etiquetat amb 0. Un hipercub amb dimensió $q > 0$ s'obté a partir de dos hipercubs amb dimensió $q-1$, afegint-hi connexions entre nodes amb el mateix identificador i modificant-ne els identificadors finals, cal afegir un 0 a la dreta als identificadors d'un hipercub i un 1 a la dreta dels identificadors de l'altre.

El lema següent presenta dues propietats que fan de l'hipercub una xarxa amb bones propietats de connexió.

Lema 5.1. *Un hipercub amb n nusos té diàmetre $\log n$ i amplada de bisecció $n/2$.*

La demostració es deixa com a exercici **5:1**. El lema ens indica que, com a xarxa, l'hipercub té diàmetre petit i bisecció gran, que com ja hem esmentat són dues de les característiques que fan que una xarxa sigui eficient.

Exemple 5.1 (Suma). *Volem sumar $n = 2^q$ elements en un hipercub q-dimensional. Inicialment cada valor x_i és emmagatzemat a la memòria local del procesador p_i. El resultat final el volem a la memòria local del processador p_0.*

L'algorisme té q fases. A cada fase i, $0 \leq i \leq q-1$, tot parell de processadors actius connectats per arestes de dimensió $q-i$ emmagatzemen la suma dels valors emmagatzemats en ells. Al començament, tots els processadors són actius, a la iteració i-èsima els processadors actius són els que formen un hipercub de dimensió $q-i$, amb etiquetes que tenen i zeros en els bits més significatius.

Cada fase triga un temps constant; així, doncs, el temps total és $\log n$, amb un cost $O(n \log n)$.

Exemple 5.2 (Producte de matrius). *Volem calcular el producte C de dues matrius, A i B, de dimensions $n \times n$.*

Assumirem que $n = 2^q$ i que fem el càlcul en un hipercub amb n^3 processadors. Etiquetem els processadors per tripletes, de manera que el processador $P_{l,i,j}$ representa el

processador P_r amb $r = l\,n^2 + i\,n + j$. Observem que si expandim en binari el valor de r, els q bits més significatius corresponen al valor l; els següents q bits, al valor i, i la resta, al valor j. A més, si fixem els índexs l, i tenim un hipercub complet amb n processadors i si només fixem un índex tindrem un altre subhipercub amb n^2 processadors.

La matriu A estarà continguda a l'hipercub amb $j = 0$; així l'element $A(l,i)$ és a la memòria del processador $P_{l,i,0}$. La matriu B és a l'hipercub amb $i = 0$, ja que l'element $B(l,j)$ és a la memòria del processador $P_{l,0,j}$. L'algorisme té tres fases:

1. Els valors d'entrada es transmeten de manera que el processador $P_{l,i,j}$ contingui els valors de $A(i,l)$ i $B(l,j)$,
2. el processador $P_{l,i,j}$ calcula el producte dels dos valors que té emmagatzemats,
3. cadascun dels hipercubs obtinguts fixant els valors de i, j calculen la suma dels valors al processador $P_{0,i,j}$.

A la primera fase cal un algorisme per transmetre dades d'un processador a tots els processadors d'un subhipercub, cosa que es pot fer en temps $\log n$ i es deixa com a exercici **5:2**. La segona fase es pot implementar en temps constant. Finalment, la tercera fase es pot implementar utilitzant l'algorisme de l'exemple anterior en paral·lel a tots els n subhipercubs. Per tant, hem trobat un algorisme que calcula el producte de dues matrius en temps $O(\log n)$ utilitzant un hipercub amb n^3 processadors. El treball total és $O(n^3 \log n)$ i, per tant, l'algorisme no és òptim.

5.2 La mesh com a subgraf de l'hipercub

L'hipercub és un graf que té moltes simetries. Etiquetant de nou els nusos, podem construir una aplicació que a qualsevol nus i qualsevol aresta li fa correspondre un altre nus i aresta.

En particular, donat un hipercub amb n nusos, i donades dues arestes (u,v) i (u',v'), existeix un automorfisme σ tal que $\sigma(u) = u'$ i $\sigma(v) = v'$. Una manera efectiva de definir un automorfisme com aquest és la següent: Sigui $q = \log n$, siguin $u = u_1 \cdots u_q$ i $u' = u'_1 \cdots u'_q$ les representacions binàries de u i u', i siguin k i k', respectivament, les dimensions de (u,v) i de (u',v'). Per qualsevol permutació $\pi \in S_q$ tal que $\pi(k') = k$, definim

$$\sigma(x_1 \cdots x_q) = (x_{\pi(1)} \oplus u_{\pi(1)} \oplus u'_1) \cdots (x_{\pi(q)} \oplus u_{\pi(q)} \oplus u'_q),$$

on $\oplus$ és el *o excusiu* booleà i els punts denoten concatenació de símbols. Per exemple, si considerem l'hipercub a la figura 5.1, donades les arestes $(u,v) = (0010, 0011)$ i $(u',v') = (1011, 1001)$, tenim que $k = 4$ i que $k' = 3$. Considerem una permutació $\pi \in S_4$ tal que $\pi(3) = 4$; per exemple, $\pi(1234) = (2143)$. L'automorfisme tal que $\sigma(0010) = (1011)$ i $\sigma(0011) = (1101)$ ve definit per

$$\begin{aligned}\sigma(x_1, x_2, x_3, x_4) &= (x_2 \oplus 0 \oplus 1)(x_1 \oplus 0 \oplus 0)(x_4 \oplus 0 \oplus 1)(x_3 \oplus 1 \oplus 1) \\ &= (x_2 \oplus 1)x_1(x_4 \oplus 1)(x_3 \oplus 1).\end{aligned}$$

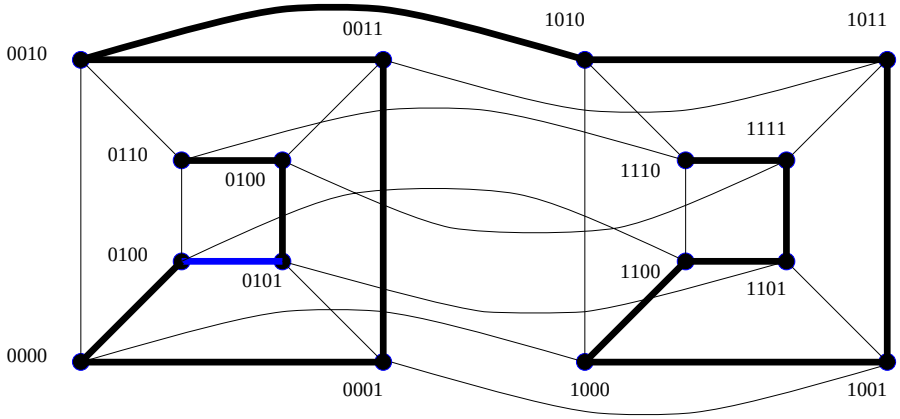


Figura 5.2: Un hipercub 4-dimensional. (En gruixut es destaca el vector lineal encaixat en ell.)

Una altra característica de l'hipercub és que conté com a subgrafs algunes de les xarxes que hem vist. Primer demostrarem que l'hipercub conté un vector lineal com a subgraf.

Lema 5.2. *Per a $q \geq 0$, un hipercub amb $n = 2^q$ processadors conté com a subgraf un vector lineal amb n processadors.*

Demostració. Inducció sobre n. Si $n = 1$, el lema és obvi. Suposem que el lema és cert per a $n/2$. Dividim l'hipercub amb n processadors en dos hipercubs amb $n/2$ processador eliminant les arestes amb dimensió $\log n$. Per la hipòtesi inductiva, cadaun dels hipercubs conté un vector lineal amb $n/2$ processadors. Com que són simètrics, el primer element en el vector lineal del primer hipercub i el primer element en el vector lineal del segon hipercub estan connectats, llavors podem construir un vector lineal a l'hipercub amb n processadors. □

Per tant, qualsevol dels algorismes dissenyats per a un vector lineal funciona amb la mateixa eficiència sobre un hipercub amb el mateix nombre de processadors.

Considerem ara el cas de la mesh. Primer de tot recordem-ne una definició. Donats k grafs $G_1 = (V_1, A_1), \ldots, G_k = (V_k, A_k)$, el seu *producte encreuat* (cross product) $G = G_1 \otimes \cdots \otimes G_k$ és un graf $G = (V, A)$ on $V = V_1 \times \cdots \times V_k$ i el conjunt d'arestes és

$$A = \{\{(u_1, \ldots, u_k), (v_1, \ldots, v_k)\} \mid \exists j (u_j, v_j) \in A_j \wedge \forall i \neq j \; u_i = v_j\}.$$

Observem que el producte encreuat és associatiu.

Qualsevol mesh es pot expressar com el producte encreuat de dos vectors lineals.

Lema 5.3. *El graf d'una mesh $n \times m$ és el* producte encreuat *d'un vector lineal amb n nusos i un vector lineal amb m nusos.*

Demostració. Per comprovar l'afirmació anterior que una mesh G amb dimensió $n \times m$ és el producte encreuat d'un vector lineal G_1 amb n nusos i un vector lineal G_2 amb m nusos, és suficient observar que dos nusos, (u_1, u_2) i (v_1, v_2), a G estan connectats per una aresta si $u_1 = v_1$ i $(u_2, v_2) \in A_2$, o si $u_2 = v_2$ i $(u_1, v_1) \in A_1$. □

Una extensió d'aquest argument ens dóna la descomposició següent per l'hipercub (vegeu l'exercici **5:5**),

Lema 5.4. *Un hipercub q-dimensional és el producte encreuat de q vectors lineals amb dos processadors cadascun.*

El lema següent dóna una caracterització de l'hipercub com a producte encreuat d'hipercubs amb dimensions més petites.

Lema 5.5. *Donat un hipercub q-dimensional H_q, si tenim que per a algun $k \geq 1$, $q = q_1 + q_2 + \cdots + q_k$, aleshores $H_q = H_{q_1} \otimes \cdots \otimes H_{q_k}$, on H_{q_i} denota un hipercub q_i-dimensional.*

Demostració. H_q es pot expressar com el producte encreuat de q vectors lineals amb 2 processadors i, com que H_{q_i} també pot ser expressat com el producte de q_i vectors lineals amb 2 processadors, per associativitat tenim el lema. □

Per demostrar el tipus d'hipercub que conté una mesh, necessitem el lema tècnic següent de la teoria de grafs.

Lema 5.6. *Si existeix un $k \geq 1$, tal que $G = G_1 \otimes \cdots \otimes G_k$ i $G' = G'_1 \otimes \cdots \otimes G'_k$ i per a tot i, $1 \leq i \leq k$, G_i és un subgraf de G'_i, aleshores G és un subgraf de G'.*

Demostració. Donat $G = (V, A)$, el demostrar que G és un subgraf de G' és equivalent a construir una aplicació del nusos de G als nusos de G' que preservi arestes. Per a cada $i, 1 \leq i \leq k$ sigui σ_i una aplicació dels nusos de G_i als nusos de G'_i que preserva arestes. Definim $\sigma : G \to G'$ de la manera següent: Per a cada nus $v = (v_1, v_2, \ldots, v_k)$ a G, $\sigma(v) = (\sigma_1(v_1), \sigma_2(v_2), \ldots, \sigma_k(v_k))$.

Si $(u, v) \in A$, aleshores existeix un j tal que $(u_j, v_j) \in A_j$, amb $u_i = v_i, \forall i, i \neq j$. Per tant, $(\sigma_j(u_j), \sigma_j(v_j)) \in A'_j$ i $\forall i, i \neq j, \sigma_i(u_i) = \sigma_i(v_i)$ cosa que implica $(\sigma(u), \sigma(v)) \in A'$. Per tant, σ preserva arestes. □

Aplicant directament els dos lemes previs, obtenim el resultat següent,

Teorema 5.1. *Qualsevol mesh de dimensió $2^{q_1} \times \cdots \times 2^{q_k}$ és un subgraf d'un hipercub amb $n = 2^q$ nusos, on $q = q_1 + \cdots + q_k$.*

Una altra manera d'enunciar el teorema anterior és dir que qualsevol mesh amb $m_1 \times m_2 \times \cdots \times m_k$ processadors pot ser encaixada a un hipercub amb n processadors, on $n = 2^{\lceil \log m_1 \rceil + \lceil \log m_2 \rceil + \cdots + \lceil \log m_k \rceil}$.

Per tant, tots els algorismes dissenyats per a una mesh poden ser directament implementats sobre un hipercub amb aproximadament el mateix nombre de processadors i utilitzant el mateix nombre de pasos. Evidentment, a partir de l'aplicació definida a la demostració del lema 5.6 podem trobar la correspondència concreta de cada nus a la mesh al seu nus corresponent a l'hipercub. A l'exercici **5:4** es demana de produir una aplicació concreta per demostrar com un algorisme sobre la mesh funciona sobre un hipercub.

5.3 Encaixaments a l'hipercub

Contràriament al que es podria pensar, l'hipercub no conté com a subgraf un arbre binari complet.

Lema 5.7. *L'hipercub amb $n = 2^q$ processadors no conté com a subgraf un arbre binari complet amb $n-1$ processadors.*

Demostració. A continuació demostrarem aquesta afirmació utilitzant la tècnica de la contradicció. Direm que un nus d'un hipercub té *paritat parella* si la cadena binària associada al nus té un nombre parell d'uns, altrament el nus té *paritat senar*. Un hipercub amb n nusos té $n/2$ nusos amb paritat parella i el mateix nombre de nusos amb paritat senar.

Si tenim que un arbre binari amb $n-1$ nusos és un subgraf d'un hipercub amb n nusos, l'arrel de l'arbre correspondrà a un nus de l'hipercub. Sense pèrdua de generalitat, considerem que aquest nus té paritat parella, aleshores tots els seus veïns tenen paritat senar i, per tant, els fills de l'arrel de l'arbre binari estan continguts en nusos amb paritat senar.

Seguint aquest raonament, tenim que en els nivells de l'arbre alternen nusos amb paritat parella i senar. Així, totes les fulles a nivell $\log n - 1$ i els nusos a nivell $\log n - 3$ estan continguts en nusos amb la mateixa paritat a l'hipercub, cosa que és impossible, ja que el nombre de nusos en aquests dos nivells d'un arbre binari complet sumen $5n/8$, que és més gran que el nombre màxim de nusos amb la mateixa paritat a l'hipercub. Per tant, és impossible tenir un arbre binari complet amb $n-1$ nusos dins d'un hipercub amb n nusos. □

Encara que un arbre binari complet no pot ser un subgra de l'hipercub, el podem *encaixar* (embedded) a l'hipercub. Diem que un graf $G = (V, A)$ es pot encaixar a un altre graf $G' = (V', A')$ si existeix una funció $f : V \to V'$ tal que si $(x, y) \in A$, aleshores o $f(x) = f(y)$ o $(f(x), f(y)) \in A'$. Observem que si podem definir un encaixament en el qual tots els nodes van a parar a diferents llocs, aleshores de fet el graf G és un subgraf de G'.

Entre les maneres d'encaixar l'arbre binari complet a l'hipercub la que donem aquí és particularment simple i útil. Definim l'aplicació següent de l'arbre a l'hipercub: comptant d'esquerra a dreta i començant per 0, la i-èsima fulla de l'arbre anirà al i-èsim nus de l'hipercub; i cada nus intern de l'arbre anirà al mateix nus que el seu descendent esquerre.

Evidentment aquesta aplicació no és injectiva; per exemple, tots els $\log n + 1$ nusos de la branca més esquerra de l'arbre, des de l'arrel fins a la fulla 0, aniran al nus $0 \cdots 00$ de l'hipercub, mentre que únicament la fulla $n-1$ anirà al nus $1 \cdots 11$. Aquesta aplicació es pot interpretar com xafar l'arbre binari contra les seves fulles, i fer que cada nus interior es projecti sobre el seu descendent esquerre i els $n-1$ nusos d'aquest arbre xafat siguin assignats en ordre als nusos de l'hipercub.

Aquesta aplicació, malgrat no ser injectiva té propietats que la fan molt interessant. Cada aresta de l'arbre va a un únic nus o a una única aresta de l'hipercub. Si definim les arestes de l'arbre que van de nusos a nivell i cap a nusos a nivell $i+1$, com arestes a *nivell* i, aleshores totes les arestes a nivell i que uneixen nusos a nivell $i-1$ amb els seus fills

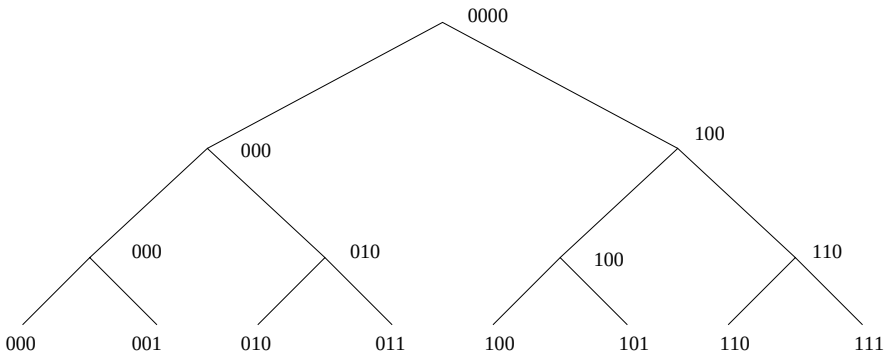


Figura 5.3: Arbre binari complet amb els nusos enumerats i enumeració del nus de l'hipercub on van a parar quan fem l'encaixament.

van a arestes amb dimensió k a l'hipercub. A més, com a màxim un nus per nivell va al mateix nus a l'hipercub. Aquestes propietats impliquen el resultat següent:

Lema 5.8. *Qualsevol algorisme sobre un arbre binari que utilitza únicament processadors al mateix nivell pot ser simulat pas a pas utilitzant un hipercub. A més, qualsevol transmisió de dades que al mateix pas involucri únicament arestes a nivell k pot ser realitzada en un pas per l'hipercub utilitzant arestes de dimensió k.*

Molts dels algorismes dissenyats per a topologies d'arbres tenen la característica que a cada pas de la computació únicament s'utilitzen processadors o arestes al mateix nivell. Utilitzant el lema anterior, aquests algorismes es poden simular amb un hipercub amb el mateix nombre de passos i amb la meitat dels processadors que necessita l'arbre i. A més, l'algorisme per a l'hipercub utilitza a cada pas únicament arestes amb la mateixa dimensió. Si l'algorisme sobre l'arbre té la propietat afegida que a passos consecutius sempre s'utilitzen nivells consecutius, aleshores l'algorisme sobre l'hipercub també té la propietat que a passos consecutius utilitzarà arestes amb dimensions consecutives. Els algorismes sobre l'hipercub que tenen aquesta propietat s'anomenen **normals** (vegeu l'exercici **5:3**).

L'estratègia descrita també serveix per trobar un encaixament eficient de la taula d'arbres a l'hipercub. Podem estendre el mètode descrit de la projecció d'arbres binaris sobre les seves fulles i l'encaixament d'aquestes sobre els nusos d'un hipercub, per aconseguir un encaixament d'una taula d'arbres a un hipercub.

Sigui una taula d'arbres r-dimensional $n \times n \times \cdots \times n$. Estenent el mètode descrit, projectem la taula d'arbres sobre la seva r-dimensional graella de processadors que formen les fulles. Després podem encaixar aquesta graella de n^r fulles als nusos d'un hipercub amb n^r nusos per via de l'aplicació següent: la fulla $(i_1, i_2, \ldots, i_r)$ va al nus $w_1 w_2 \cdots w_r$, on w_j és la representació binària de i_j que utilitza $\log n$ bits. Aquest encaixament de la taula d'arbres a un hipercub ens permet simular qualsevol algorisme per a la taula d'arbres utilitzant un hipercub, però amb un factor de retard de r passos, degut al fet que les arestes de r arbres diferents a la taula d'arbres poden anar a parar a la mateixa aresta de l'hipercub i, per tant, és possible que durant la simulació tinguem una congestió de fins a r elements.

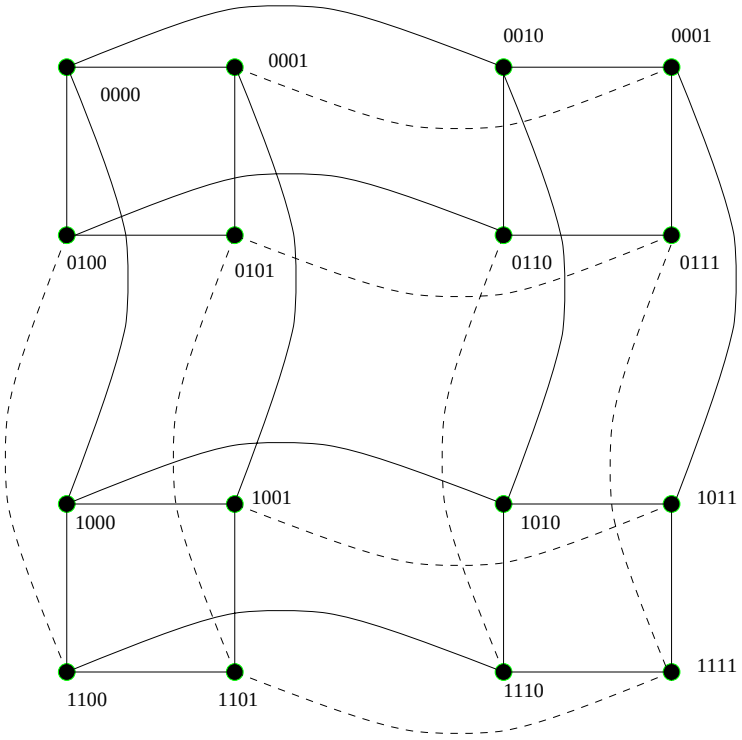


Figura 5.4: Encaixament d'una taula d'arbres 4×4 a un hipercub amb 16 nusos. (En línia discontínua, les arestes de l'hipercub no utilitzades per l'encaixament.)

Lema 5.9. *Qualsevol algorisme sobre una taula d'arbres, que a cada pas de la computació utilitza únicament el mateix nivell de nusos i arestes dels arbres, es pot implementar sobre un hipercub amb n^r nusos, amb un retard d'un factor constant multiplicatiu de r passos respecte al temps de l'algorisme sobre la taula d'arbres.*

Aquest factor de retard no té massa importància, ja que per la gran majoria dels algorismes que s'utilitzen tenim un valor de $r \in \{2, 3\}$. Per al cas $r = 2$, es pot eliminar completament el factor de retard (vegeu l'exercici **5:13**).

Exemple 5.3 (Multiplicació de matrius). *Considerem l'algorisme per a la multiplicació de dues matrius amb dimensions $n \times n$ utilitzant la taula d'arbres que vam veure a l'exemple 12 del capítol 4. Volem implementar-ho sobre un hipercub.*

Recordem que l'algorisme per computar $AB = C$ el que fa es entrar a_{ik} a l'arrel de l'arbre (i, k) amb dimensió 3, i entra b_{ki} a l'arrel de l'arbre (k, j) amb dimensió 1. Els valors d'A i B es passen cap avall en els arbres, mentre que la computació del producte $a_{ik}b_{kj}$ es realitza en el processador i, k, j. El valor c_{ij} s'obté sumant tots els valors obtinguts a les fulles de l'arbre (i, j) amb dimensió 2.

Per implementar aquest algorisme sobre un hipercub amb n^3 processadors, entrem a_{ik} al nus $\text{bin}(i)\text{bin}(k)00\cdots00$ a l'hipercub, i entrem b_{kj} al nus $00\cdots00\text{bin}(k)\text{bin}(j)$.

Al pas $t, 1 \leq t \leq \log n$, de la simulació sobre l'hipercub, el valor a_{ik} es transmet a través de les arestes amb dimensió $2\log n + t$, dels nusos $\text{bin}(i)\text{bin}(k)w_t00\cdots00$ fins als $\text{bin}(i)\text{bin}(j)w_t10\cdots00$. En el mateix temps, per a tot $|w_t| = t-1$, el valor b_{kj} viatja per les arestes amb dimensió t, dels nusos $w_t00\cdots00\text{bin}(k)\text{bin}(j)$ als $w_t10\cdots00\text{bin}(k)\text{bin}(j)$. En el pas $\log n$, els a_{ik} i b_{kj} són al nus $\text{bin}(i)\text{bin}(k)\text{bin}(k)$ i en un pas es pot computar la multiplicació $a_{ik}b_{kj}$. Per obtenir c_{ij}, en el pas $2\log n - t + 1$ se sumen els continguts dels nusos $\text{bin}(i)w_t00\cdots0\text{bin}(j)$ i $\text{bin}(i)w_t10\cdots0\text{bin}(j)$, $\forall w_t : |w_t| = t-1$. El resultat s'emmagatzema al nus $\text{bin}(i)w_t00\cdots0\text{bin}(j)$. Aquesta suma es pot computar en $\log n$ passos; per tant, després de $2\log n$ passos, c_{ij} estarà contingut al nus $\text{bin}(i)00\cdots00\text{bin}(j)$.

5.4 La papallona

L'hipercub, com hem vist, és una xarxa potent que pot simular eficientment qualsevol de les altres xarxes estudiades fins al moment. Però té un inconvenient: que el nombre de connexions a cada processador creix logarítmicament amb la grandària de la xarxa. En hipercubs amb pocs processadors, això no representa cap problema, pero quan s'utilitza l'hipercub com a arquitectura subjacent a una màquina de propòsit general amb parallelisme massiu, la xarxa pot tenir un nombre elevat de processadors, i aleshores el grau de cada processador pot ser gran. Així, en un hipercub amb 10^7 processadors, cada processador té 20 veïns. Aquest increment del grau dels processadors en funció de la grandària complica la comunicació entre processadors a la xarxa. Amb vista a simplificar aquests problemes de comunicació, s'han dissenyat diversos models de xarxes que conserven les bones propietats computacionals de l'hipercub, però els processadors tenen el grau fitat per una constant petita (3 o 4). Entre les xarxes d'aquestes característiques, una de les més utilitzades és la **papallona** (butterfly).

Una papallona amb dimensió q té $(q+1)2^q$ nusos i $q2^{q+1}$ arestes. Els nusos són representats per parelles $< w, i >$, on $i, 0 \leq i \leq q$ és un enter que indica el *nivell* del nus, i w és un nombre binari amb q-bits que denota la *fila* on està el nus $(0\cdots0 \leq w \leq 1\cdots1)$. Dos nusos $< w, i >$ i $< w', i' >$ estan connectats per una aresta sii $i' = i + 1$ i es compleix una de les dues condicions següents:

1. $w = w'$, o
2. w i w' difereixen únicament al i-èsim bit.

Les arestes entre nusos a nivell i i nusos a nivell $i+1$ s'anomenen *arestes a nivell* $i+1$ Una variació comuna al model de papallona és l'anomenada *papallona tancada* (wrapped butterfly), que s'obté a partir de la papallona normal identificant els nusos a la primera i a la darrera columnes. Observem que en aquest cas tots els nusos tenen el mateix grau.

El lema següent ens dóna una primera relació entre la papallona i l'hipercub,

Lema 5.10. *Qualsevol computació sobre un hipercub amb n processadors pot ser simulada amb una papallona amb $n(\log n + 1)$ processadors, amb un factor de retard de $\log n$ passos.*

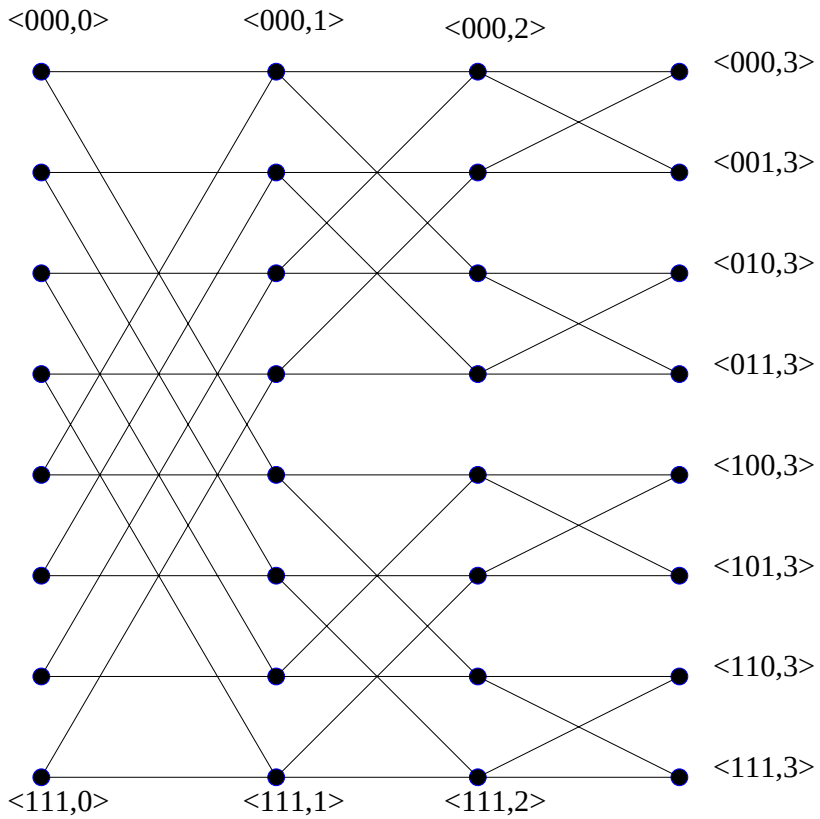


Figura 5.5: Una papallona amb dimensió 3.

Demostració. Observem que el i-èsim nus d'un hipercub q dimensional correspon a la fila i-èsima d'una papallona q dimensional. Una aresta (u, v) de dimensió i a l'hipercub correspon a les arestes $(< u, i-1 >, < v, i >)$ i $(< v, i-1 >, < u, i >)$, de nivell i a la papallona. Per tant, per a tot i, la i-èsima fila de la papallona pot simular l'operació realitzada pel nus i de l'hipercub. Per tant, un pas de l'hipercub pot ser simulat en $\log n$ passos de la papallona. □

De fet, és possible simular amb una papallona un algorisme **normal** dissenyat per a un hipercub, amb el mateix nombre de processadors i amb un factor de retard constant. Però la demostració escapa al propòsit d'aquest curs de nivell bàsic.

Una propietat molt útil de la papallona és el fet de que en una papallona q-dimensional, entre nusos $< w, 0 >$ i $< w', q >$ qualssevol hi ha un únic camí de longitud q.

La similitud entre l'hipercub i la papallona fa que la papallona també tingui les qualitats de l'hipercub. La demostració del següent lema es deixa com a exercici **5:8**.

Lema 5.11. *Una xarxa de papallona amb n nusos té un diàmetre de $O(\log n)$ i una amplada de bisecció de $\Theta(n/\log n)$.*

Una altra propietat que també comparteix la papallona amb l'hipercub és que té una estructura recursiva; si eliminem la primera columna en una papallona amb dimensió q obtenim dues papallones amb dimensió $q-1$, una que correspon a la part de sota i l'altra a la part de dalt. Però si eliminem la darrera columna, també obtenim dues papallones amb dimensió $q-1$, una formada per les files parelles i l'altra per les files senars.

Estudiem ara la implementació directa d'un algorisme sobre la papallona.

Exemple 5.4 (Classificació). *Anem a implementar l'algorisme d'ordenació per fusió parell-senar sobre una papallona.*

Donada una llista amb n elements, l'algorisme d'ordenació per fusió parell-senar comença per dividir la llista en n llistes de longitud 1, aleshores es fusionen parells de llistes i formen $n/2$ llistes ordenades, cadascuna de longitud 2. Aquestes llistes es fusionen dos a dos per obtenir $n/4$ llistes ordenades, cadascuna de longitud 4, i es continua així fins a fusionar dues llistes ordenades cadascuna, de longitud $n/2$.

El que fa eficient l'algorisme és la manera de fer la fusió. Per fusionar dues llistes ordenades $A = a_0, a_1, \ldots, a_{m-1}$ i $B = b_0, b_1, \ldots, b_{m-1}$ en una única llista ordenada L, dividim cadascuna de les llistes A i B en dues subllistes; una amb els elements amb subíndex parell i l'altra amb els elements amb subíndex senar. Per tant, es té $P(A) = a_0, a_2, \ldots, a_{m-2}$; $S(A) = a_1, a_3, \ldots, a_{m-1}$; $P(B) = b_0, b_2, \ldots, b_{m-2}$ i $S(B) = b_1, b_3, \ldots, b_{m-1}$. Utilitzant recursió, es fusiona $P(A)$ amb $S(B)$ per formar una llista ordenada, $C = c_0, c_1, \ldots, c_{m-1}$. De la mateixa manera, es fusiona $S(A)$ i $P(B)$ per formar $D = d_0, d_1, \ldots, d_{m-1}$. Utilitzant C i D construïm $L' = c_0, d_0, c_1, d_1 \ldots, c_{n-1}, d_{n-1}$. La llista final ordenada L es forma comparant cada c_i amb el d_i següent, $(0 \leq i \leq m-1)$ i, si cal, intercanviar parelles per obtenir la llista final L. És fàcil de demostrar, que l'algorisme descrit ordena correctament qualsevol llista amb m elements.

Per implementar aquest algorisme amb una papallona, comencem per implementar la fusió de $A = a_0, \ldots, a_{m/2-1}$ i $B = b_0, \ldots, b_{m/2-1}$ sobre una papallona amb dimensió $\log m$. Primer, per a cada $0 \leq i < m/2$ entrem a_i al nus $< 0\text{bin}(i), \log n >$ i b_i al nus $< 1\text{bin}(i), \log n >$. A continuació passem el valor de a_i a través d'una aresta horitzontal i passem el valor de b_i a través d'una aresta encreuada, fins a nusos a nivell $\log m - 1$. Observem que en aquest moment les files parelles de la papallona $\log m$-dimensional contenen una papallona $\log m - 1$ dimensional, i es repeteix el mateix procediment amb les files senars a la papallona $\log n$ dimensional.

Per computar la part recursiva de l'algorisme, observem que, tal com s'han col·locat les dades als nusos a nivell $\log n - 1$, podem fusionar $P(A)$ amb $S(B)$ a la papallona continguda a les files parelles. De la mateixa manera, a la papallona continguda a les files senars, podem fusionar $S(A)$ amb $P(B)$. D'aquesta manera obtenim C i D. Observem que C i D queden preparats a la xarxa de manera que en un únic pas cada nus a nivell $\log m$ en una fila parella escolleix la mínima sortida d'entre els seus veïns a nivell $\log m - 1$, mentre que cada nus amb nivell $\log m$ a una fila senar escolleix la màxima sortida d'entre els seus veïns a nivell $\log m - 1$. El resultat és la llista ordenada L.

A les figures 5.6 i 5.7 es mostra la part de la xarxa que s'activa primer quan les dades travessen des de la columna $\log m$ fins a la columna 0 i després els nodes que calculen la funció max o min quan les dades travessen la xarxa de la columna 0 fins a la columna $\log m$.

Per estudiar la complexitat d'aquest algorisme, observem que en els primers $\log m$ passos les dades encreuan la papallona i inverteixen l'ordre dels elements a la llista B. Als següents $\log m$ passos, els elements fan una segona pasada per la xarxa, intercanviant els parells per deixar-los en l'ordre correcte. Per tant, la complexitat total és de $2 \log m$ passos. Utilitzant l'algorisme anterior, n/m parells de llistes, cadascuna amb $m/2$ elements, es poden fusionar en n/m llistes, cadascuna amb m elements en $2 \log m$ passos, i sobre una papallona amb dimensió $\log n$. Per fer-ho, observem que els primers $\log m + 1$ nivells de la papallona formen un conjunt de n/m papallones de dimensió $\log m$; per tant, podem assignar una d'aquestes papallones per a cada fusió. Utilitzant fusions successives, podem ordenar n elements amb una papallona amb dimensió $\log n$ en $\sum_{i=2}^{n} 2 \log i = \log^n + \log n$ passos.

Observ que la implementació del mateix algorisme, però dissenyat per a una mesh, utilitzant un hipercub per a simular-lo, i de l'hipercub a la papallona, tindria un cost $O(\log^2 n)$ passos, I, a més, en ser un problema normal, es pot simular sense retard.

5.4.1 Encaminament a la papallona

Considerem el problema d'encaminar n paquets a una papallona. La situació al començament és que cada nus $< n, 0 >$ conté un paquet que té com a destinació un nus $< \pi(u), \log n >$, on $\pi \in S_n$. Considerarem l'encaminament *on-line*, on a cada pas prenem la decisió de quin camí prendrà el paquet. Recordant l'encaminament de paquets a la mesh, una manera d'encaminar els paquets és enviar cada paquet cap a la seva destinació utilitzant l'únic camí amb longitud $\log n$. Aquest camí és el que defineix el diàmetre d'una

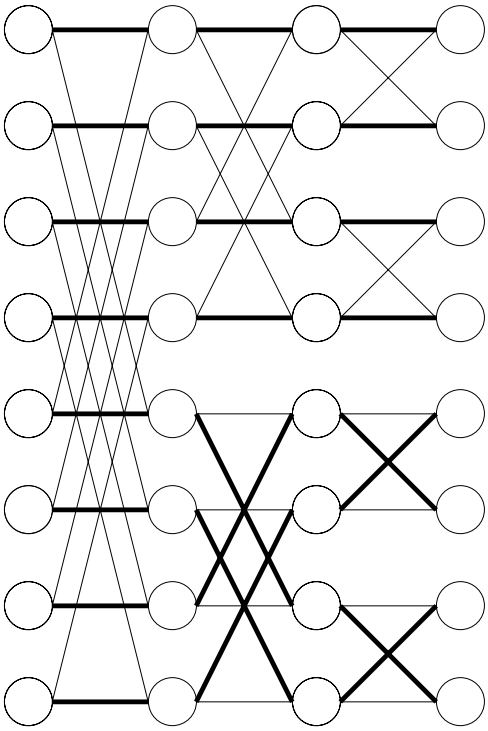


Figura 5.6: Connexions actives a la primera fase de l'algorisme de classificació per fusió parell-senar.

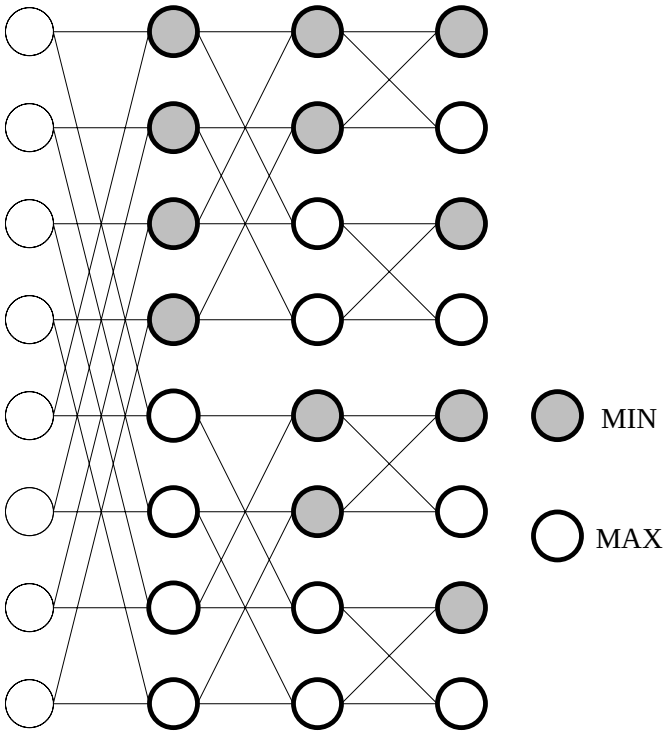


Figura 5.7: Operacions que es realitzen a la segona fase de l'algorisme de classificació per fusió parell-senar

papallona, s'anomena *camí voraç* i l'algorisme descrit és un algorisme voraç sobre la papallona. El problema amb l'algorisme voraç sorgeix quan l'apliquem en paral·lel a tots els n paquets, aleshores pot passar que més d'un paquet hagi de passar al mateix temps pel mateix nus o la mateixa aresta, és el que s'anomena una *congestió de la xarxa*. En aquesta situació, els paquets han de passar d'un en un, mentre que els que encara no han passat han d'esperar el seu torn.

El cas pitjor és dóna quan considerem la permutació que a cada element w li associa el element w^R. En aquest cas, sigui $\log n$ senar, considerem un paquet que ha d'anar del nus $< u_1 \cdots u_{(\log n-1)/2}00\cdots 0, 0 >$ fins al nus $< 0\cdots 00u_{(\log n-1)/2}\cdots u_1, \log n >$; el camí que seguirà serà:

$$\begin{aligned}
< u_0 \cdots u_{(\log n-1)/2}0, 0\cdots 0, 0 > \rightarrow & < 0, u_1 \cdots u_{(\log n-1)/2}0, 0\cdots 0, 0 > \\
\rightarrow \cdots \rightarrow & < 0\cdots 0, u_{(\log n-1)/2}0\cdots 0, 0 > \\
\rightarrow & < 0\cdots 0, u_{(\log n-1)/2}0\cdots 0, 0 > \\
\rightarrow \cdots \rightarrow & < 0, \cdots 0, u_{(\log n-1)/2}, \cdots u_0 >
\end{aligned}$$

Si ens fixem en l'aresta del mig, com que tenim $2^{\frac{(\log n-1)}{2}} = \sqrt{n/2}$ valors possibles de $u_0 \cdots u_{(\log n-1)/2-1}$, hi ha $\sqrt{n/2}$ paquets que han de passar per ella quan utilitzem l'algorisme voraç. Això significa que almenys un paquet s'endarrerirà $\sqrt{n/2} - 1$ passos i el nombre total de passos que l'algorisme voraç farà serà $O(\sqrt{n/2} + \log n)$.

Teorema 5.2. *Donat un problema d'encaminament a una papallona n-dimensional, on al començament tenim com a màxim un paquet per nus al nivell 0, i tenim cada paquet va destinat a un nus diferent a nivell* $\log n$*, l'algorisme voraç encaminarà tots els paquets en* $O(\sqrt{n})$ *passos.*

Demostració. Sigui n senar (l'argument per al cas de n parell és simètric). Si e és una aresta a nivell $i, 1 \leq i \leq \log n$, denotem per n_i el nombre de camins voraços que passen per e. Per a cada i tenim que $n_i \leq 2^{i-1}$. A més, com que des de e i utilitzant un camí $i+1, i+2, \ldots, \log n$ podem arribar com a màxim a $2^{\log n-i}$ nusos a nivell $\log n$, llavors $n_i \leq 2^{\log n-i}$.

Considerant que únicament $n_i - 1$ paquets poden endarrerir el pas d'un paquet per e, el temps màxim d'espera d'un paquet que creua els nivells $1, 2, \ldots, \log n$ és

$$\sum_{i=1}^{\log n}(n_i - 1) \leq 3\sqrt{n}/\sqrt{2} - \log n - 2$$

Per tant, el temps per encaminar els n paquets és $O(\sqrt{n})$ □

A la pràctica, per a valors petits de $n, (n \leq 100)$ el comportament de l'algorisme voraç no és dolent, ja que $\sqrt{n}$ i n no són molt diferents, però per a valors grans de n l'algorisme voraç pot trigar massa. Hi ha un ample ventall d'algorismes *on-line* eficients,

tant deterministes com probabilistes per a encaminaments a la papallona, en particular Ranade té un algorisme d'encaminament a la papallona amb $O(\log n)$ passos en el cas pitjor.

5.5 Referències bibliogràfiques

En aquest capítol hem presentat una revisió molt superficial de les xarxes de tipus hipercub i dels algorismes d'encaminament. Per al lector que vulgui aprofundir-hi, el capítol 3 del [Lei93] és una exposició magnífica sobre el tema. Aquest capítol inclou la demostració esmentada a la secció 5.4, que una papallona pot simular un hipercub sense increment de processadors o temps. L'algorisme millor per a l'encaminament *on-line* a la papallona va ser donat per Ranade [Ran87].

Exercicis

5:1 Demostreu el lema 5.1.

5:2 Dissenyeu un algorisme per a l'hipercub que resolgui el problema de la transmissió d'una dada emmagatzemada al processador 0 a tots els processadors de l'hipercub.

5:3 Implementeu la suma prefixada de $n = 2^q$ elements sobre un hipercub amb $n/2$ processadors. Recordeu que, donada una seqüència de n elements $\{x_1, \ldots, x_n\}$ d'un conjunt S i una operació binària $*$, les *sumes prefixades* són totes les operacions definides com $S_i = x_1 * x_2 * x_3 * \cdots * x_i$, per tot i, $1 \leq i \leq n$. (Ajut: Utilitzeu l'algorisme per l'arbre binari complet i encaixeu-lo sobre l'hipercub.)

5:4 Expliciteu amb cura com a partir de l'algorisme per la multiplicació de dues matrius sobre una mesh podem implementar sobre un hipercub la multiplicació d'una matriu $m \times s$ per una matriu $s \times n$.

5:5 Demostreu el lema 5.4.

5:6 Demostreu que una mesh de 3×5 no pot estar continguda dintre d'un hipercub amb 16 processadors.

Ajut: Utilitzeu un argument per contradicció. Etiqueteu les arestes de la mesh, on cada etiqueta indiqui la dimensió de l'aresta a l'hipercub. Demostreu que si l'afirmació és certa, les arestes incidents en un mateix processador a la mesh han de tenir etiquetes diferents i que les arestes paral·leles a la mesh tenen la mateixa etiqueta. (Per demostrar la darrera afirmació, considereu els cicles amb longitud 4 a la mesh). De les dues observacions prèvies es pot deduir que cada etiqueta d'una aresta vertical és diferent de cada etiqueta a una aresta horitzontal, i arribareu a una contradicció.

5:7 Implementeu el problema de trobar la clausura transitiva d'un digraf amb n vèrtexs sobre un hipercub. De quina grandaria ha de ser l'hipercub? Quin serà el temps que triga l'algorisme?

5:8 Demostreu que una papallona q-dimensional té diàmetre $O(\log n)$ i amplada de bisecció $\Theta(n/\log n)$

5:9 Demostreu que l'algorisme de classificació per fusió parell-senar ordena correctament qualsevol seqüència amb n elements arbitraris.

5:10 Dissenyeu un algorisme bitònic d'ordenació capaç d'ordenar un gran nombre de claus mitjançant un nombre petit de processadors. (Ajut: Utilitzeu pipeline i ordenació per fusió.

5:11 Expliqueu com fer la transposició d'una matriu de $n \times n$, utilitzant una papallona.

5:12 Dissenyeu un algorisme per a l'avaluació d'un polinomi, que utilitzi la papallona.

5:13 Construïu una aplicació d'una taula d'arbres amb dimensió $n \times n$, a un hipercub amb dimensió n^2, de manera que com a màxim un nus a cada nivel de la taula vagi a parar a cada nus de l'hipercub. Per tant, l'aplicació porta arestes de la taula cap a arestes de l'hipercub.

Capítol 6

La SIMD amb memòria compartida: La PRAM

Considerem ara una SIMD en què els processadors es comuniquen entre ells per via d'una memòria comuna, que s'utilitza de la mateixa manera que un grup de gent pot utilitzar una pissarra. Per exemple, si el processador P_i vol passar una dada al processador P_j, primer P_i escriu la dada a la memòria comuna, en una posició M_{ij} coneguda pels dos processadors, i després P_j la llegeix de la posició M_{ij}. El model més representatiu de SIMD amb memòria comuna és la **Parallel Random Access Machine**, PRAM, on cada un dels processadors és una màquina RAM seqüencial.

La PRAM no pot ser considerada un model directament implementable. Més endavant veurem la simulació de la PRAM per una SIMD de xarxa d'interconnexió. En la PRAM, ignorem el cost d'inicialització dels processadors i el cost de comunicació. Com ja hem dit, el nostr objectiu és considerar un model teòric que permeti estudiar quins són els mecanismes estructurals que fan que un problema pugui ser eficientment paral·lelitzat o no, ignorant els problemes de comunicació i sincronització. En particular, la PRAM permet eliminar la complexitat algorísmica respecte a la sincronització, seguretat, granularitat, comunicació entre processadors, i ens permet dissenyar algorismes focalitzats en les propietats estructurals del problema en consideració. El resultat és un nombre d'algorismes eficients, dissenyats en aquest model, i un nombre elevat de paradigmes i mètodes per al disseny d'algorismes paral·lels, que són utilitzables sobre màquines reals de paral·lelisme massiu.

6.1 La PRAM

Recordem que la **Random Access Machine**, RAM (vegeu la figura 6.1) és un model seqüencial de computació format per:

1. una memòria, amb un nombre arbitrari de posicions de memòria,

2. una unitat d'interconnexió, que permet accedir a qualsevol posició de la memòria,

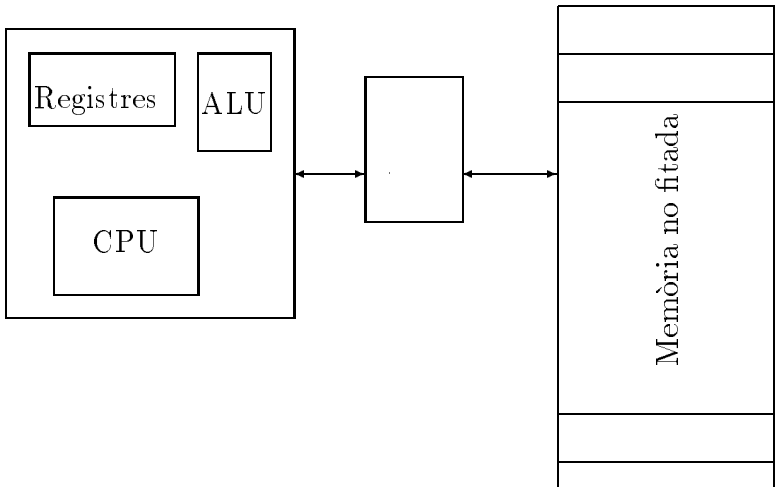


Figura 6.1: Màquina RAM.

3. un únic processador amb memòria local, que pot agafar i emmagatzemar dades de i a la memòria. Aquest processador també pot executar dades aritmètiques i lògiques.

Cada pas d'un algorisme consisteix en el següent:

- el processador llegeix una dada de la memòria i l'emmagatzema a la seva memòria local,
- el processador realitza operacions elementals (comparar, sumar, etc.) sobre les dades que conté a la memòria local,
- el processador n'escriu el resultat a la memòria.

Definirem ara un model paral·lel anàleg a la RAM. Una PRAM és un model SIMD amb memòria compartida i consta d'una memòria comuna, n processadors que comparteixen la memòria comuna i la utilitzen per comunicar-se entre ells, llegir les dades, escriure resultats intermedis o finals. Cada processador també té memòria local per emmagatzemar una petita quantitat d'informació i realitzar operacions aritmètiques o lògiques. Hi ha un control comú a tots els processadors que fa que, en un moment donat, tots els processadors executin la mateixa instrucció. A la figura 6.2 es dóna un esquema de la màquina.

Les tasques principals d'un algorisme paral·lel són:

- **la lectura**: Com a màxim n processadors llegeixen n posicions de la memòria comuna com a màxim, en temps constant. Cada processador llegeix, com a màxim, d'una única posició de memòria. Les dades obtingudes s'emmagatzemen a la memòria local de cada processador.
- **la computació**: Tots els processadors executen la mateixa operació lògica o aritmètica sobre les dades emmagatzemades a les seves memòries locals. Pot ser el cas que en a

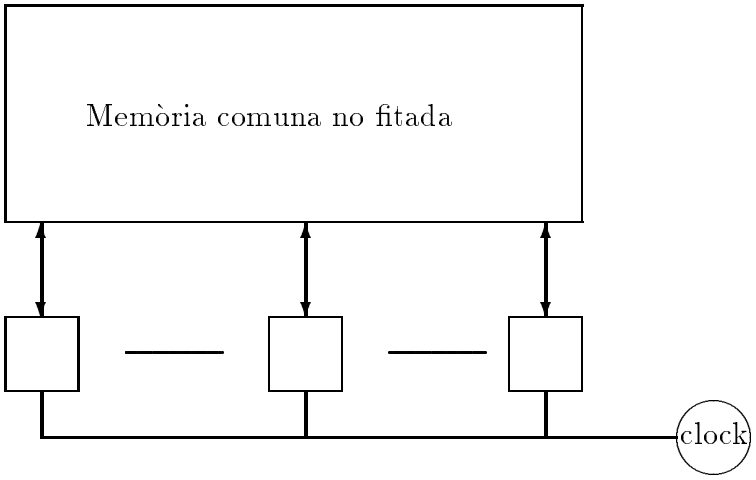


Figura 6.2: Màquina PRAM.

un pas donat, un o més processadors no estiguin actius mentre que la resta executen la mateixa instrucció.

- **l'escriptura**: Com a màxim n processadors escriuen sobre n posicions de memòria en temps constant. Cada processador escriu sobre una única posició de memòria.

Les PRAM es classifiquen en 3 tipus, d'acord amb les seves capacitats de lectura i escriptura:

- **EREW** (PRAM amb lectura i escriptura exclusiva). Dos processadors no poden escriure o llegir al mateix temps sobre la mateixa posició de memòria.
- **CREW** (PRAM amb lectura concurrent i escriptura exclusiva). Dos o més processadors poden llegir de la mateixa posició de memòria, però no hi poden escriure.
- **CRCW** (PRAM amb lectura i escriptura concurrents). Dos o més processadors poden llegir i escriure sobre la mateixa posició de memòria.

El fet de tenir diferents tipus de màquines implica que cal dissenyar els algorismes de manera que siguin correctes en el model de màquina utilitzat. Així, si utilitzem una CREW hem d'escriure algorismes que han de funcionar sense escriptura concurrent.

El fet de permetre que dos o més processadors puguin llegir de la mateixa posició de la memòria comuna, no presenta problemes importants, ja que el valor emmagatzemat no canvia i cal que cada processador faci una còpia de la dada a la seva memòria local. El problema es presenta quan diversos processadors intenten escriure sobre la mateixa posició de memòria. Aleshores es necessita un protocol per decidir quin és el processador que finalment escriu i quina cosa escriu. Hi ha diferents mecanismes de resolució de conflictes per a l'escriptura concurrent. Depenent del protocol tenim una nova classificació de les màquines PRAM-CRCW. Els protocols més usuals són:

1. **Prioritat.** Els processadors estan enumerats de manera que aquesta enumeració representan prioritats. Si més d'un processador intenta escriure a la mateixa posició de memòria, només ho aconsegueix el processador amb la prioritat màxima.

2. **Comú.** En cas de conflicte, els processadors que intenten escriure sobre la mateixa posició de memòria poden fer-ho si tots els processadors intenten escriure el mateix valor. Altrament, cap processador no escriu res.

3. **Arbitrari.** En cas de conflicte, un dels processadors que intenten escriure sobre la mateixa posició de memòria escriurà. Això vol dir que els algorismes han de ser correctes independentment de quin processador escrigui.

4. **Mínim.** Si dos o més processadors intenten escriure sobre la mateixa posició de memòria, finalment ho aconsegueix el processador que intenta escriure la dada més petita.

5. **Suma.** Quan dos o més processadors intenten escriure sobre la mateixa posició de memòria, finalment s'escriu la suma dels valors.

Hi ha altres tipus de protocols per a escriptura concurrent, però aquests ja són suficients. De fet, com veurem més endavant, alguns d'aquests models són un xic massa potents.

6.2 Complexitat en la PRAM

Per dissenyar algorismes per a les PRAM, ens interessa suposar que el nombre de processadors de la PRAM és il·limitat. Donada una entrada de grandària n per a un problema, la complexitat de l' algorisme PRAM es mesura pel nombre de cicles, en el cas pitjor, que realitza la màquina per resoldre el problema sobre l'entrada donada; això és el temps $T(n)$, si comptem el nombre total d'operacions, en el cas pitjor, realitzades al llarg de l'execució de tots els cicles, tenim una mesura del treball $W(n)$ realitzat per l'algorisme.

Evidentment, per obtenir el màxim de paral·lelització, necessitaríem un nombre de processadors igual al màxim nombre d'operacions per realitzar en un cicle de la computació. Com veurem, aquesta minimització del temps fent-lo igual al nombre de cicles sovint no dóna algorismes òptims, perquè en gran part del temps molts processadors són inactius. En aquests casos, és millor utilitzar menys processadors encara que això impliqui que el nombre de cicles sigui més gran. El resultat següent ens diu que si reduïm el nombre de processadors utilitzats per un algorisme PRAM, ho farem a costa d'incrementar el temps total de computació. Aquest resultat s'anomena *principi de seqüenciació de Brent*

Lema 6.1 (Principi de Brent). *Donat un algorisme paral·lel amb temps $T(n)$, sigui $w_i(n)$ el nombre d'operacions que ha de realitzar en el pas i-èsim, sigui $W(n) = \sum_{i=1}^{T(n)} w_i(n)$ el nombre total d'operacions realitzades, i sigui $m = \max_{1 \le i \le T(n)} w_i(n)$. Si implementem l'algorisme amb una* PRAM *amb $p(n)$ processadors ($1 \le p(n) \le m$), aleshores el temps paral·lel total de l'algorisme serà $\frac{W(n)}{p(n)} + T(n)$*

Demostració. Tenim un algorisme que necessita $T(n)$ cicles per resoldre un problema i cada cicle $i, 1 \leq i \leq T(n)$, ha d'implementar $w_i(n)$ operacions idèntiques. Sigui $W(n) = \sum_{i=1}^{T(n)} w_i(n)$ el nombre total d'operacions per realitzar. Si utilitzem un nombre $p(n)$ de processadors, cada cicle necessitarà $w_i(n)/p(n) + 1$ passos per ser implementat; per tant, el nombre de passos totals és

$$\sum_{i=1}^{T(n)} \left(\frac{w_i(n)}{p(n)} + 1 \right) = \frac{W(n)}{p(n)} + T(n),$$

i d'aquesta manera queda demostrat el principi de Brent. □

El principi de Brent ens dòna el cost de simular un algorisme PRAM amb temps $T(n)$ i treball $W(n)$ per un algorisme seqüencial, ja que un algorisme seqüencial no és més que un algorisme amb un únic processador. L'algorisme seqüencial anirà computant els cicles de l'algorisme paral·lel, un darrere l'altre, i les operacions a cada cicle de forma seqüencial, seguint un ordre arbitrari. El temps total de la simulació serà $W(n)$. Per tant, tenim el resultat següent,

Teorema 6.1. *Si per a un problema donat existeix un algorisme seqüencial A_s que és òptim i té complexitat t_s, qualsevol algorisme paral·lel per al mateix problema haurà de tenir una eficiència ≤ 1.*

Observem que encara que el treball sigui una fita més propera a la complexitat real de l'algorisme, sovint utilitzarem la cota superior donada pel cost. Recordem que el cost és el producte del temps pel màxim nombre de processadors que és necessiten en algun pas. Així, dissenyarem els algorismes independentment del nombre de processadors, tenint en compte que el temps total sí que dependrà del nombre de processadors. I només quan ens interessi obtenir algorismes òptims tindrem en compte el treball real.

6.3 Exemples d'algorismes PRAM

Vegem ara com funciona una PRAM i quin llenguatge utilitzarem per descriure algorismes per aquest model. Amb vista a l'escriptura d'algorismes assumirem que a cada pas cada processador té assignat un nombre i que el processador té aquest nombre a la seva memòria local. Al llenguatge habitual hi afegirem una nova instrucció:

per tot ⟨condició⟩ **parfer**

aquesta instrucció farà que tots els processadors pels quals el seu identificador compleixi la condició executin en paral·lel el que vingui a continuació. També hem de diferenciar entre variables a la memòria global i variables a la memòria local en cada processador. Per defecte, tota variable serà a la memòria global. Vegen-ne un primer exemple:

Exemple 6.1 (Cerca). *Considerem de nou el problema de la cerca. Donat un vector amb n elements diferents no ordenats $S = \{s_1, \ldots, s_n\}$ i un element x, el problema consisteix a decidir si $x \in S$.*

Hem vist que l'algorisme seqüencial òptim té cost $O(n)$. L'algorisme paral·lel CERCAP, donat com a algorisme 4, utilitza dues variables, x i c. També utilitza una tabla S de n elements a la memòria global. En finalitzar, la variable c tindrà un 1 en cas que trobem x i 0 altrament. La línia 1 de l'algorisme 4 indica que s'activaran els n primers processadors.

Algorisme 4 Cerca a la PRAM.

```
      CERCAP(s[n], x)
1     per a tot 1 ≤ i ≤ n parfer
2           si x = s[i]
3           llavors c := 1
            fsi
      fper
```

A la línia 2 cada processador haurà de llegir a la memòria local els valors de x i $s[i]$. Així, doncs, aquest algorisme només serà correcte en una PRAM amb lectura concurrent. A la línia 3, tots els processadors que trobin l'element x escriuran un 1 a la posició de memòria on s'emmagatzema c. Per tant, si S no té elements repetits, l'algorisme funcionarà en qualsevol dels models amb lectura concurrent, però si té elements repetits l'algorisme només funciona en una PRAM amb escriptura concurrent, ja que els processadors han d'escriure el mateix, i necessitem una PRAM-CRCW comuna. Per tant, a la PRAM-CRCW comuna l'algorisme CERCAP resol el problema de la cerca amb elements repetits (i a la PRAM-CREW el problema de la cerca sense elements repetits) en temps constant, i utilitza n processadors.

Considerem ara una PRAM-EREW, amb n processadors; per poder resoldre el problema de lectura simultània considerem el t problema següen: A la memòria comuna, en una posició coneguda per tots els processadors tenim una dada x, i en una altra posició coneguda per tots tenim l'adreça c de la memòria comuna on comença el vector que conté els elements de S. Cada processador ha d'obtenir x i c, i emmagatzemar-les a la memòria local. Com que no poden llegir en concurrent, els n procesadors hauran de llegir en seqüencial, cosa que representa un temps de $O(n)$ passos. Després, en paral·lel i en temps constant, cada processador P_i compara x amb el contingut de la posició de memòria $M(c+i)$. Si un processador troba la x, la resposta és SI. (Observem que no hi pot haver més d'un processador que respongui SI, ja que hem assumit que els elements de S són diferents). Malgrat que la comparació en paral·lel es fa en temps constant, el fet d'haver de llegir x de processador en processador converteix l'algorisme descrit en altament ineficient, amb cost $O(n^2)$.

A l'exemple següent dissenyarem una tècnica fonamental per resoldre el problema d'**emissió** (broadcasting) d'una dada de la memòria comuna d'una PRAM-EREW en tots els

n processadors, utilitzant $O(\log n)$ passos.

Exemple 6.2 (Emissió). *Donada una* PRAM-EREW*, amb n processadors que, s'anomenen $P_1, \ldots, P_n$, considerem que a la posició 1 del vector $A[1..n]$ a la memòria comuna de la* PRAM*, hi ha una dada x i volem escriure aquesta dada en totes les posicions d'A (transmetre x a tots els n processadors) en el mínim nombre de passos.*

Per implementar l'emissió, utilitzarem una taula auxiliar $A[1..n]$. Es realitzan els passos següents:

1. P_1 llegeix x d'$A[1]$ i ho escriu a $A[2]$. Ja tenim dues còpies de x !
2. P_1 i P_2 llegeixen x d'$A[1]$ i d'$A[2]$ i l'escriuen a $A[3]$ i $A[4]$. I ja en tenim quatre !
3. Es continua duplicant fins que en tinguem les n còpies

En $O(\log n)$ passos, tots els n processadors tindran n còpies de x disponibles a la memòria global. L'algorisme que implementa aquest esquema es dóna com a algorisme 5.

Algorisme 5 Emissió d'un valor a tots els processadors.

```
      EMISSIÓ(A)
1     per j = 0 fins log n − 1 fer
2        per a tot 2^i + 1 ≤ j ≤ 2^{i+1} parfer
3              A[j] := A[j − 2^i]
         fper
      fper
```

Tornant a l'exemple 6.1, l'algorisme de cerca amb una PRAM-EREW es pot implementar en $O(\log n)$ passos, utilitzant la tècnica d'emissió.

Suposem ara que tenim $m < n$ processadors, on $n = |S|$, i volem resoldre el problema de la cerca amb només m processadors. Per implementar l'algorisme de cerca fem el següent:

1. Subdividim S en subcadenes de longitud n/m.
2. En paral·lel, P_1 cercarà seqüencialment entre els n/m primers elements, P_2 cercara seqüencialment entre l'element $n/m + 1$ i el $2n/m$, i així successivament.

La qüestió és com el processador P_i coneix on ha de començar i finalitzar la seva cerca seqüencial. Cada processador té n a la seva memòria local, coneix el nombre m de processadors i la posició k on comença S. (Totes aquestes dades es poden adquirir en $O(\log m)$ passos). Aleshores, cada P_i realitza les operacions següents a la seva memòria local: computa la posició de començament $\lfloor n/m \rfloor i + 1 + k$ i la posició de terminació $\lfloor n/m \rfloor (i+1) + k$.

Observem que en cas que un processador trobi la x, els altres processadors continuaran operant fins a explorar tot el seu segment. Aquesta ineficiència es pot millorar introduint-hi un nou registre diferenciat b (tots els processadors coneixen la seva adreça) que conté

una variable booleana. Al començament b està inicialitzat a 0. Quan un processador troba x, escriu 1 a b. A cada pas de l'algorisme, tots els processadors miren si b és a 0 o a 1 (utilitzant *broadcasting*). Quan es troben 1, s'aturen. Si $x \notin S$, la complexitat és elevada, $O(\log m(1+n/m))$ ja que hem de multiplicar cada pas per $O(\log m)$. Però si un processador troba x aviat, aleshores la complexitat es pot reduir molt.

Si en lloc d'utilitzar el model PRAM-EREW, utilitzem una màquina PRAM-CREW, aleshores ja no cal utilitzar emissió. La complexitat de l'algorisme paral·lel de cerca és $O(n/m)$. En cas que la PRAM tingui n processadors, la complexitat és $O(1)$.

Exemple 6.3 (Cerca amb repeticions). *Suposem ara que el conjunt S, on volem realitzar la cerca, pot tenir elements repetits.*

Si $x \in S$, amb l'algorisme que hem vist es pot donar el cas que dos processadors detectin x al mateix temps, amb un conflicte d'escriptura en intentar escriure els dos sobre la mateixa posició de memòria al mateix temps. Amb escriptura exclusiva, es pot fer un procediment simètric al d'emissió, però d'escriptura, anomenat **emmagatzemament** (store) i que, com l'emissió, es pot resoldre utilitzant $O(\log n)$ passos, on n és el nombre de processadors (vegeu l'exercici **6:1**). De tota les manera, si S té repeticions, la manera més eficient de cercar x a S és utilitzant una PRAM-CRCW, de manera que quan dos o més processadors intenten escriure 1 sobre b, puguin fer-ho, en temps constant.

Per veure la plena potència de l'escriptura concurrent, considerem l'exemple següent:

Exemple 6.4 (Classificació). *Considerem el problema d'ordenar una taula S amb n elements, $S = \{s_1, \ldots, s_n\}$, en temps constant.*

Utilitzarem una PRAM-CRCW amb protocol de suma i amb n^2 processadors, en què cada processador tindrà assignat un parell (i, j) com a identificador. L'algorisme 6 utilitza un vector auxiliar C amb dimenssió n on anirà el resultat de l'ordenació, i el vector S amb les dades d'entrada.

Per fer l'ordenació els processadors amb identificador (i, j) determinen quin dels dos valors és més gran. Cal tenir en compte que la posició final de l'element $S[i]$ és el nombre d'elements més petits que ell més un.

El temps d'ordenar n nombres és $O(1)$, però el treball és $O(n^2)$, per tant no és òptim respecte de l'ordenació seqüencial. Evidentment el protocol suma és molt potent, en el sentit que enmmascara una operació com la suma de n nombres. De fet, no tornarem a utilitzar aquest tipus de protocols tan potents. Malgrat això, aquest algorisme ha estat utilitzat per implementar l'ordenació sobre la NYU *ultracomputer* [GGK+83].

És possible classificar n elements en $O(\log n)$ passos i de manera òptima. No és massa difícil el trobar un algorisme no òptim per classificar n elements amb $O(\log n)$ passos utilitzant una PRAM-EREW (vegeu l'exercici **6:5**). A la secció següent utilitzarem l'existència de l'algorisme de classificació òptim de Coole per a la simulació entre els diferents tipus de PRAM.

Algorisme 6 Classificació a la PRAM.

```
     CLASSIFICACIÓ(S)
1    per a tot 1 ≤ i ≤ n parfer
2         per a tot 1 ≤ j ≤ n parfer
3              si (S[i] > S[j]) o (S[i] = S[j] and i > j)
4                 llavors C[i] := 1
5                 altrament C[i] := 0
                  fsi
6    per a tot 1 ≤ i ≤ n parfer
7         C[C[i] + 1] := S[i]
     fper
```

6.4 Relacions entre models de PRAM

La PRAM-EREW és el model més feble de PRAM, mentre que la PRAM-CRCW és el més fort. Però, el nombre de passos per simular una operació d'una PRAM-CRCW en una PRAM-EREW és $O(\log n)$.

Teorema 6.2. *Qualsevol problema que es pot resoldre en temps t amb una* PRAM-CRCW *prioritat i amb n processadors es pot resoldre utilitzant una* PRAM-EREW *amb n processadors i temps $O(t \log n)$.*

Demostració. Simularem cada pas de la PRAM-CRCW per $O(\log n)$ passos d'una màquina PRAM-EREW. Com que la potència de computació de les dues màquines és similar, únicament necessitarem preocupar-nos de l'accés a la memòria.

• Simulació de l'escriptura concurrent
Considerem que $P_1, P_2, \ldots P_n$ són els processadors tant de la CRCW com de l'EREW amb n prioritats diferents $p_1, \ldots p_n$. Sigui un pas qualsevol de la CRCW, on els processadors escriuen sobre posicions de memòria $M(1), M(2), \ldots, M(r)$, $r \leq n$. L'EREW que simularà el procés defineix a la seva memòria un vector A amb dimensió n; cada posició contindrà tres dades, la posició de memòria, la prioritat del processador i el valor que s'ha d'escriure. Si el processador P_i de la CRCW vol escriure x_i a la posició de memòria l_i de la CRCW, el processador P_i de la EREW escriu (en exclusiva) el parell (l_i, p_i, x_i) a la posició $A[i]$.

Després es fa una classificació dels continguts d'A utilitzant com a clau primària la primera component l_i i com a clau secundària la segona p_i. Com ja hem esmentat, utilitzant una EREW es pot fer en $O(\log n)$ passos. Per tant, tindrem que totes les dades escrites per la CRCW sobre la mateixa posició de memòria estaran en posicions consecutives d'A i, a més, classificades per prioritat això vol dir que només els processadors encarregats de la primera ocurrència d'una posició hauran d'escriure.

Si $A[1] = (l_j, p_j, x_j)$ aleshores el processador P_1 de l'EREW escriu x_j a la posició l_j de la memòria de l'EREW.

Per a $2 \leq i \leq m$, cada P_i de l'EREW compara $A[i]$ amb $A[i-1]$. Si $l_j \neq l_k$, aleshores P_i escriu x_k a la posició de memòria l_k. Altrament P_i no actua.

Com que el vector A està classificat per la primera coordenada, únicament un dels processadors escriurà a cada posició de memòria. Per tant, tindrem escriptura exclusiva. Cada moviment d'escriptura concurrent es pot simular amb una EREW en $O(\log n)$ passos.

• Simulació de la lectura concurrent

De la mateixa manera, podem simular la lectura concurrent. En lloc d'escriure llegirem, però la simulació per l'EREW és gairebé idèntica. Emmagatzema a A tots els parells (l_i, x_i, i), on l_i és la posició de la memòria de la CRCW sobre la qual el procesador P_i vol llegir. Observem que ara la prioritat no té cap importància però l'algorisme ha de saber a quin processador ha de tornar el valor llegit. Es fa primer una classificació per claus l_i; això col·locarà en posicions consecutives totes les peticions de lectura dels processadors de la mateixa posició l_1, després vindran els que volen llegir de l_2, i així successivament. A continuació el primer processador de cada bloc llegeix, després l'EREW utilitzant emissió copia una mostra per cada processador que volia llegir de l_i. En el darrer pas, cada processador copia el valor x_i a una posició M_i, d'on la llegirà el processador P_i. Observem que en aquest pas el processador que fa la còpia ha de ser el processador i, sinó el processador encarregat de la posició j de A tal que $A[j] = (l_i, x_i, i)$.

Tots els passos tenen cost constant, excepte la classificació i l'emissió, que es poden fer en $O(\log n)$ passos. □

Corol·lari 6.1. *Qualsevol problema que es pot resoldre utilitzant una* PRAM-CRCW *amb protocol prioritat amb $O(n)$ processadors i en t passos, es pot resoldre utilitzant una* PRAM-CREW *o una* PRAM-EREW *amb $O(n)$ processadors i temps $O(t \log n)$.*

Al teorema 6.2, hem utilitzat el model de prioritat per al protocol d'escriptura concurrent; per a altres tipus de PRAM-CRCW es poden obtenir el mateix tipus de resultats utilitzant idees similars. Observem que si volem utilitzar una PRAM-EREW no podem disminuir massa el temps de la simulació, però per ha simular diferents tipus de PRAM-CRCW podem fer que la simulació no alteri el temps de càlcul, incrementant el nombre de processadors.

Teorema 6.3. *Qualsevol problema que es pot resoldre utilitzant una* PRAM-CRCW *amb protocol de prioritat i amb n processadors, també es pot resoldre amb una* PRAM-CRCW *amb protocol comú, utilitzant el mateix nombre de passos i afegint $O(n^2)$ nous processadors*

Demostració. Sigui una PRAM-CRCW amb prioritat i amb n processadors, denotats per $P_1, P_2, \ldots, P_n$, que resol un problema donat en temps t. Observem que ara només cal simular una fase d'escriptura.

En un pas de càlcul els n processadors volen escriure a les posicions de la memòria comuna $M(1), \ldots, M(r)$. Considerem una CRCW amb protocol "comú" que simularà cada pas d'escriptura prioritària per escriptura "comuna" sense canvis en el nombre total de passos. Per fer la simulació, a més dels n processadors, la CRCW-comuna utilitzarà n^2 processadors més $\{P_{i,j}\}_{1 \leq i,j \leq n}$ i n noves posicions de memòria $A(j)$, $1 \leq j \leq n$, que al

començament estan inicialitzades a 0. Suposem que dos processadors, P_i, P_j, de la PRAM amb prioritat, volen escriure en les posicions de memòria $M(i)$ i $M(j)$. El processador $P_{ij}, (i \leq j)$ de la PRAM comuna determina si $M(i) = M(j)$, i si és així mirarà quin dels dos processadors té menys prioritat i escriurà un 1 a la posició del vector A que correspongui al menys prioritari. En la segona fase, el processador P_i de la PRAM comuna pot determinar si és el processador amb més prioritat que ha intentat escriure sobre $M(i)$. Això es pot determinar mirant si $A(j)$ encara conté 0. Si és així, P_j escriu sobre la posició $M(j)$ de la PRAM comuna, el valor que P_j a la PRAM prioritat escriu sobre $M(j)$. □

Per a altres simulacions, vegeu l'exercici **6:2**.

6.5 Consideracions sobre la SIMD amb memòria compartida

Les màquines SIMD amb memòria compartida són models potents de computació. El fet que al mateix moment tots els processadors puguin escriure o llegir de la memòria comuna en un temps constant li dóna molta potència. Evidentment és un model teòric que principalment serveix per extreure la natura del paral·lelisme a alguns problemes concrets, sense consideracions de costos de comunicacions entre processadors i costos d'implementació.

Quan un processador necessita accedir de una dada de la memòria compartida, cal una connexió física entre la posició de memòria i el processador. El cost de la transmissió s'expressa com el nombre de portes necessàries per descodificar la direcció donada pel processador. Si la memòria té m posicions, el cost de la descodificació es pot expressar com $f(m)$. Si la SIMD té n processadors, el cost total és $n \times f(m)$, que per a valors grans de n i m pot ser prohibitiu. Una manera de reduir el cost de descodificació és dividir la memòria comuna en R blocs amb m/R posicions de memòria cada un. Existeixen $n + R$ connexions entre els blocs de memòria i els processadors. Observem que únicament un processador pot escriure o llegir en un bloc donat en un instant donat. En aquest cas, el cost de descodificació és $R \times f(m/R)$ (més el *cross-bar* necessari). Aquest model és més feble que la SIMD amb memòria comuna.

De tota manera, més endavant veurem que un cop hem dissenyat un algorisme per a una màquina PRAM, la seva implementació amb el model "realistic" de xarxes d'interconnexió es pot fer amb un cost baix.

6.6 Circuits i PRAM

UN altre model important de computació paral·lela són els circuits booleans. Un **circuit** és una xarxa acíclica de portes lògiques, cada porta amb un nombre fitat d'entrades. Considerem que cada porta té exactament una sortida. La **grandaria** d'un circuit és el nombre de portes que conté. La **profunditat** d'una porta és el nombre de portes al camí més llarg entre la porta i un dels elements d'entrada. La **profunditat** del circuit és la

profunditat màxima entre les portes que formen el circuit. Donat un circuit, el nivell k correspon a totes les portes que són a la mateixa profunditat k.

Teorema 6.4. *Qualsevol circuit com els definits, amb profunditat d i amb n portes, es pot simular per una* CREW-PRAM *amb m processadors en un temps $O(n/m + d)$.*

Demostració. Sigui C un circuit com l'especificat. Emmagatzemem els elements de C a la memòria comuna de la PRAM, deixant lloc de memòria perquè al costat de cada porta es pugui emmagatzemar el valor computat per la porta.

Cada processador de la PRAM simularà una porta de la manera següent: el processador llegeix els valors d'entrada, de la posició de memòria, i en temps $O(1)$ simula la funció lògica que correspon al tipus de porta, i n'escriu el resultat al lloc reservat a la memòria comuna.

Globalment la simulació de C es realitzarà per nivells, d'aquesta manera ens assegurem que quan simulem una porta ja tenim computades les seves entrades. Al primer pas, la PRAM simula les portes al primer nivell, ja que les seves entrades únicament depenen de les entrades del circuit. En el segon pas, se simularan les portes al segon nivell, i continuarem així fins a simular tots els nivells. A cada nivell podem simular totes les portes al nivell en paral·lel, ja que les seves computacions són independents.

Per a $i = 1, \ldots, n$, sigui n_i el nombre de portes a nivell i, es compleix que $\sum_{i=1}^{d} n_i = n$. Si tenim m processadors, per simular n_i portes necessitarem $O(\lceil n - i/m \rceil)$. Per tant, el temps total de simulació serà $n/m + d$ □

El fet que cada porta pugui tenir un nombre no fitat de sortides fa imprescindible que la PRAM tingui lectura concurrent. Si la sortida està fitada, és fàcil veure que podem realitzar la simulació amb un model EREW. Els detalls de la demostració del resultat següent es deixa com a exercici.

Corol·lari 6.2. *Qualsevol circuit amb profunditat d, grandaria n, i on cada porta té fitat el nombre d'entrades i de sortides, pot ser simulat per una* PRAM-EREW *amb m processadors en un temps $O(n/m + d)$.*

6.7 Referències bibliogràfiques

Algunes de les referències d'aquest capítol ja han estat donades en el text. El model PRAM va ser introduït a [FW78]. Després ha estat àmpliament utilitzat. Hagerup [Hag88] dóna una construcció explícita d'un programa per a la màquina PRAM, que computa la disjunció de n variables booleanes. El programa utilitza instruccions del tipus RAM, com per exemple LOAD, STORE, READ, etc. Al mateix treball, Hagerup presenta amb força cura com emmagatzemar a la memòria comuna d'una PRAM, diferents estructures combinatòries. El nostre objectiu és utilitzar la PRAM com a model abstracte de càlcul, i no entrarem en detalls. Altres introduccions a la PRAM, més adients al nostre objectiu amb les simulacions entre diferents models de PRAM, s'exposen als llibres [JáJ92, GR90] i a [KR90].

El primer algorisme per classificar n elements en $O(\log n)$ passos i de manera òptima. va ser la implementació sobre una PRAM-EREW de l'algorisme de classificació per a xarxes de comparació que van dissenyar Ajtai, Komlos i Szemeredi[AKS83]. Una versió "entenedora" de l'algorisme d'Ajtai es pot trobar al capítol 5 del llibre [GR88]. El problema amb l'algorisme d'Ajtai, Komlos i Szemeredi és que en l'expressió del temps paral·lel $T(n) = c \log n$ la constant c és massa gran per considerar l'algorisme "pràctic". Existeix un algorisme òptim ($T(n) = O(\log n), W(n) = O(n \log n)$) i "pràctic" per classificar n elements amb una PRAM-EREW [Col88].

Un camp de recerca molt actiu gira al voltant de la utilització de les idees desenvolupades amb les màquines SIMD amb memòria comuna per dissenyar màquines reals de paral·lelisme massiu. Per exemple, els projectes següents tenen com a objectiu dissenyar màquines inspirades en la PRAM: PUMA ESPRIT (INMOS); TOUCHSTONE (INTEL, DARPA, CALTECH); J-MACHINE (MIT, INTEL). També s'han desenvolupat models teòrics de PRAM, en què d'alguna manera es consideren costos de comunicació, com la BSP [Val90, Val91] o la YPRAM [TK91]. Un dels models que ha adquirit més rellevància és el model logP de [CKP+93]. El camp dels models teòrics de paral·lelisme síncron per via de memòria compartida és una línia de recerca molt interessant. Malauradament l'estudi d'aquests models escapa als objectius d'un curs bàsic. Una visió global molt bona s'exposa a l'article [McC93].

Exercicis

6:1 Una altra manera de simular una PRAM-CRCW comuna amb una PRAM-EREW és la següent: la lectura concurrent en una PRAM-CRCW pot ser simulada per *broadcasting*. L'escriptura concurrent pot ser simulada pel procediment simètric del*broadcasting*, que anomenarem *store*. Suposem que n processadors d'una PRAM-CRCW comuna, poden escriure sobre la posició de memòria $M(j)$ (finalment escriuran si tots pretenen escriure el mateix valor). Sigui x_i el valor que P_i vol escriure. El procediment de simulació perquè l'EREW conegui si tots els processadors a la CRCW volen escriure el mateix valor és:

- Per a tot i, $1 \leq i \leq n/2$, si $x_i = x_{i+n/2}$ aleshores P_i posa la variable booleana b_i a 1. Altrament, $b_i = 0$.
- Per a $1 \leq i \leq n/4$, si $b_i = b_{i+n/4} = 1$ i a més $x_i = x_{i+n/4}$, aleshores P_i posa $b_i = 1$, altrament $b_i = 0$.

Després de $\log n$ passos, P_1 sap si tots els valors x_i són iguals. Si ho son, P_1 escriu x_i a $M(j)$. Altrament, P_1 no escriu res. Doneu una subrutina STORE per implementar el procediment descrit.

6:2 Indiqueu com simularíeu:

1. Una PRAM-CRCW suma per una PRAM-CRCW prioritat.
2. Una PRAM-CRCW arbitraria per una PRAM-CRCW comuna.

Quin cost té la simulació, en temps i nombre de processadors?

6:3 Dissenyeu un algorisme per computar les funcions booleanes sobre n variables AND i OR, amb una màquina PRAM-CRCW comuna amb n processadors i temps constant $O(1)$.

6:4 Demostreu el corol·lari 6.1.

6:5 Donats n elements, dissenyeu un algorisme no necessariàment òptim, per classificar els n elements en $O(\log n)$ passos. Quin serà el treball?

6:6 Donada una seqüència de k elements, $l = l_1, \ldots, l_k$ dissenyeu un algorisme PRAM per poder trencar la seqüència en subseqüències *maximals* sense cap element repetit. És a dir, suposant que ens donin la seqüència *aabacdcbbacbc* les subseqüències són a, ab, acd, cb, bac, bc. (Ajut: Penseu com es podria calcular el punt de tall si per a cada posició calculéssim el punt on tallaríem quan prenem la subseqüència sense repeticions amb menys de k caràcters a partir d'aquesta posició.)

Capítol 7

Algorismes PRAM bàsics

En aquest capítol presentarem alguns algorismes pel model PRAM de computació. Aquests algoritmes s'utilitzen molt sovint com a eines auxiliars per resoldre problemes més complexos. En general, la computació d'un algorisme amb una PRAM segueix un esquema de graf amb nivells, on a cada nivell s'han d'executar en paral·lel les operacions representades pels nusos. Evidentment, per poder executar en un pas paral·lel totes les operacions al mateix nivell, és necessari tenir un nombre de processadors de, com a mínim, el nombre màxim de nusos per nivell, encara que, com ja hem vist, de vegades utilitzant el principi de Brent es poden repartir millor les tasques per realitzar entre els processadors.

7.1 La suma prefixada

Aquesta és una operació bàsica al llarg de tot el llibre. De fet ja van veure com resoldre el problema amb la topologia d'arbre binari. Recordem que, donada una seqüència de n elements $\{x_1, \ldots, x_n\}$ d'un conjunt S i una operació binària $*$, les **sumes prefixades** són tots els valors definits com

$$S_i = x_1 * x_2 * x_3 * \cdots * x_i$$

La fita inferior seqüencial del problema és $\Omega(n)$, perquè com a mínim hem de calcular n vegades l'operació $*$; a mes, l'algorisme seqüencial obvi té complexitat $O(n)$ i, per tant, és òptim.

L'algorisme bàsic es dóna com a algorisme 7. La idea és compactar els valors de dos en dos i emmagatzemar-ne els resultats intermedis per utilitzar-los després, de la mateixa manera com fèiem quan vam resoldre el problema en una xarxa amb topologia d'arbre binari. Donat com a entrada un vector $x = \{x_0, \ldots, x_{n-1}\}$ d'elements de S, amb l'element x_i col·locat a la posició $x[i]$ de memòria, i assumint que $n = 2^q$, en finalitzar l'execució la posició $x[i]$ conté la i-èsima suma prefixada.

L'algorisme PPRESUMS té dos bucles que realitzen cadascun $O(\log n)$ iteracions. Els dos fan el mateix nombre d'instruccions. A la primera iteració s'executen $n-1$ operacions, i en el cicle j-èsim haurà d'implementar $n - 2^j$ operacions. En total, el treball serà

Algorisme 7 Algorisme per a la suma prefixada.

```
     PPRESUMS(x[0 : n − 1])
1    from d = 0 to log n − 1 fer
2        per a tot 0 ≤ i ≤ n − 1 i i  mod 2^{d+1} = 0 parfer
3            x[i + 2^{d+1} − 1] := x[i + 2^d − 1] + x[i + 2^{d+1} − 1]
         fper
     ffer
4    a := x[n − 1]; x[n − 1] := 0
5    from d = log n − 1 to d = 0 fer
6        per a tot 0 ≤ i ≤ n − 1 i i  mod 2^{d+1} = 0 parfer
7            t := x[i + 2^d − 1]
8            x[i + 2^d − 1] := x[i + 2^{d+1} − 1]
9            x[i + 2^{d+1} − 1] := t + x[i + 2^{d+1} − 1]
         fper
     ffer
10   per a tot 0 ≤ i < n − 1 parfer
11       x[i]:=x[i+1]
     fper
12   x[n-1]:=a
```

$W(n) = \sum_{j=0}^{\log n-1} n - 2^j = O(n \log n)$. Per tant, si utilitzem $n/\log n$ processadors, tenim que l'algorisme 7 és òptim.

D'altra banda, en cap moment es necessita escriptura concurrent, ja que el valor a es pot emmagatzemar a la memòria local del processador $n-1$ i, pel que fa a la lectura concurrent, la línia 3 es pot desdoblar en dos passos, en el primer cada processador llegeix $x[i]$ i en el segon pas cada processador llegeix $x[i+2^j]$. Per tant, l'algorisme es pot implementar amb una PRAM-EREW.

Teorema 7.1. *L'algorisme* PPRESUMS *calcula les sumes prefixades de n nombres, i es pot implementar en una* PRAM-EREW *en $O(\log n)$ passos i treball de $O(n \log n)$.*

7.2 Cerca d'elements ordenats

Al capítol 6 vam introduir les PRAM utilitzant diferents variants del problema de la cerca sobre conjunts no ordenats. Què passa si el conjunt S sobre el qual volem cercar l'element y està ordenat? Recordem que utilitzant cerca binària o dicotòmica, aquest problema té una complexitat seqüencial de $O(\log n)$.

Sigui $S = \{x_1, \ldots, x_n\}$ un conjunt d'elements diferents amb $x_i < x_{i+1}$, i assumim que $x_0 = -\infty$ i que $x_{n+1} = +\infty$. L'algorisme 8 utilitza l'estratègia següent per a una PRAM amb $p+1$ processadors: escollirem p elements "equidistants" a S, això dividirà S en $p+1$

elements. Cada processador P_j s'encarrega de comparar un dels x_j equidistants amb y, per restringir el conjunt de cerca al segment entre els elements equidistants x_i i x_j tals que $x_i \leq y < x_j$. Tornem a repetir el procés restringint-nos al segment entre $\{x_i, x_j\}$. Continuem així fins que el segment a explorar tingui $p+1$ o menys elements. Si arribem a aquesta darrera situació, una comparació en paral·lel dels $p+1$ processadors decidirà el problema en temps $O(1)$.

Algorisme 8 Cerca en un vector classificat.

```
    PCERCADIC(x[0 : n + 1], y)
1   s := 0; l := 0; r := n + 1
2   mentre r − l > p fer
3       per a tot 0 ≤ j ≤ p parfer
4           a[j] := l + ⌊j ∗ (r − l)/p⌋
5           si x[a[j]] > y llavors c[j] := 1 altrament c[j] := 0 fsi
        fper
6       per a tot 0 ≤ j < p parfer
7           si c[j] ≠ c[j + 1] llavors r := a[j + 1];l := a[j] fsi
        fper
    fmentre
8   per a tot 0 ≤ j ≤ r − l parfer
9       si x[l + j] = y llavors s := l + j fsi
    fper
```

L'algorisme PCERCADIC té diferent comportament segons el nombre de processadors que utilitzem. Sigui p el nombre de processadors utilitzats. Els passos 1, 4, 5, 7 i 9 es poden implementar en un temps $O(1)$. La iteració i-èsima del **mentre** al pas 2 redueix la grandària de la cerca de $s_i = r - l$ fins a $s_{i+1} \leq (r-l)/(p+1) + p = (s_i/p + 1) + p$. Fent $s_0 = n + 1$, la solució de la recurrència és $s - i \leq (n + 1/p + 1) + p + 1$. El nombre d'iteracions és $O(\log(n+1)/\log(p+1))$ i cada iteració triga un temps $O(1)$. Per tant, el temps total és $O(\log(n+1)/\log(p+1))$ i el treball coincideix amb el cost.

Podem variar el nombre de processadors per tenir temps diferents: si utilitzem un nombre constant el temps és $O(\log n)$ i si prenem $p = n$ el temps serà constant. És fàcil veure que es necessita una PRAM-CREW. De fet, es pot demostrar que una fita inferior per al temps paral·lel de cerca en una PRAM-EREW és $\Omega(\log n - \log p)$.

Teorema 7.2. *L'algorisme* PCERCADIC *es pot implementar en una* PRAM-CREW *utilitzant p processadors i temps $O(\log(n+1)/\log(p+1))$.*

7.3 Fusió de seqüències ordenades

Siguin $A = (a_1, a_2, \cdots a_n)$ i $B = (b_1, b_2, \cdots, b_m)$ dues seqüències no decreixents tals que $a_i, b_i \in S$, on S és un conjunt linealment ordenat. Volem fusionar les dues seqüències en

una única seqüència ordenada $C = (c_1, \cdots, c_{n+m})$. Com recordareu, en seqüencial, la fita inferior necessària és $\Omega(n+m)$ i l'algorisme obvi seqüencial té complexitat $O(n+m)$ i per tant, és òptim.

Un algorisme senzill per fusionar dues seqüències es basa en el càlcul del rang d'un element en una seqüència. Sigui $X = (x_1, \cdots, x_t)$ una seqüència d'elements de S. Direm que $Z = (x_{i_1}, \ldots, x_{i_p})$ és una **subseqüència** de X si $1 \leq i_1 < i_2 < \cdots < i_p \leq t$. Sigui $x \in S$, el **rang** de x a X, $rg(x : X) = |\{x_i \in X | x_i \leq x\}|$.

Sense pèrdua de generalitat, podem assumir que no hi ha elements repetits a A i B. L'algorisme que presentem per resoldre el problema de la fusió determina el rang de cada element x a $A \cup B$ i, com que $rg(x : A \cup B) = rg(x : A) + rg(x : B)$, aleshores si coneixem el rang de cada element el podem col·locar a la seva posició correcta.

Algorisme 9 Fusió, basat en el càlcul del rang.

```
    FUSIÓ(x, a[1:n], b[1:m])
1   per a tot 1 ≤ i ≤ n parfer
2       j := i + rg(a[i], b)
3       c[j] := a[i]
    fper
4   per a tot 1 ≤ i ≤ m parfer
5       j := i + rg(b[i], a)
6       c[j] := b[i]
    fper
```

Per analitzar l'algorisme FUSIÓ cal tenir en compte que les seqüències A (B) estan ordenades i, per tant, podem calcular $rg(x : A)$ en paral·lel amb $O(\log n)$ passos i treball $O(\log n)$ utilitzant una petita variació de l'algorisme 8 de cerca. Així la complexitat de l'algorisme de fusió és $T(n) = O(\log n + \log m)$, i amb $W(n) = n \log n + m \log m$. En aquest cas el treball no és òptim i, per tant, no podem utilitzar el lemma de Brent per obtenir un algorisme òptim.

Per obtenir un algorisme òptim cal utilitzar una altra aproximació que minimitzi l'ús del càlcul del rang. Utilitzem la tècnica del **dividir i vèncer** per dissenyar l'algorisme paral·lel. Recordem que aquesta tècnica "trenca" el problema per resoldre en p subproblemes independents; perquè tingui sentit en el context dels algorismes paral·lels cal que, a més, els subproblemes siguin de grandària semblant. Després hem de resoldre els p problemes en paral·lel i combinar-ne els resultats per obtenir la solució final.

En el nostre cas, trencarem tant A com B en subseqüències més petites, $A_1 A_2 \ldots A_j \ldots A_k$ i $B_1 B_2 \ldots B_j \ldots B_r$, de tal manera que la solució es pot obtenir combinant les solucions dels parells corresponents. Ens interesa que la partició verifiqui dues propietats: primer les seqüències han de ser petites i, a mes, volem que combinar-les sigui poc costós en temps.

Per refinar més l'algorisme, comencem per escollir un particionat d'A i B. Dividirem B en k subseqüències consecutives, totes de la mateixa grandària r, i després dividirem A en $k = m/r$ seqüències consecutives, de manera que la fusió d'A i B ($fs(A, B)$) es pugui

calcular com la concatenació de les fusions $fs(A_1, B_1) \cdots fs(A_k, B_k)$. Per determinar la partició utilitzem l'algorisme 10, que calcula les posicions d'A, B a cada element; així: $A_i = A[j[i-1]+1 : j[i]]$ i $B_i = A[l[i-1]+1 : l[i]]$,

Algorisme 10 Divisió per a la fusió.

```
    PARTICIÓ(a[1 : n], b[1 : m], r)
1   k := m/r
2   j[0] := 0; j[k] := n
3   l[0] := 0; l[k] := m
4   per a tot 0 ≤ i ≤ k − 1 parfer
5         l[i] := i ∗ r
6         j[i] := rg(b[l[i]], a)
```

Quan prenem $r = \log m$, tenim:

Lema 7.1. *Sigui C la seqüència ordenada resultant de fusionar A i B amb longituds n i m. Aleshores, l'algorisme de divisió, parteix A i B en parelles (A_i, B_i) tals que $|B_i| = O(\log m)$, $\sum_{i=1}^{k(m)} |A_i| = n$ i $C = (C_0, C_1 \ldots)$, on C_i és la seqüència ordenada obtinguda fusionant A_i i B_i. A més, l'algorisme té una complexitat de $O(\log n)$ passos i un nombre total d'operacions de $O(n+m)$ amb una* PRAM-CREW.

Demostració. Per demostrar la correcció de l'algorisme, comencem per observar que tots els elements d'A_i i B_i són més grans que qualsevol element d'A_{i-1} i B_{i-1}. Els elements més petits d'A_i i B_i són $a_{j(i-1)+1}$ i $b_{i \log m+1}$, mentre que els elements més grans d'A_{i-1} i B_{i-1} són $a_{j(i)}$ i $b_{i \log m}$. Com que $rg(b_{i \log m} : A) = j(i)$, tenim $a_{j(i)} < b_{i \log m} < a_{j(i)+1}$, cosa que implica $b_{i \log m+1} > b_{i \log m} > a_{j(i)}$ i $a_{j(i)+1} > b_{i \log m}$. Per tant, tots els elements d'A_i i B_i són més grans que els elements d'A_{i-1} i B_{i-1}.

D'altra banda, l'algorisme triga $O(\log n)$ passos i el nombre d'operacions que realitza és $O(\log n(m/\log m)) = O(n+m)$. Observem que l'algorisme utilitza el càlcul del rang i que l'algorisme de cerca binària necessita una PRAM-CREW. □

L'algorisme 11 utilitza l'algorisme de partició i llavors obtenim una descomposició en subproblemes, si bé no tots són de mides semblants, però, per descomptat que tots els B_i tenen el mateix nombre r d'elements, encara que els A_i poden tenir qualsevol nombre d'elements. Quan la grandària d'A_i és gran el que farem és una partició de la parella (B_i, A_i) fixant la grandària de les subseqüències d'A_i. Després d'aquesta partició, prenen com a valor de r primer $\log m$ i després $\log n$, i tindrem una divisió del problema en subproblemes, de grandària com a màxim $\log n$, $\log m$, i farem la fusió seqüencial en temps $O(\log n + \log m)$.

Ens falta determinar quin és el cost d'aquest algorisme. Observem que en el pas 3 el temps és $\log |B_i| = \log \log m$ i el treball és $|A_i| + |B_i|$ si sumem el treball per tots els possibles parells tenim que, com a molt, es realitzaran $O(n+m)$ operacions en temps $O(\log \log m)$. En el pas 5 fem la fusió seqüencial de dues seqüències, en temps (i treball)

Algorisme 11 Fusió utilitzant dividir i vèncer.

```
    FUSIÓ2(A, B)
1   Partició(A, B, log |B|)
2   per a tot |A_i| > log |A| parfer
3       PARTICIÓ(B_i, A_i, log |A_i|)
    fper
4   per a tot parell (A_i, B_i) parfer
5       FUSIÓ-SEQÜENCIAL
    fper
```

$|A_i| + |B_i|$ per tant en total utilitzarem temps $O(\log n + \log m)$ i el treball total serà la suma del nombre d'elements a cada subseqüència. Com que el que tenim és una partició, això dóna treball $O(n + m)$. Per tant, hem demostrat el teorema següent:

Teorema 7.3. *Siguin A i B dues seqüències ordenades. Es poden fusionar de manera òptima en paral·lel en un temps $O(\log |A| + \log |B|)$ i utilitzant $O(|A| + |B|)$ operacions, en una* PRAM-CREW.

Existeixen a la bibliografia algorismes òptims més ràpids, de temps $O(\log \log n)$.

7.4 Classificació

Ja vam veure al capítol 6 com classificar un vector de n elements amb una PRAM-CRCW en temps constant però utilitzant $W(n) = O(n^2)$ operacions. Malauradament aquell algorisme no era òptim i, a més, utilitzava un protocol de concurrència massa fort. Vegem ara un algorisme per ordenar n elements que utilitza la fusió. Sigui $X = \{x_1, \ldots, x_n\}$ una seqüència que volem ordenar. Per dissenyar L'algorisme 12 seguirem una estratègia de dividir i vèncer. Dividirem X en dues parts, X_1, X_2, d'aproximadament la mateixa grandària. Ordenarem les dues meitats i fusionarem el resultat, a continuació, ho repetirem amb cada part. Per implementar aquesta idea en paral·lel, representarem tota la computació com un arbre binari, en què els elements de X estan distribuïts entre les fulles de l'arbre. Els nusos a nivell 1 representen el resultat de fusionar parells consecutius d'elements, donats pels fills del nus. Cada nus representa la seqüència obtinguda de fusionar les seqüències de fills i, per tant, l'arrel representarà tot X ordenat. Per (h, j) denotarem el nus j-èsim (a partir de l'esquerra) a alçada h de l'arbre binari que representa la computació. $L(h, j)$ representa la seqüència ordenada corresponent al nus (h, j). Observem que si considerem $|X| = n = 2^l$, l'alçada de l'arbre de computació és $O(\log n)$. També el nus (h, j) conté la llista ordenada dels elements que van de $X[2^h(j-1)+1]$ fins a $X[2^k j]$.

Analitzem ara l'algorisme CLASSIFICACIÓ. En el pas 1 hem de fer n operacions en total amb temps constant. El bucle en el pas 3 es repeteix $\log n$ cops, i cada cop utilitzant l'algo-

Algorisme 12 Classificació per fusió.

```
    CLASSIFICACIÓ(x[1 : n])
1   per a tot 1 ≤ j ≤ n parfer
2       l[0, j] := x[j]
    fper
3   per h = 1 fins log n fer
4       per 1 ≤ j ≤ n/(2^h) parfer
5           Fusió( L[h − 1, 2j − 1],L[h − 1, 2j] )
        fper
    fper
```

risme de fusió anterior; tenim un temps total $O(\log^2 n)$; el treball que es fa a cada pas és la grandària total de totes les seqüències que es fusionen, que dóna $W(n) = O(n \log n)$. Per tant, encara que la implementació que donem necessita n processadors, prenent un nombre de processadors adequat de $p = n/\log^2 n$ i utilitzant el lema de Brent podem obtenir un algorisme òptim. A causa del model de màquina utilitzat a la fusió, la implementació d'aquest algorisme requereix una PRAM-CREW.

Teorema 7.4. *Donada una seqüència de n elements, l'algorisme* CLASSIFICACIÓ *els classifica correctament en una* PRAM-CREW *i es pot implementar en temps $O(\log^2 n)$ i amb treball $O(n)$.*

7.5 Ordenació de n enters

Recordem que el problema d'ordenar n enters, cada un amb clau entre 0 i $m-1 = O(n^k)$, es pot fer en seqüencial amb temps $O(n)$, utilitzant ordenació pel mètode *radix*. En aquesta secció presentarem la versió paral·lela del *radix*.

Recordem que la classificació *radix* fa una iteració sobre els bits de les claus, començant pel més baix, i a cada pas executa una operació de divisió en què els elements que tenen aquest bit a 0 es col·loquen al començament i els que tenen un 1, al darrere, i aquesta divisió es repeteix amb el bit següent, fins que tinguem els elements classificats, cosa que es dóna quan tractem el darrer bit.

L'algorisme 13 conté la implementació paral·lela; per a cada element es calcula la nova posició a la llista i després es col·loca a la posició correcta. Per calcular la posició d'un 0 només necessitem conèixer quants elements amb aquest bit a 0 té davant. Aquest valor el podem calcular utilitzant una operació de suma prefixada al vector on tenim el bit que estem mirant complementat. Per calcular la posició d'un 1 s'utilitza una operació de suma sufixada al vector de bits corresponent. La suma sufixada és la mateixa operació que la suma prefixada, però utilitzant el vector invertit.

El bucle al pas 1 de l'algorisme PRADIX s'executa k cops, i a cada iteració la complexitat bé donada pel càlcul de les sumes prefixades, cosa que podem fer amb una PRAM-EREW

Algorisme 13 Classificació *radix*.

```
     PRADIX(a[1 : n], k)
1    per j = 0 fins k fer
2        per a tot 1 ≤ i ≤ n parfer
3            b[i] := bit j de a[i]
4            c[i] := 1 − b[i]
         fper
5        PPRESUMS(c, d, n)
6        per a tot 1 < i ≤ n parfer
7            d[i] := d[i] − 1
         fper
8        d[0] := 0
9        PSUFSUMS(b, e, n)
10       per a tot 1 ≤ i ≤ n parfer
11           si b[i]
12               llavors l[i] := d[i]
13               altrament l[i] := e[i]
             fsi
             a[l[i]] := a[i]
         fper
     fper
```

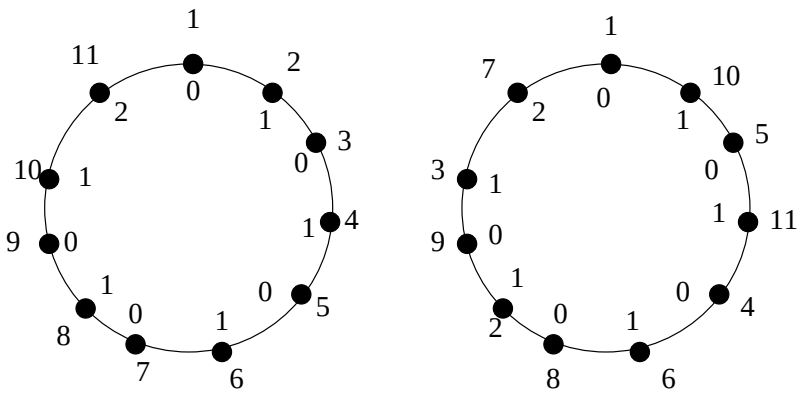
amb treball $O(n)$ i temps $O(\log n)$. Prenent $k = \log n$ tenim que el temps total és $T(n) = O(\log^2 n)$ i el treball és $W(n) = O(n \log n)$. Així tenim el teorema següent.

Teorema 7.5. *L'algorisme* PRADIX *classifica correctament n enters, i es pot implementar a una* PRAM-EREW *amb treball $O(n \log n)$ i temps $O(\log n)$.*

7.6 3-Coloració d'un cicle

Vegem ara un problema en què la tècnica seqüencial no és directament paral·lelitzable. El problema consisteix a colorir amb 3 colors un cicle de n vèrtexs. Donat un graf dirigit $G = (V, A)$, una *3-coloració* és una aplicació $c : V \to \{0, 1, 2\}$ tal que si $(i, j) \in A$ aleshores $c(i) \neq c(j)$. L'algorisme obvi seqüencial assigna colors 0 i 1 alternativament a cada vèrtex del cicle i, si cal, utilitza el color 2 per al darrer vèrtex de G (en cas que n sigui parell). Aquest algorisme té complexitat $O(n)$ i és òptim. Si tots els vèrtexs del cicle estiguessin enumerats consecutivament (vegeu la figura 7.1(a)), seria fàcil dissenyar un algorisme paral·lel: assignar 1, en paral·lel, als parells, després assignar 0 als senars, en paral·lel, i finalment mirar si el color del vèrtex n és 0 i, si ho é,s canviar-lo a 2.

En general, l'enumeració del cicle no té perquè ser consecutiva (figura 7.1(b)) i aleshores l'algorisme seqüencial no serveix per derivar un algorisme paral·lel, ja que tots els vèrtexs



10	9	7	6	11	8	1	2	3	5	4

c

Figura 7.1: Exemples de cicle per 3-coloració.

són semblants. El cicle G tindrà com a estructura de dades una taula T de longitud n tal que $T[i] = j$ sii $(i, j) \in A$ (a la figura 7.1(c) tenim la taula per al cicle de 7.1(b)).

Presentarem un algorisme paral·lel, l'algorisme 14, basat en el trencament de la simetria del cicle per tal de poder classificar d'alguna manera els vèrtexs de G en classes, de manera que puguem assignar en paral·lel el mateix color a tots els elements de la mateixa classe així, en finalitzar tindrem només tres classes.

Donada l'estructura de dades per al cicle, mitjançant una taula T és molt fàcil obtenir el predecessor d'un vèrtex i, si mirem la posició que ocupa i a T. L'algorisme utilitza la representació binària de les etiquetes que anirem assignant a cada element. Utilitzarem la notació següent: Donats enters i i t, sigui $i_t i_{t-1} \cdots i_k \cdots i_1 i_0$ l'expansió binària de i amb t bits; per tant, i_k denota el k-*èsim* bit de i. Per exemple, el 21_3 és l'1 subratllat a 110$\underline{1}$01.

Partint d'un cicle $G = (V, A), |V| = n$ amb coloració inicial $c[i] = i$ amb n colors, l'algorisme a cada pas obté una altra coloració amb menys colors.

Primer veurem que l'algorisme REDUCE calcula una coloració. Com que c és una coloració, l'índex k ha d'existir (dos vèrtexs consecutius no poden tenir el mateix color). Si existeix $(i, j) \in A$ amb $2k + c(i)_k = 2l + c(j)_l$, tindrem que $k = l$ i, a més, $c(i)_k = c(j)_l$ però per construcció a la posició $l = k$ han de discrepar. Per tant, si c és una coloració al començament de l'algorisme 14, c també és una coloració al final. L'algorisme funciona en $O(1)$ passos paral·lels i $W(n) = O(n \log t)$ operacions. A més, el nombre de colors en la nova coloració és $2 \log t + 1$.

Algorisme 14 Reducció exponencial de colors al cicle.

```
REDUCE(t[1 : n], c[1 : n], )
per a tot 1 ≤ i ≤ n parfer
    sigui k l'últim bit significatiu
      on c[i] i c[t[i]] discrepen
    c[i] := 2 * k + c[i]_k
fper
```

Si comencem amb n colors, per aplicació reiterada de l'algorisme 14 podem arribar a tenir un nombre constant de colors. Encara que no podem assegurar que arribarem a 3 colors, hem de tenir en compte que per reduir el nombre de colors cal que $t > 2\log t + 1$, i això vol dir que $t > 3$. A cada pas passem de t colors a $O(\log t)$ colors, és a dir, tenim una disminució exponencial en el nombre de colors. Aquest procés el podem repetir mentre $t > \log t + 1$. L'algorisme donarà una coloració c amb 5 colors o menys. Per avaluar el nombre d'iteracions necessàries fins arribar a 5 colors, observem que en la primera iteració reduïm el nombre de colors a $O(\log n)$, en la segona iteració reduïm fins a $O(\log\log n)$; per tant, després de $\log^* n$[1]iteracions el nombre de colors serà, com a màxim, 5.

Un cop hem aplicat l'algorisme 14 $O(\log^* n)$ cops, tenim com a màxim $\{0,1,2,3,4\}$ colors i podem reduir el nombre de colors amb 2 noves iteracions, com es veu a l'algorisme 15, per eliminar primer el 4 i després el 3 de la coloració. Aquest algorisme té una complexitat

Algorisme 15 Reducció lineal de colors al cicle.

```
    REDUCE3(t[1 : n], c[1 : n])
1   per i = 4 fins 3 fer
2       per a tot 1 ≤ j ≤ n parfer
3           si c[j] = i llavors c[j] := k
4                 on k ∈ {0, . . . , i}
                    és el color més petit possible
            fsi
        fper
    fper
```

de $T(n) = O(1)$ amb $W(n) = O(n)$. Posant junts els algorismes 14 i 15, tenim:

Teorema 7.6. *L'aplicació seguida dels algorismes* REDUCE *i* REDUCE3 *ens permet assignar 3 colors a un cicle amb n vèrtexs en* $O(\log^* n)$ *passos amb* $O(n\log^* n)$ *operacions; l'algorisme es pot implementar utilitzant una* PRAM-EREW.

Per tant, tenim una manera molt ràpida però no òptima d'assignar 3 colors a un cicle. Modifiquem l'algorisme de manera que sigui més lent i òptim. L'estratègia de l'algorisme 16

[1]Recordem que $\log^* n = \min\{i \mid \log^i n \leq 1\}$.

consisteix a utilitzar l'algorisme 14 un cop fins a arribar a $O(\log n)$ colors, després classificar els vèrtexs per colors, de manera que puguem assignar processadors a tots els vèrtexs amb el mateix color, i després utilitzar un algorisme semblant a l'algorisme 15 per reduir el nombre de colors.

Algorisme 16 3-coloració del cicle.

```
    3-COLORS(t[1 : n])
1   per a tot 1 ≤ i ≤ n parfer
2       c[i] := i
    fper
3   REDUCE(t,c)
4   Classificar els vèrtexs per colors
5   per i = 3 fins 2 log n fer
6       per a tot j amb c[j] = i parfer
7           c[j] és el color més petit d'entre {0, 1, 2}
            que sigui possible
        fper
    fper
```

Teorema 7.7. *L'aplicació de l'algorisme* 3-COLORS *ens permet assignar 3 colors a un cicle amb n vèrtexs en $O(\log n)$ passos amb $O(n)$ operacions (per tant, és òptim) utilitzant una* PRAM-EREW.

Demostració. La correcció és òbvia. Respecte a la complexitat, els passos 1–3 tenen $T(n) = O(1)$ i $W(n) = O(n)$. Classificar n enters amb claus entre 0 i $\log n$ té una complexitat paral·lela de $T(n) = O(\log n)$ amb $O(n)$ operacions (exercici 7 d'aquest capítol). El darrer bucle té una complexitat de $T(n) = O(\log n)$ amb $W(n) = O(\sum n_i) =)(n)$ operacions, on n_i és el nombre de vèrtexs amb color i. L'algorisme 16 pot ser implementat amb una PRAM-EREW. □

7.7 Referències bibliogràfiques

Per consultar implementacions de la fusió paral·lela en $O(\log\log n)$, vegeu el capítol 4 del llibre [JáJ92] o [Val75]. Existeix un algorisme d'ordenació amb una PRAM-EREW que funciona en temps $O(\log n)$ amb $W(n) = O(n)$ [Col88, JáJ92]. Tant l'algorisme 8 com la demostració de la fita inferior es poden trobar a [Sni85].

Exercicis

7:1 Escriviu un algorisme òptim que troba el màxim de n elements. Quin tipus de PRAM es necessita?

7:2 Donats n elements emmagatzemats en un vector A, conjuntament amb un vector L tal que $L(i) \in \{0, \ldots, k\}$ representa l'etiqueta d'un element $A(i)$ (on k és una constant), doneu un algorisme òptim $O(\log n)$ sobre una PRAM-EREW que emmagatzema tots els elements d'A etiquetats 1, al començament d'A, preservant l'ordre relatiu. A continuació emmagatzema els elements amb 2, etc.

7:3 Donat un vector $A = (a_1, \cdots, a_n)$ on cada a_j pertany a un conjunt linealment ordenat S, i donats dos elements $x, y \in S$, emmagatzemeu tots els elements a_i de A, tals que $x \leq a_i \leq y$, a posicions de memòria consecutives. L'algorisme realitzarà $O(\log n)$ passos i $O(n)$ operacions. Especifiqueu quin model de PRAM es necessita.

7:4 Donada una llista de n elements, calculeu el rang de cada element amb $O(\log n)$ passos. (Recordeu que el rang d'un element és la distància de l'element a la cua de la llista.)

7:5 Donat un vector d'objectes de color, dissenyeu un algorisme (PRAM) que trobi tots els objectes d'un mateix color. Quins seran el temps i el treball? Quin tipus de PRAM utilitzarà?

7:6 Escriviu acuradament l'algorisme de la fusió de dues llistes en temps $O(\log n)$ i amb $O(n)$ operacions, utilitzant una PRAM-CREW.

7:7 Donats n enters amb valors entre 0 i $\log n$, dissenyeu un algorisme per la PRAM-EREW que classifiqui els enters i tingui cost $O(n)$.

7:8 Doneu un algorisme paral·lel per 2-colorir els vèrtexs d'un arbre. Quins seran el temps i el nombre d'operacions? Quin tipus de PRAM fareu servir?

7:9 Donat un graf bipartit $B = (V_1, V_2, E)$, amb $|V_1| = |V_2| = n$ considerem que per a qualsevol $k, 1 \leq k \leq n$, M_k denota el conjunt de *matchings* amb cardinalitat k. Recordem que un *matching* és un conjunt d'arestes tal que dues arestes del conjunt mai no tenen un vèrtex en comú. Representarem un *matching* mitjançant una taula en la qual és diu per a cada element de V_1 l'element corresponent a V_2 o un zero, si no es correspon amb cap. Expliqueu com resoldríeu els problemes següents:

Donats *matching* $m_1 \in M_i$,i $m_2 \in M_j$, sigui H el graf induït per $m_1 \cup m_2$.

1. Per cada vèrtex computeu-ne el grau a H.
2. Computeu els components connexos del subgraf H.
3. Computeu la grandària del *matching* més gran a cada component connex.

7:10 Donat un univers $U = \{1, 2, \ldots, n\}$ i un subconjunt $I \subset U$, dues maneres de representar I sobre l'univers U són les següents:

1. *Vector de bits.* Definim una taula $A : \{1, \ldots, n\} \to \{0, 1\}$ tal que, per a tot $i = 1, \ldots, n$, $A[i] = 1$, si i només si $i \in I$.
2. *Representació amb punters.* Definim un triple (Primer, Últim, Punter) on, si $I = \{i_1, \ldots, i_m\}$, tenim que Primer $= i_1$, Últim $= i_m$, i on Punter és una taula amb dimensió n tal que $\forall i_j, 1 \leq j \leq m-1$, Punter$[i_j] = i_{j+1}$.

Dissenyeu una algorisme per la PRAM tal que, donada una representació d'un conjunt I amb vector de bits, computi la representació amb punters de I.

Capítol 8

Algorismes PRAM per a llistes encadenades

Fins ara, hem assumit que les dades venien donades a la memòria comuna en forma d'una taula o vector, és a dir, en posicions consecutives. Per tant, un cop els processadors coneixen quina és la direcció de començament de la taula és fàcil conèixer l'element que ocupa un lloc donat i fer tota mena de particions de la taula. En molts casos, l'estructura subjacent en les dades d'un problema no és una taula sinó una llista encadenada amb punters; cada element té un punter que diu la posició a memòria de l'element següent. Així, doncs, el processador assignat a una dada només pot accedir-hi a l'element següent a la llista. En aquest capítol desenvoluparem tècniques bàsiques per tractar aquestes estructures de dades.

8.1 Salt de Punter (*pointer jumping*)

Considerem que en lloc de tenir n elements emmagatzemats en posicions consecutives de memòria, tenim una llista d'elements amb punters; cada element x de la llista té un punter a l'element següent, i tenim en una posició de memòria un punter al primer element, com a la figura 8.1. L'existència d'aquesta informació sobre el primer element de la llista no suposa cap restricció ja que, si no hi fos, es podria calcular en temps constant (vegeu l'exercici **8:1**). A més assumirem que el darrer element apunta a ell mateix. Observem que a la PRAM no cal accedir de manera seqüèncial als elements de la llista; la màquina assignarà un processador a cadascun dels elements en la llista, encara que l'ordre dels processadors no té cap relació amb la posició relativa de l'element a la llista. A la figura 8.1 tenim una assignació possible de processadors a elements.

La tècnica del **salt de punter** (pointer jumping) consisteix a repetir la computació

$$s(x) = s(s(x)).$$

Iteracions succesives d'aquesta instrucció donen com a resultat que els n elements de la llista apunten cap al darrer element de la llista (vegeu la figura 8.2). Si tenim n processadors,

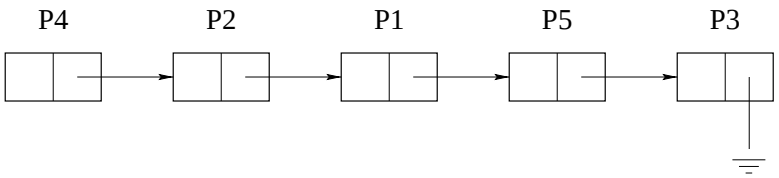


Figura 8.1: Una llista encadenada

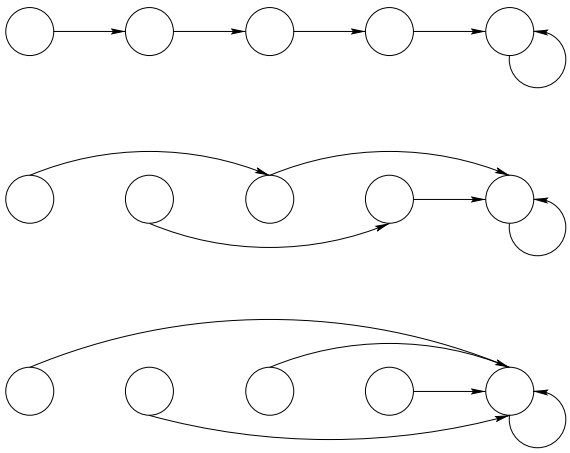


Figura 8.2: Aplicació de salt de punter.

el fet de realitzar un salt de punter en paral·lel es pot implementar en $O(1)$ passos. Per tant, en $O(\log n)$ passos tots els punters dels elements de la llista apuntaran cap al darrer element. Però necessitarem una PRAM-CREW, ja que dos o més processadors poden haver de que llegir l'adreça de l'últim processador al mateix pas. Per utilitzar una PRAM-EREW, a cada iteració, abans de fer el salt, el penúltim processador copia l'adreça de l'últim, trenca la llista en dues llistes i evita la necessitat de lectura concurrent (vegeu la figura 8.3 i l'exercici **8:4**).

A partir d'aquesta representació de dades, és fàcil calcular en paral·lel amb temps constant una taula s de $n+1$ elements, associada a la llista en la qual a cada posició i tenim l'identificador del processador assignat a l'element següent de l'assignat al processador i-èsim, a la posició $s[0]$ tindrem l'identificador del processador assignat al cap de llista i per al darrer element tindrem $s[i] := i$. A la figura 8.1 també es donan els continguts de la taula s corresponent. Aquest tipus d'informació es pot obtenir en temps constant amb $n+1$ processadors (vegeu l'exercici **8:2**). Finalment observem que de fet totes aquestes operacions de salt de punter es poden fer utilitzant la taula associada i que, per tant, es poden implementar sense que es produeixi cap modificació en l'estructura de dades original.

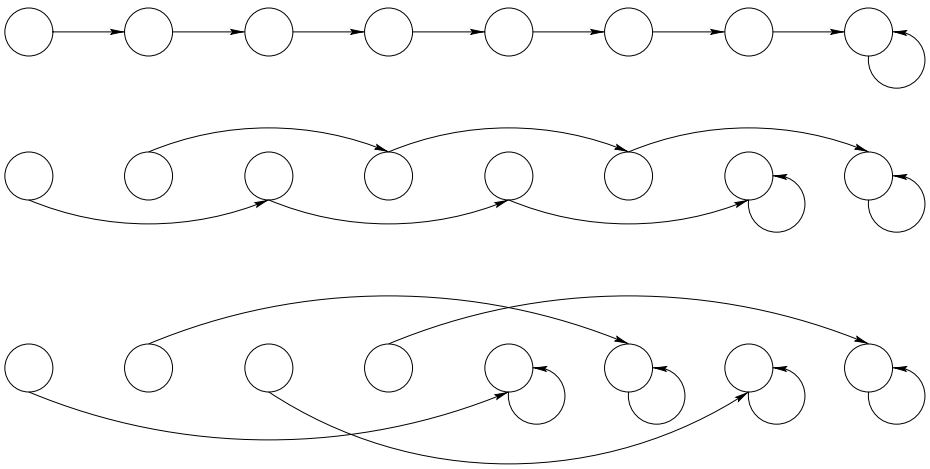


Figura 8.3: Salt del punter a la PRAM-EREW.

8.2 Enumeració d'una llista

Considerem una llista L de n elements amb l'ordre especificat pels punters, i sigui s la taula associada. Considerem que a més disposem d'una taula p en la qual a cada posició $i \in L$ té un punter $p(i)$ cap a l'element previ de i a L. Aquest tipus de taula es pot construir en temps constant amb n processadors (vegeu l'exercici **8:3**).

El problema de **l'enumeració de la llista** (list ranking) consisteix a determinar la distància (nombre d'elements) $d(i)$ de cada nus i al final de la llista L. En seqüèncial existeix un algorisme òptim lineal. El primer algorisme que veurem, l'algorisme 17, utilitza el salt de punter combinat amb una taula auxiliar d on anirem acumulant les enumeracions parcials; la taula d s'inicialitza a 1 per a tots els elements menys per a l'últim que, li assignem el valor 0. L'algorisme ENUM-SALT té una complexitat de $T(n) = O(\log n)$ amb $W(n) = O(n \log n)$ operacions, i es pot implementar amb una PRAM-EREW. Però no és òptim.

Algorisme 17 Enumeració utilitzant només salt de punter.

```
    ENUM-SALT(L, s, p)
1   per a tot 1 ≤ i ≤ n parfer
2       si s[i] ≠ i llavors d[i] := 1 altrament d[i] := 0 fsi
3   per j = 1 fins log n fer
4       per a tot 1 ≤ i ≤ n parfer
5           d[i] := d[i] + d[s[i]]
6           s[i] := s[s[i]]
        fper
    fper
```

Per disenyar un algorisme que enumeri una llista (encadenada) de manera òptima, utilitzarem l'estratègia següent:

1. Comprimir la llista L fins que tingui $O(n/\log n)$ nusos.

2. Aplicar l'algorisme 17 a la nova llista L'.

3. Restituir la llista original i enumerar els nusos eliminats al pas 1.

L'apartat més difícil és 1. L'apartat 2 ja hem vist com fer-lo, i l'apartat 3 és el procés contrari a l'apartat 1.

Comencem per definir el que és un **conjunt independent** en una llista encadenada. Un conjunt I s'anomena **independent** si, per a tot $i \in I$, es compleix que $s(i) \notin I$. Aleshores, si l'eliminació d'un nus $i \in I$ es realitza amb la instrucció $s(p(i)) = s(i)$, el fet que I sigui independent implica que aquesta operació pot ser aplicada en paral·lel sense conflictes a tots els elements de I. Evidentment, amb vista al pas 3, hem d'emmagatzemar els elements eliminats en una taula E. Donada una llista L amb $|L| = n$ amb punters s cap endavant i p cap endarrere, un conjunt independent I de nodes que no inclouen el primer i el darrer element de la llista, i una quantitat $d(i)$ associada a cada i, l'algorisme 18 dóna una llista L' amb els nusos de I eliminats i amb les quantitats $d(i)$ convenientment incrementades. Hem suposat que els elements de I es donen amb una taula I amb valor 1 quan l'element és a I i 0 quan no hi és.

Algorisme 18 Compressió d'una llista, donat un conjunt independent.

```
     COMPRESS(L, s, p, d, I)
1    PPRESUMS(I, m)
2    per a tot i ∈ I parfer
3        E[m[i]] = (i, s[i], p[i], d[i])
4        d[p[i]] = d[p[i]] + d[i]
5        s[p[i]] = s[i]
6        p[s[i]] = p[i]
     fper
```

Analitzarem ara l'algorisme COMPRESS. En utilitzar suma prefixada sobre el conjunt I el que fem és assignar un nombre diferent (però consecutiu) a cada element de I; aquest valor ens servirà per emmagatzemar la informació necessària i després poder reconstruir la llista. Podem realitzar el pas 1 en paral·lel amb una complexitat de $T(n) = O(\log n)$ i $O(n)$ operacions. Els passos 3–6 utilitzen temps $O(1)$ i $O(n)$ operacions. Únicament necessitem una PRAM-EREW. A la figura 8.4 es dóna un exemple d'aplicació de l'algorisme.

Hem de dissenyar una manera ràpida de trobar I. Per això utilitzem l'algorisme 16 d'acolorir amb 3 colors un cicle, però aplicant-lo al vector s. Primer cal transformar la llista en un cicle, només hem de canviar el valor del següent a l'ultim pel valor 0, (recordem que la posició 0 apunta cap al primer element). Sigui k un enter constant donat, i sigui

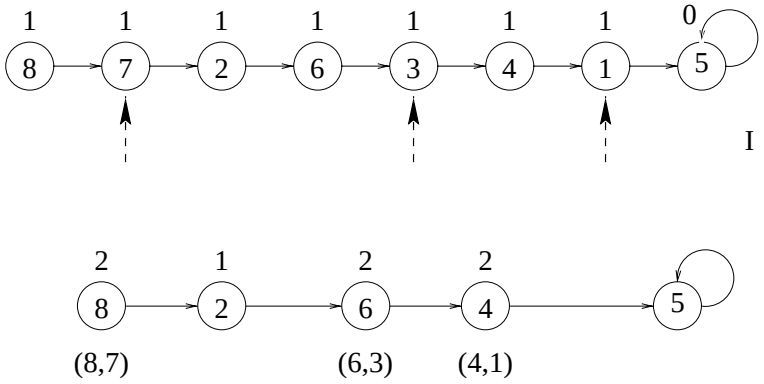


Figura 8.4: Exemple d'aplicació de l'algorisme COMPRESS.

c una k-coloració de L, un nus x de L s'anomena un *mínim local* si el color de x és més petit que els colors del seu predecessor i del seu successor. El resultat següent ens fita la grandària del subconjunt de mínims locals.

Teorema 8.1. *Donada una k-coloració de la llista L ($|L| = n$), el conjunt I de mínims locals és un conjunt independent amb grandària $\Omega(n/k)$.*

Demostració. Siguin x i y dos mínims locals a L, tals que no existeix cap altre mínim local entre ells. Aleshores,

- x i y no poden ser adjacents,
- els colors dels nusos entre x i y formen una seqüència creixent seguida d'una seqüència decreixent.

Per tant, el nombre de nusos entre x i y és, com a màxim, $2k-3$. Si L té n elements, $|I| \geq n/2k - 1 = \Omega(n/k)$. □

Amb una PRAM-EREW, podem decidir si un nus és un mínim local comparant amb el veí esquerre i amb el dret (2 passos). En paral·lel podrem identificar I en $O(1)$ passos i amb $O(n)$ operacions. Per tant, l'algorisme 19 d'identificació de I a L es pot implementar amb una PRAM-EREW amb $T(n) = O(\log n)$ i $W(n) = O(n)$.

Executar l'algorisme INDEP sobre L, donarà com a resultat un conjunt independent de grandària $O(n/3)$. Com que volem un conjunt independent de grandària $O(n/\log n)$, hem d'aplicar z cops l'algorisme 16, on $n/\log n = n/3^z$; per tant, $z = O(\log\log n)$.

Posant-ho tot junt, donada una llista L com la descrita, i amb longitud n, l'algorisme 18 ens associa amb cada nus i la distància $d(i)$ al darrer element de L.

Teorema 8.2. *L'algorisme* ENUMERA2 *enumera L en $O(\log n \log\log n)$ passos amb $O(n)$ operacions i pot ser implementat amb una* PRAM-EREW*; per tant, és òptim.*

Algorisme 19 Càlcul d'un conjunt independent per a una llista.

```
    INDEP(s, I)
1   COLORACIÓ(s, c)
2   per a tot 1 ≤ i ≤ n parfer
3       si c[i] ≤ c[s[i]] i c[i] ≤ c[p[i]]
4           llavors I[i] := 1
5           altrament I[i] := 0
        fsi
    fper
```

Algorisme 20 Enumeració òptima.

```
    ENUMERA2(L, s, p)
1   per a tot 1 ≤ i ≤ n − 1 parfer
2       d[i] := 1
    fper
3   d[n] := 0
4   ll[0] := n; k := 0
5   mentre ll[k] > n/ log n fer
6       COMPRESSIÓ(L, I[k])
7       k := k + 1; ll[k] := ll[k − 1] − |I[k − 1]|
    fmentre
8   ENUMERA(L, s, p)
9   per j=k-1 fins 0 fer
10      EXPANSIÓ(L, I[k])
    fper
```

Demostració. La demostració de la correcció es deixa com a exercici **8:6**. Els passos 1–4 tenen complexitat, temps $O(1)$ i treball $O(n)$. Utilitzant el mateix argument que a la demostració del teorema 8.1, al pas 5, el nombre d'iteracions necessàries per reduir la grandària de L fins a $O(n/\log n)$ és $O(\log\log n)$. Com que l'algorisme 18 té una complexitat de $O(\log n)$ passos i $O(n)$ operacions, el pas 6 es pot implementar amb aquests recursos. Per tant, el temps total del bucle en el pas 5 és $O(\log n \log\log n)$ i el nombre d'operacions $O(\sum_k (2/3)^k n) = O(n)$. Per al pas 8, $T(n) = O(\log n), W(n) = O(n)$. Tenint en compte els resultats de l'exercici **8:5** i les fites obtingudes per l'algorisme 18, el teorema està demostrat. El model de PRAM és conseqüència del fet que els algorismes utilitzats no necessiten concurrència d'escriptura o lectura. □

8.3 Referències bibliogràfiques

El primer algorisme òptim per enumerar una llista amb punters amb temps $O(\log n)$ va ser donat a [CV86]. Un algorisme òptim més senzill amb temps $O(\log n)$ i per PRAM-EREW ve donat a la secció 3.2 del [JáJ92].

Exercicis

8:1 Dissenyeu un algorisme tal que, donada una llista amb punters, calculi les posicions de memòria del primer i el darrer elements. L'algorisme ha de trigar un temps constant.

8:2 Dissenyeu un algorisme tal que, donada una llista amb punters, calculi una taula s on a la posició i té l'identificador del processador assignat a l'element següent de l'element assignat al processador i. L'algorisme ha de trigar temps constant.

8:3 Dissenyeu un algorisme tal que, donada una llista amb punters, calculi una taula p on a la posició i es té l'identificador del processador assignat a l'element anterior de l'element assignat al processador i. L'algorisme ha de trigar temps constant.

8:4 Escriviu amb cura l'algorisme per implementar el salt de punter amb una EREW-PRAM.

8:5 Detalleu el pas 9 a l'algorisme 20. Demostreu la correcció i les fites de temps i nombre operacions.

8:6 Demostreu la correcció de l'algorisme 20.

8:7 Donada una llista encadenada amb punters, doneu un algorisme PRAM per trobar el màxim de la llista. És òptim? Quin tipus de PRAM necessitem?

8:8 Doneu un algorisme òptim per computar la distància del començament d'una llista fins a cada element.

8:9 Reescriviu l'algorisme 19 assumint que a la definició de I es poden incloure el darrer o el primer element.

8:10 Doneu un algorisme òptim per realitzar una suma prefixada sobre una llista d'elements que ve donada com a llista encadenada amb punters. Quin tipus de PRAM necessitem?

8:11 Doneu un algorisme per trobar les arrels d'un bosc dirigit. Recordem que un arbre arrelat dirigit és un arbre dirigit que, a la versió no dirigida, és un arbre arrelat amb un camí de cada vèrtex fins a l'arrel. Donat un bosc $\mathcal{B}$, considerem punters tals que $s(i) = j$, si (i, j) és un arc a $\mathcal{B}$. Si i és una arrel, aleshores tindrem $s(i) = i$. El problema consisteix a determinar per a cada j l'arrel $r(j)$ de l'arbre que conté j. Quina serà la complexitat del problema? Quin tipus de PRAM serà necessària?

Capítol 9

Algorismes PRAM per a arbres

La representació canònica dels arbres amb punters és ben coneguda i es pot trobar a qualsevol llibre clàssic d'estructures de dades, per exemple, [CLR89]. Nosaltres utilitzem una representació diferent, amb llistes encadenades (vegeu la figura 9.1). En certa manera aquesta representació destrueix l'estructura jerarquitzada d'arbre, però això és el que volem amb vista a dissenyar algorismes paral·lels eficients (ja que el fet que la informació d'un nus depengui de la informació dels pares o fills pot tenir cost n per a certs tipus d'arbres). No cal insistir sobre la importància que tenen els arbres en informàtica. En aquest capítol presentem dues tècniques bàsiques per dissenyar algorismes paral·lels sobre arbres. La tècnica basada en un **recorregut eulerià** d'un arbre, que s'utilitza per calcular diferents paràmetres associats a un arbre, i la tècnica de **contracció** d'un arbre, que s'utilitza per a l'avaluació paral·lela d'expressions aritmètiques que es poden representar mitjançant un arbre de càlcul.

9.1 Recorregut eulerià d'un arbre

Sigui $T = (V, A)$ un arbre i sigui $T' = (V, A')$ el digraf dirigit obtingut a partir de T per substitució de cada aresta $\{u, v\} \in A$ per dues arestes dirigides $(u, v) \in A'$ i $(v, u) \in A'$. Observem que a T' la cardinalitat de sortida de cada vèrtex és igual a la cardinalitat d'entrada; per tant, existeix un recorregut eulerià de T', és a dir, un circuit que recorre totes les arestes de A' i passa per cada aresta exactament una vegada. A la figura 9.1 tenim el digraf T' corresponent a l'arbre T. Per produir un recorregut eulerià, escollim un vèrtex, "tallem" per aquest vèrtex i fem tot el recorregut del graf. Per exemple, si a la figura 9.1 escollim el vèrtex 2, un possible recorregut eulerià és: (2,5) (5,9) (9,5) (5,7) (7,3) (3,7) (7,6) (6,4) (4,6) (6,8) (8,6) (6,7) (7,1) (1,7) (7,5) (5,2). Evidentment, l'algorisme seqüencial obvi té complexitat $O(n)$ i és òptim.

Sigui d_v el grau d'un vèrtex v a T, i $L[v] =< u_0, u_1, \ldots, u_{d-1} >$ la llista dels vèrtexs adjacents a v. Al graf T' la llista $L[v]$ representa les arestes $(v, u_0) \cdots (v, u_{d-1})$ que surten de v.

Un recorregut eulerià a T' pot ser definit especificant el successor de cada aresta en el

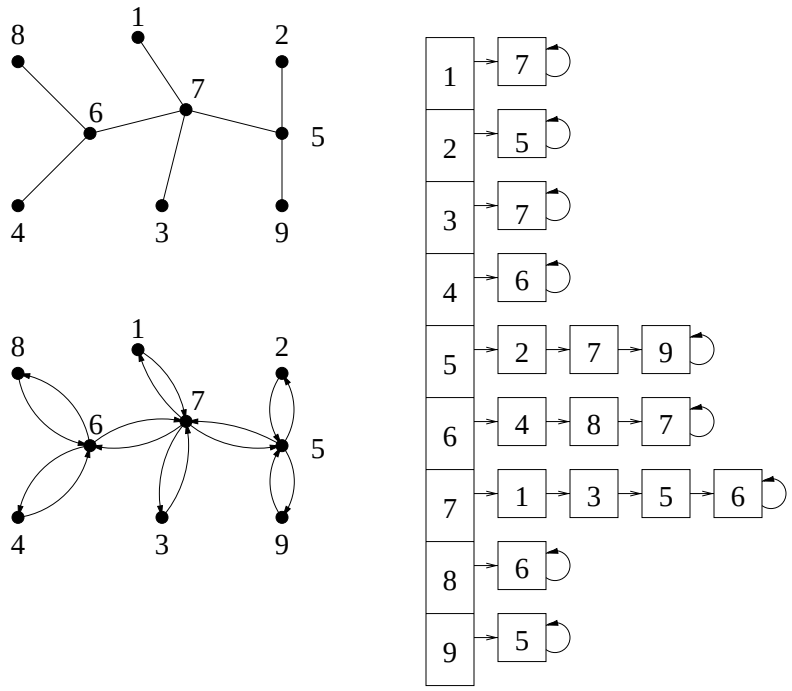


Figura 9.1: Arbre amb les arestes dirigides, amb la seva estructura de dades.

recorregut, per exemple amb la funció $s(u_i, v) = (v, u_{(i+1) \bmod d_v})$, per a $0 \leq i < d_v$. No és difícil verificar que la funció s especifica un recorregut eulerià sobre T', és a dir, demostrar que s defineix un cicle (i no un conjunt disjunt d'arcs).

A l'exemple de la figura 9.1, per calcular $s(6,7)$ busquem el vèrtex $v = 7$ al vector de l'estructura de dades i, com que $d_7 = 4$ i $L[7] = \langle 1, 3, 5, 6 \rangle$ $6 = u_3$, l'aresta $(v, u_{4 \bmod 4})$ serà $(7,1)$. Observem que la computació de $s(u_i, v)$ esdevé una computació local, on únicament necessitem conèixer l'arc (u_i, v) i el seu successor $u_{(i+1) \bmod d}$ a $L[v]$. A més, vegem que l'aresta (u, v) ès representada per l'ocurrència de v a la llista de u; així cal tenir una connexió entre el component que té v a la llista de u i el component que té u a la llista de v. Amb aquesta estructura de dades el successor és fàcil de trobar, excepte si u_i és el darrer element de $L[v]$. Per obtenir l'arc $s(u_i, v)$ de manera ràpida, podem afegir punters addicionals del vèrtex u_i a $L[v]$ fins al vèrtex v a $L[u_i]$, i viceversa. A més, afegirem un punter del darrer element de $L[v]$ cap al primer, fent $L[v]$ una llista **circular**.

Anomenarem aquesta nova estructura de dades **estructura de dades estesa** , i d'ara endavant partirem sempre d'aquesta estructura de dades estesa. A la figura 9.2 ve l'estructura de dades estesa per a l'arbre de l'exemple de la figura 9.1. Passar de l'estructura de dades per llista d'adjacència a l'estructura de dades estesa té un cost paral·lel de $O(\log n)$ passos amb $O(n)$ operacions i es pot implementar amb una PRAM-EREW; aquest resultat es deixa com a exercici **9:1**.

Si considerem un arbre T que ve donat per una estructura de dades estesa de manera que hi ha un processador per a cada nus de l'estructura, l'algorisme 21 ens dóna com a

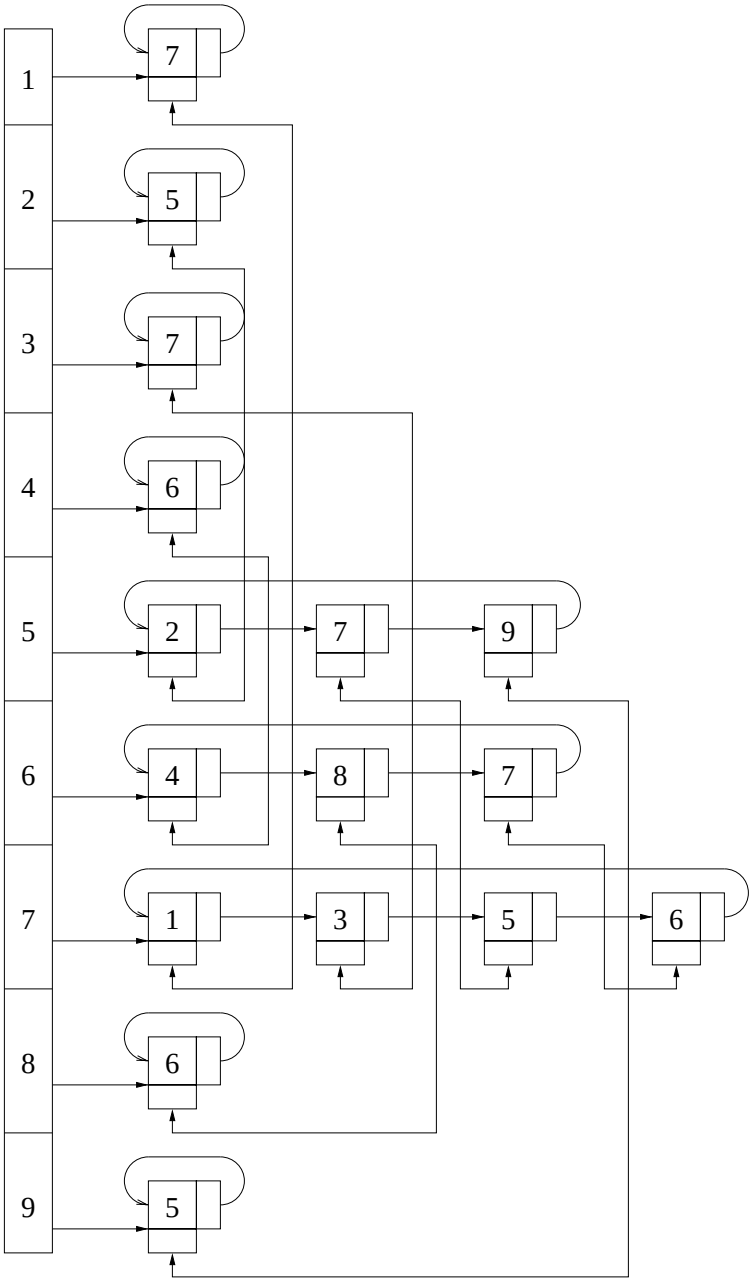


Figura 9.2: Estructura de dades estesa

resultat l'estructura de dades estesa, a la qual s'afegirà un nou punter per emmagatzemar el circuit eulerià de T. A la figura 9.3 es dóna el resultat de l'aplicació de l'algorisme 21 a

Algorisme 21 Circuit eulerià

```
    EULER(T, E)
1   per a tot aresta e = (u, v) parfer
2       E[e] := s(e)
    fper
```

l'exemple de la figura 9.1.

Per a la implementació concreta d'aquest algorisme considerem que tenim un processador per a cada nus de l'estructura estesa i disposem d'un camp auxiliar on seran els punters corresponents al circuit eulerià. Si un processador P s'encarrega del nus u_i a $L[v]$, en temps constant pot identificar l'aresta que segueix a (v, u_i), utilitzant el punter encreuat i el punter a la llista circular. Això es pot fer en temps $O(1)$ amb $O(n)$ operacions. Per tant, l'algorisme 21 pot ser implementat amb una PRAM-EREW en $T(n) = O(1)$ i $O(n)$ operacions. A la figura 9.3 podem veure l'estructura estesa per a l'exemple de la figura 9.1 amb el circuit eulerià E, que correspon als punters afegits a aquesta estructura en la figura 9.3. A partir d'ara, sempre que ens donin com a entrada un circuit eulerià d'un arbre, assumirem que se'ns dóna la llista de punters E afegida a l'estructura de dades estesa. Donat un vèrtex de tall $r \in V$, l'algorisme 22 ens permet calcular un recorregut eulerià que comença a r. L'algorisme calcula la posició de cada aresta a aquest recorregut. Observem que al pas 3 l'algorisme produeix una enumeració ordenada del recorregut eulerià, començant per l'aresta (r, u_1), on u_1 és el primer element de la llista $L[r]$. Al final cada aresta coneixerà la seva posició en el recorregut eulerià, que comença i acaba a r. L'algorisme 22 es pot implementar amb una PRAM-EREW en $O(\log n)$ passos i $O(n)$ operacions. Per tant, donat

Algorisme 22 Càlcul de la seqüència d'un recorregut eulerià

```
    RECORREGUT(E, r, R)
1   Identificar l'aresta e corresponent a
        l'últim vèrtex u que apareix a L[r]
2   R[0] := E[e]; R[e] := e
3   ENUMERA(R, d)
```

un arbre, hem demostrat el teorema següent.

Teorema 9.1. *Podem calcular un recorregut eulerià d'un arbre, començant per un vèrtex donat, en $O(\log n)$ passos i amb $O(n)$ operacions en una* PRAM-EREW

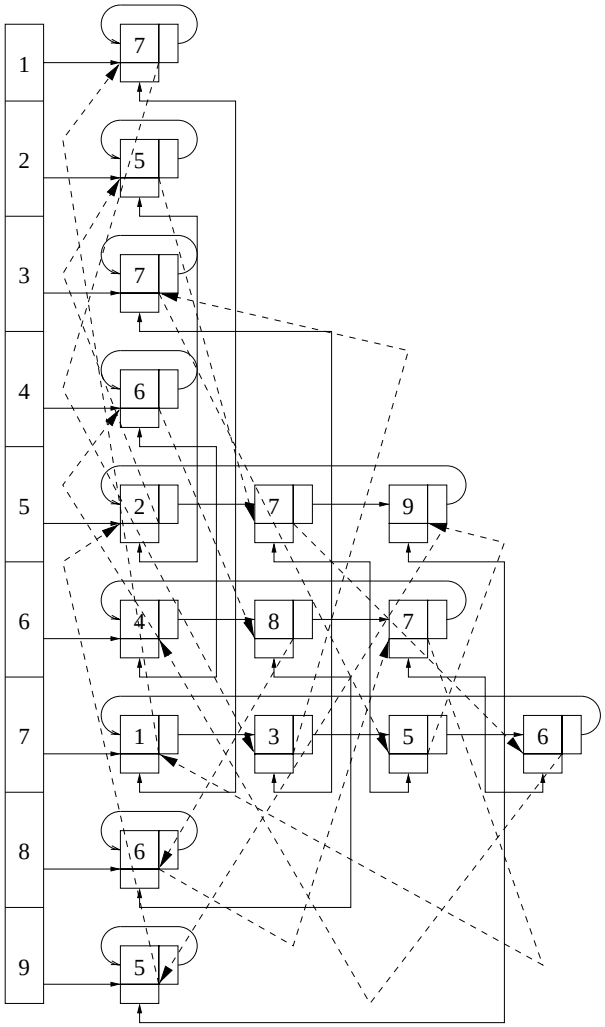


Figura 9.3: Estructura de dades estesa amb circuit eulerià.

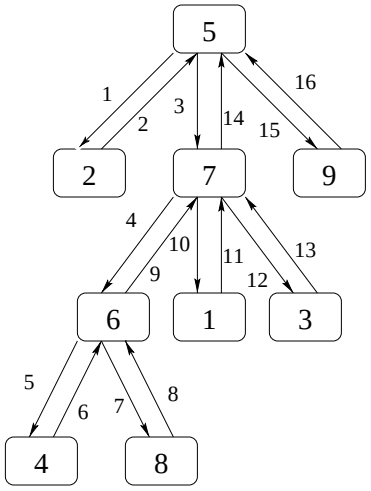


Figura 9.4: Arbre arrelat amb el seu recorregut eulerià començant a l'arrel.

9.2 Aplicacions del recorregut eulerià

La tècnica del recorregut eulerià d'un arbre s'utilitza per processar i computar diverses funcions sobre arbres. La primera aplicació que presentarem és **l'arrelament d'un arbre**. Aquesta tècnica es pot utilitzar per calcular diferents paràmetres i recorreguts d'un arbre, com els recorreguts en postordre, preordre o inordre, el càlcul del nivell d'un nus de l'arbre, el nombre dels descendents d'un nus.

Exemple 9.1 (Arrelament). *Siguin un arbre T donat per l'estructura estesa, un circuit eulerià definit per E i un vèrtex r, que és l'arrel. Volem calcular per a cada nus $v \neq r$ el nus $f(v)$ que és el seu pare a T.*

En seqüèncial aquest problema es pot resoldre de manera òptima en $O(n)$ passos. Per implementar-lo en paral·lel cal tenir en compte la manera com hem calculat el recorregut eulerià. A la figura 9.4 es presenta el recorregut eulerià començant pel vèrtex 5 del graf a la figura 9.1. Observem que al circuit eulerià l'aresta que va d'un nus al seu pare apareix després de l'aresta, que va del seu pare a ell. Així, per determinar quin és el pare d'un nus només cal determinar quina de les dues arestes apareix primer al circuit. Per determinar aquesta condició, enumerem la llista corresponent al recorregut eulerià. L'algorisme 23 dóna l'arrelament d'un arbre, que ve donat com un circuit eulerià.

El pas 1 es pot fer en $O(\log n)$ passos amb (n) operacions; el pas 2 també es pot fer en $O(\log n)$ passos amb $O(n)$ operacions. Per tant, podem implementar l'algorisme 23 amb una PRAM-EREW amb $T(n) = O(\log n), W(n) = O(n)$ i, per tant, és òptim.

Exemple 9.2 (Postordre). *Sigui T un arbre arrelat a r i amb n vèrtexs, que ve donat per l'estructura de dades estesa, calculem la funció* post $: V \to [n]$ *que a cada vèrtex li assigna la seva enumeració en postordre.*

Algorisme 23 Arrelament d'un arbre.

```
    ARRELAR(E, r, p)
1   EULER(E, R)
2   ENUMERA(E, d)
3   per a tot aresta (u, v) parfer
4       si d[(u, v)] < d[v, u]
6           llavors p[v] := u
        fsi
    fper
```

Recordem que en l'enumeració en postordre ve donada de manera recursiva: recorre en postordre els subarbres d'esquerra a dreta, visita l'arrel. Aquest algorisme té una complexitat seqüèncial $O(n)$ i és òptim. Per calcular en paral·lel la funció post, assumirem que tenim donat un recorregut eulerià E de T, que comença i acaba amb r. Per obtenir-ne el resultat, observem que podem calcular aquest recorregut assignant valors adequats a les arestes del recorregut eulerià, de manera que si, calculem les sumes prefixades a una de les dues arestes que uneixen un nus amb el seu pare, aparegui el valor correcte del nus. Si mirem el recorregut eulerià a la figura 9.4, es veu fàcilment que si assignem pes 0 a les arestes que van del pare al fill i pes 1 a les que van del fill al pare, i calculem les sumes prefixades, el valor assignat a les arestes del fill cap al pare tindran l'ordre del fill en el recorregut en postordre. Aquest argument és formalitzat a l'algorisme 24, que calcula la funció post.

Algorisme 24 Enumeració d'un arbre en postordre.

```
    POSTORDRE(E, r, c)
1   ARRELAR(E, r, p)
2   per a tot v ≠ r parfer
3       w[(v, p[v])] := 1; w[(p[v], v)] := 0
    fper
4   Suma-prefixada(E, w, s)
5   per a tot v ≠ r parfer
6       c[v] := s[(v, p[v])]
    fper
7   c[r] := n
```

Una anàlisi semblant a la que hem fet per a l'algorisme 23 ens dóna que podem implementar l'algorisme 24 amb una PRAM-EREW en $O(\log n)$ passos i $O(n)$ operacions. Per tant, és òptim.

Els exercicis **9:2–9:5** presenten altres exemples d'aplicació directa de la tècnica del recorregut eulerià d'un arbre.

9.3 Avaluació d'expressions aritmètiques

Considerem el problema d'avaluar expressions aritmètiques construïdes sobre nombres reals i amb els operadors $(+, -, *, /)$. Aquest tipus d'expressions poden venir donades en forma d'una cadena o en forma d'un arbre de derivació, on les fulles contenen els reals i els nusos interns les operacions binàries. En aquest darrer cas, el resultat de l'avaluació serà el valor a l'arrel de l'arbre. (Observem que amb les operacions definides, l'arbre serà binari). En aquesta secció considerarem el problema d'avaluar expressions donades com arbres de derivació.

Un primer algorisme d'avaluació, és basat en el paradigma de data flow: Associem un processador amb cada nus intern. Cada processador espera fins que els arguments del seu operador han estat computats i després executa l'operador. Un exemple de l'aplicació d'aquest paradigma es troba a la figura 9.5. Observem que la implementació seqüencial d'aquest algorisme té una complexitat òptima de $O(n)$. En paral·lel, l'algorisme es pot implementar amb una complexitat de $O(h)$ passos, on h és l'alçada de l'arbre. Per a un arbre equilibrat, aquest algorisme té un temps paral·lel $O(\log n)$. Malauradament, si $h = \Omega(n)$, aleshores el temps serà $\Omega(n)$. Volem assegurar que es pot avaluar qualsevol arbre de manera òptima en $O(\log n)$ passos independentment de quin sigui el seu perfil.

Per realitzar aquestes fites òptimes en paral·lel, necessitem utilitzar l'anomenada tècnica de **contracció** (shunt o rake), que ens permet contraure un arbre amb n nodes a un arbre amb només tres nusos en $O(\log n)$ passos. Sigui $T = (V, A)$ un arbre binari arrelat a r. Per qualsevol nus u, sigui $p(u)$ la funció que ens dóna el pare de u a T. Definim ara una operació de contracció de fulles. Sigui v una fulla a T, de manera que $p(v) \neq r$ i sigui u el seu germà. La contracció de v, $\mathrm{sh}(v)$ consisteix a substituir els nusos u, v i $p(u)$ en un sols nus, que té com a pare $p(p(v))$ i com a fills els fills del nus u (vegeu l'exemple a la figura 9.6).

Per contraure un arbre, volem eliminar de cop tantes fulles com sigui possible. Direm que dues fulles són **independents** si es poden eliminar simultàniament, és a dir, els conjunts de nodes involucrats en la contracció de cada fulla és disjunt. Per exemple, una fulla no penja directament de l'avi d'una altra fulla. El lema següent ens dóna condicions suficients perquè les fulles siguin independents. La demostració és fàcil i es deixa com a exercici.

Lema 9.1. *Siguin $w_1, \ldots, w_m$ totes les fulles de T enumerades d'esquerra a dreta. Aleshores, w_i i w_j són independents si*

1. $|i - j| > 1$, *i*
2. *w_i i w_j són les dues o esquerres o dretes*

Donat un arbre d'avaluació T arrelat a r, per aplicar l'operació de contracció en paral·lel fins a obtenir un arbre amb únicament 3 nusos seguirem l'esquema de l'algorisme 25 A la figura 9.7 es presenta l'aplicació d'aquest esquema a l'arbre de la figura 9.5. A l'enumeració del pas 1, es deixen el primer i el darrer sense enumerar per evitar els casos que la primera

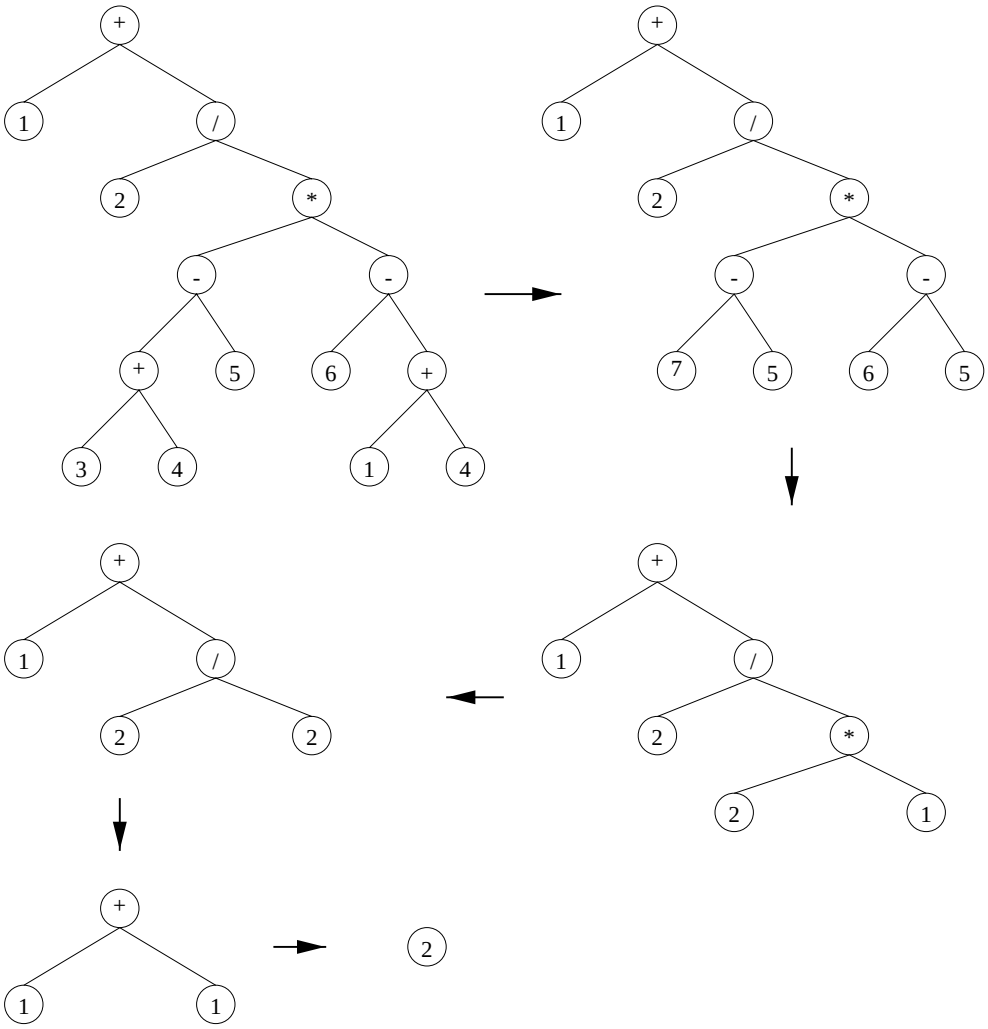


Figura 9.5: Avaluació d'una expressió seguint el flux de dades.

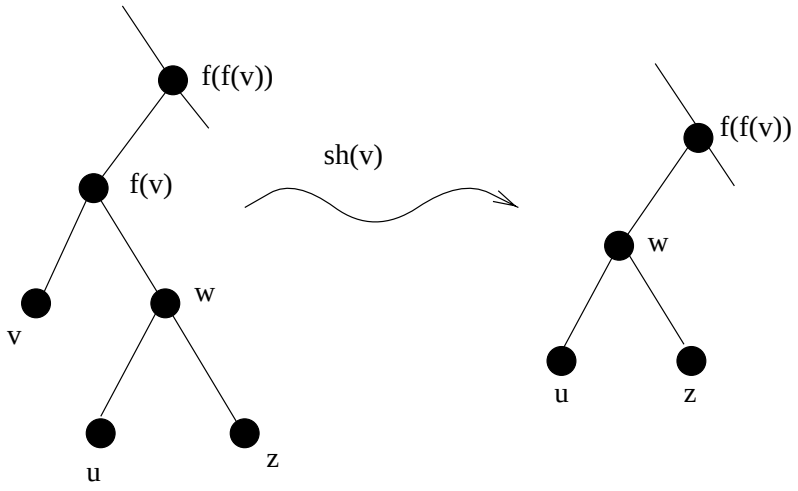


Figura 9.6: Resultat de la operació de contracció de una fulla.

Algorisme 25 Esquema per identificar conjunts independents de fulles.

CONTRACCIÓ

1. Enumerar totes les fulles d'esquerra a dreta, exceptuant la primera i la darrera
2. Repetir fins que no hi hagi fulles:
   - 2.1 Eliminar totes les fulles dretes que estan enumerades amb un senar.
   - 2.2 Eliminar totes les fulles esquerres que estan enumerades amb un senar.
   - 2.3 Tornar a enumerar les fulles que queden.

(o la darrera) fulla de T pengin directament de r. Al bucle 2 separem el cas de les fulles enumerades amb parells del cas de les fulles enumerades amb senars, per evitar aplicar al mateix temps l'operació de contracció a v i a u, on $p(v)$ i $p(u)$ són adjacents a T. Si al començament de la iteració k-èsima de 2 tenim m_k fulles, amb l'aplicació de 2.1 i de 2.2 s'eliminen totes les $\lfloor m/2 \rfloor$ fulles amb índex senar. Per tant, si comencem amb n fulles, necessitem $\lceil \log n \rceil$ iteracions, per arribar a tenir dues fulles i l'arrel. Observem que al pas 2.3, totes les fulles de l'arbre després de la contracció o be són fulles que ja ho eren abans, però amb enumeració parella; o bé s'han obtingut per contracció de dues fulles, una amb enumeració parella i l'altra amb enumeració senar. Utilitzant l'enumeració parella associada, dividint-ho per dos, ens dóna la nova enumeració. Per tant, aquest pas es pot implementar amb temps constant.

Anàlisi i implementació: Cadascun dels passos 2.1 i 2.2 es pot implementar amb temps paral·lel de $O(1)$ i amb $O(n)$ operacions. Per tant, tot l'algorisme té $T(n) = O(\log n)$, però amb $O(n \log n)$ operacions. Podem reduir el nombre d'operacions a $O(n)$ de la manera

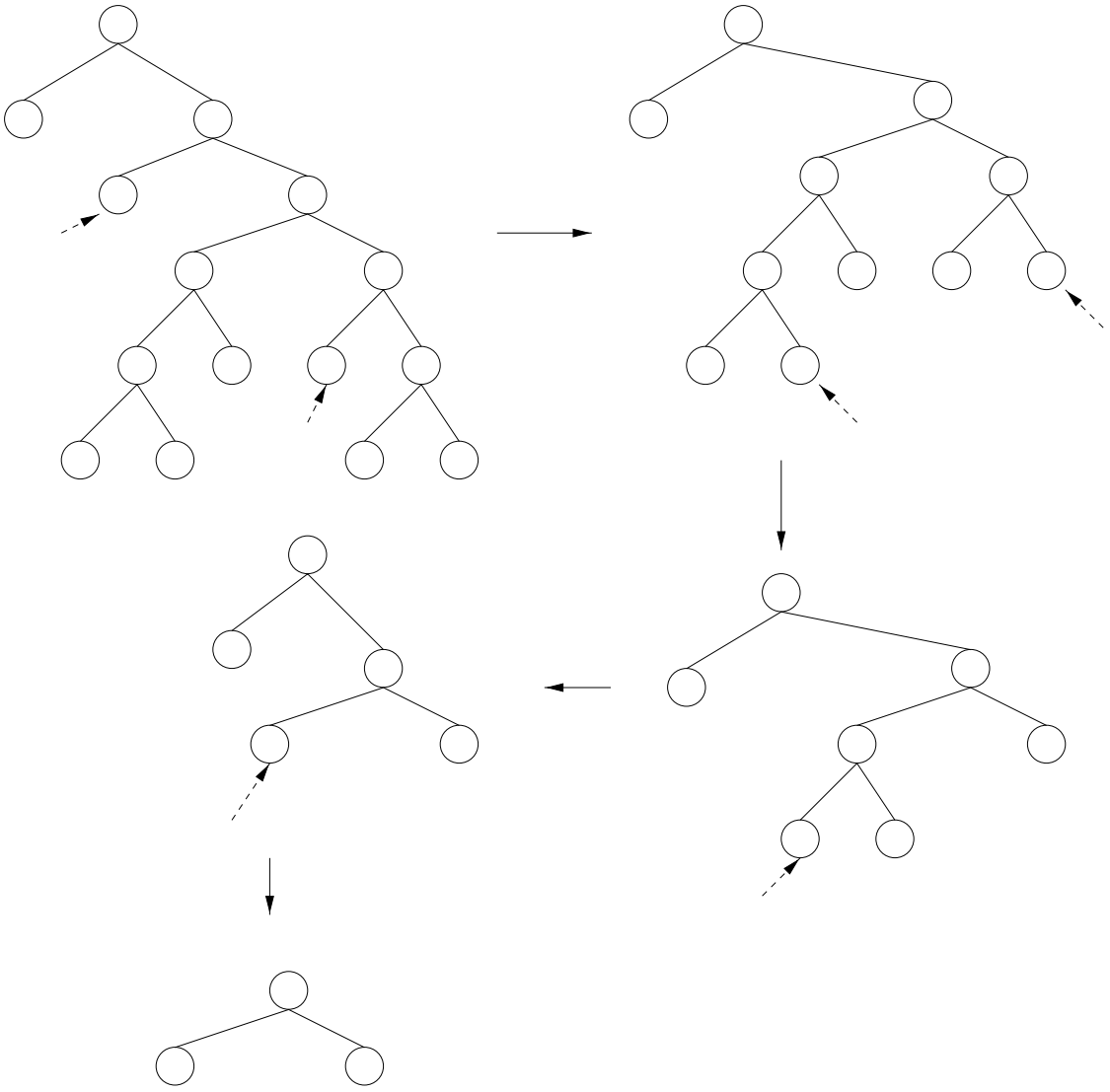


Figura 9.7: Contracció d'un arbre.

següent: En la iteració k-èsima, únicament necessitem utilitzar les fulles $1, \ldots, n/2^{k-1}$. Per tant, podem implementar la iteració k-èsima amb $O(n/\log n)$ processadors en temps

$$O\left(\left\lceil \frac{n/2^{i-1}}{n/\log n} \right\rceil\right) = O(\lceil \log n/2^i \rceil)$$

i això es pot fer per a totes les $\log n$ iteracions. Per tant

$$O\left(\sum_{i=1}^{\log n} \lceil \log n/2^i \rceil\right) = O(\log n(1 + \sum_{i=1}^{\infty} 2^{-i})) = O(\log n)$$

Respecte al pas 1, hem de determinar quina aresta s'identifica amb una fulla; això ho farem quan l'aresta acaba en fulla. És a dir, si tenim un recorregut eulerià començant a l'arrel podem identificar les fulles quan l'aresta següent de $(p(u), u)$ és la $(u.p(u))$ i, en aquest cas, la fulla s'identifica amb l'aresta $(p(u), u)$. Assignant pes 1 a les arestes associades a fulles i calculant sumes prefixades, obtenim l'enumeració de les fulles. Aquest procediment queda reflectit a l'algorisme 26. La iteració al pas 2 de l'algorisme 26 es pot

Algorisme 26 Enumeració de les fulles d'un arbre

```
    ENUMERA-FULLES()
1   EULER(E, R)
2   per a tot (u, v) a T parfer
3       si (u, v) s'identifica amb una fulla
4           llavors , b[(u, v)] := 1
5           altrament b[(u, v)] := 0
        fsi
    fper
6   Suma-prefixada(b, c)
```

realitzar en $O(1)$ passos utilitzant l'estructura de dades estesa. Al pas 3 identifiquem en un vector només les fulles; per tant, les sumes prefixades ens donen una enumeració de les fulles. Amb els resultats nateriors hem demostrat el teorema següent.

Teorema 9.2. *Utilitzant l'operació de contracció de fulles podem reduir un arbre binari arrelat amb n nusos, a un arbre binari arrelat amb 3 nusos, en $O(\log n)$ passos i $O(n)$ operacions, utilitzant una* PRAM-EREW. *Per tant, l'algorisme* CONTRACCIÓ *és òptim.*

Utilitzarem ara la contracció per avaluar expressions aritmètiques. Utilitzarem bàsicament l'algorisme 25, però tenint en compte que quan eliminem un nus també hem de fer alguns canvis i emmagatzemar els resultats parcials necessaris per avaluar correctament els nusos. Sigui $\mathcal{F}$ la classe de funcions $f(x) = \frac{ax+b}{cx+d}$ per $a, b, c, d \in \mathbb{R}$ i recordem que

$$\mathrm{Dom}(f) = \{x \in \mathbb{R} \mid cx + d \neq 0\}.$$

Les funcions a $\mathcal{F}$ es poden representar per la quàdrupla (a, b, c, d). Recordem que una **extensió** de g és una funció h tal que $\mathrm{Dom}(g) \subseteq \mathrm{Dom}(h)$ i tal que $\forall x \in \mathrm{Dom}(g)\ h(x) = g(x)$. Enunciarem dos lemes: el primer estableix el tancament de la classe $\mathcal{F}$ respecte a les operacions amb constants i el segon estableix el tancament de la classe $\mathcal{F}$ respecte a la composició.

Lema 9.2. *Sigui $f \in \mathcal{F}$ i sigui g una funció de la forma $g = f \diamond k$ ($g = k \diamond f$), on $k \in \mathbb{R}$ i l'operador $\diamond \in \{+, -, *, /\}$. Aleshores $\mathcal{F}$ conté una extensió de g que es pot computar en temps $O(1)$.*

Demostració. Veurem la demostració d'un cas i la resta es deixa com a exercici. Vegem aquest resultat per la divisió. De la definició de $g = f/k$ tenim

$$g(x) = f(x)/k = \frac{\frac{ax+b}{cx+d}}{k} = \frac{ax+b}{k\,cx + k\,d}$$

i aquesta fórmula ens permet calcular la representació de g a partir de la representació de f en $O(1)$ passos. □

El lema següent es demostra de manera similar a l'anterior i es deixa com a exercici.

Lema 9.3. *Si $f, g \in \mathcal{F}$, aleshores $\mathcal{F}$ conté una extensió de $f \circ g$ que es pot computar en temps $O(1)$.*

La base de l'algorisme és associar amb cada nus intern u, a més del seu operador $\diamond_u$, una funció d'actualització f_u. En general un nus intern ha de calcular un valor en funció dels valors dels seus dos fills, quan una fulla es contrau, un d'aquest dos valors és conegut mentre que l'altre pot ser que encara no hagi estat calculat. En aquest cas, l'avaluació del valor la podem veure com una funció d'una variable, corresponent al fill que no és una fulla, que ens donarà un valor concret quan l'apliquem al valor associat quan el fill sigui avaluat. La representació d'aquesta funció és el que emmagatzemarem al nus contret.

Inicialment, per a tots els nusos interns u de T, tindrem $f_u(x) = x$ (la funció identitat). Observem que aquesta funció ve representada per $(1, 0, 0, 1)$. Sigui w la fulla que es vol contraure i que és el germà de v; l'operació $sh(w)$ associada a la contracció té dues possibilitats:

1. v és una fulla. Si u és el pare de v, siguin a_v, a_w els valors de v i w, respectivament. La contracció de w ens donarà una fulla i $sh(w)$ determina que el real $a_v = f_u(a_v \diamond_u a_w)$ sigui el valor associat a la nova fulla (vegeu la figura 9.8).

2. v no és una fulla. Sigui f_v la funció associada amb v i f_u la funció associada amb u; aleshores l'operació $sh(w)$ contrau els nusos u i w al nus v, amb operació $\diamond_v$. A més, si w és una fulla esquerra, la funció $f_v = f_u \circ (f_v \diamond_u a_w)$, i si w és la fulla dreta, la funció $f_v = f_u \circ (a_w \diamond_u f_v)$. Recordem que després de l'operació de contracció, del nou nus pengen els mateixos descendents que abans de la contracció penjaven del nus v (vegeu la figura 9.9).

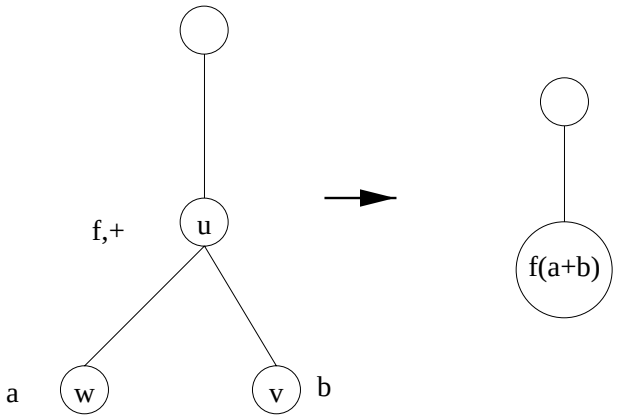


Figura 9.8: Funció shunt corresponent a la contracció de dues fulles.

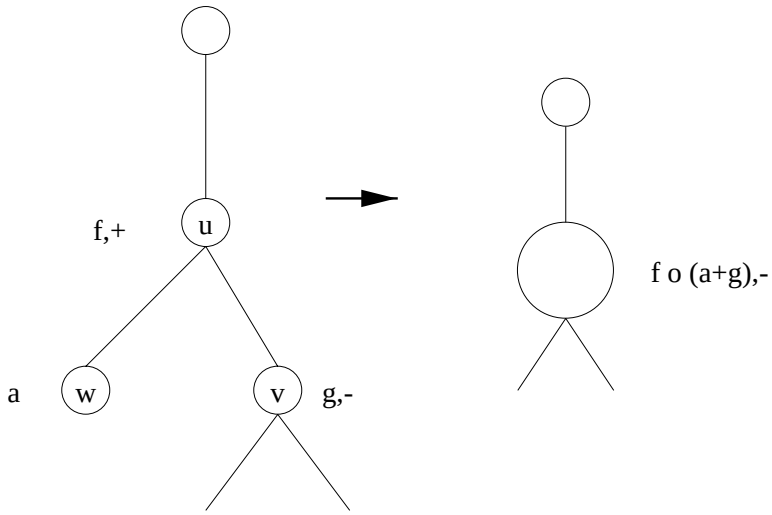


Figura 9.9: Funció shunt corresponent a la contracció d'una fulla amb un nus no fulla.

Incorporant aquests dos casos a l'algorisme de contracció tenin l'esquema de l'algorisme 27. Un exemple de l'aplicació d'aquest algorisme es dóna a la figura 9.10, on per a cada pas hem computat, a més, els coeficients de la funció emmagatzemada a cada nus.

Algorisme 27 Contracció per l'avaluació d'expressions.

AVALUA()

1. Per tot nus intern $u \in T$ **fer** $f_u =$ Identitat
2. Aplicar l'algorisme CONTRACCIÓ, afegint informació als nusos contrets d'acord amb les regles de l'operació **shunt**.
3. Quan ens quedem amb l'arrel i dues fulles u i w, computar $a_r = f_r(a_v \diamond_r a_w)$

En conclusió, hem demostrat el teorema següent,

Teorema 9.3. *Donat un arbre binari T corresponent a una expressió aritmètica sobre el conjunt d'operacions $\{+, -, *, /\}$, aplicant l'algorisme* AVALUA *podem avaluar T en $O(\log n)$ passos i $O(n)$ operacions; a més, l'algorisme es pot implementar amb una* PRAM-EREW.

9.4 Referències bibliogràfiques

La tècnica del recorregut eulerià d'un graf va ser descrita per Tarjan and Vishkin [TV85]. Avui es pot trobar a tots els llibres d'algorismes paral·lels. L'avaluació paral·lela d'expressions va ser introduïda per Brent [Bre73]. La utilització de la tècnica de contracció implementada amb PRAM, tal com la hem explicat, és de [ADKP89]. També ha estat utilitzada per a l'avaluació de context free gramàtiques lliures de context [GR89], codi lineal [MRK88] o polinomis [VSBR83]. Altres aplicacions s'especifiquen al llibre [GR88, JáJ92, Rei93].

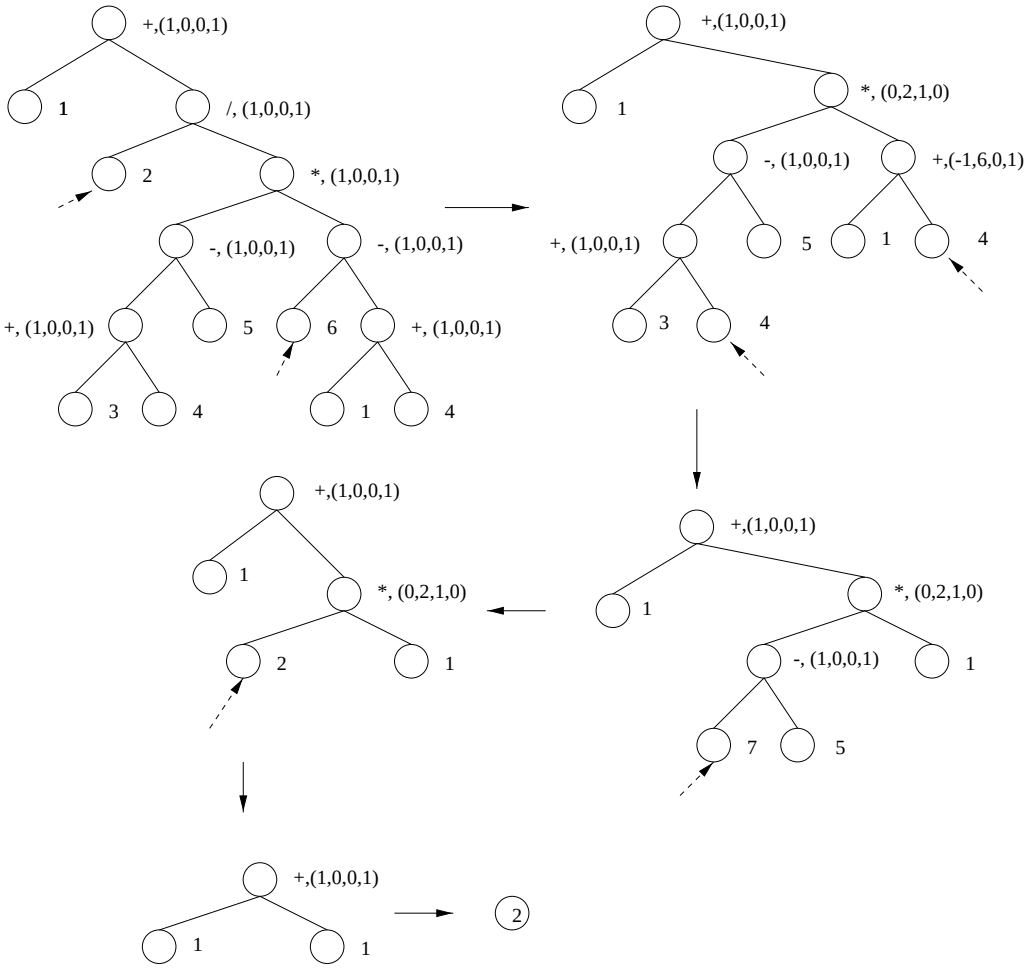


Figura 9.10: Avaluació paral·lela d'una expressió.

Exercicis

9:1 Dissenyeu un algorisme per a la PRAM-EREW tal que, donada una estructura de dades del tipus llista d'adjacència, la converteix en una estructura de dades estesa, en $O(\log n)$ passos amb $O(n)$ operacions.

9:2 Dissenyeu un algorisme per la PRAM-EREW òptim, amb $O(\log n)$ passos, per calcular el **preordre** d'un arbre T ($|T| = n$) arrelat a r. (L'arbre ve donat per una estructura de dades estesa.)

9:3 Dissenyeu un algorisme EREW òptim, amb $O(\log n)$ passos, per calcular el **nivell** de cada vèrtex d'un arbre T ($|T| = n$) arrelat a r. (L'arbre ve donat en forma de llistes d'adjacència esteses. Recordem que el nivell d'un vèrtex v és el nombre d'arestes entre v i r.)

9:4 Dissenyeu un algorisme EREW òptim amb $O(\log n)$ passos per calcular el **nombre de descendents** d'un vèrtex a un arbre T ($|T| = n$) arrelat a r. (L'arbre ve donat en forma de llistes d'adjacència esteses. El nombre de descendents de v és el nombre de vèrtexs del subarbre de T arrelat a v.)

9:5 Dissenyeu un algorisme EREW òptim, amb $O(\log n)$ passos, per calcular l'**inordre** d'un arbre T ($|T| = n$) arrelat a r. (L'arbre ve donat en forma de llistes d'adjacència esteses.)

9:6 Demostreu els lemes 9.2 i 9.3.

9:7 Dissenyeu un algorisme per avaluar les subexpressions corresponents a tots els nusos d'un arbre binari arrelat que representa una expressió aritmètica sobre $\{+, *\}$. L'algorisme dissenyat ha de tenir $T(n) = O(\log n)$ i $W(n) = O(n)$.

9:8 Donat un arbre binari, tal que cada nus v conté un enter e_v, i un punter $p(v)$ al pare de v (si v és l'arrel $p(v) = v$), dissenyeu un algorisme tal que per a cada nus v calculi el mínim enter emmagatzemat al subarbre arrelat a v, amb $T(n) = O(\log n)$ i $W(n) = O(n)$.

Capítol 10

Algorismes PRAM per a Grafs

En aquest capítol, presentem alguns algorismes per a grafs. Gran part del treball desenvolupat per l'algorísmica SIMD tracta sobre la resolució de problemes en la teoria de grafs. Als capítols 2, 3 i 4, en parlar d'algorismes sistòlics, ja hem presentat alguns problemes sobre grafs i la seva resolució paral·lela a la xarxa corresponent.

Moltes vegades, les solucions seqüencials de problemes sobre grafs utilitzen el mètode de **cerca amb profunditat prioritària** (depth first search) (que d'ara endavant anomenarem amb l'acrònim DFS). Aquest mètode no s'ha pogut paral·lelitzar.[1] Per tant, per a molts dels algorismes NC desenvolupats per a problemes de grafs, la metodologia utilitzada és completament diferent del cas seqüencial. Assumirem les definicions i els conceptes bàsics sobre grafs.

10.1 Components connexos d'un graf

El primer problema que considerem és un problema important en la teoria de grafs: trobar components connexos d'un graf no dirigit. Recordem que dos vèrtexs d'un graf estan connectats si existeix un camí entre ells. Donat un graf $G = (V, A)$, els **components connexos** de G són els subgrafs connexos de grandària màxima. El problema que volem resoldre és determinar per a cada vèrtex a quin component connex pertany. En seqüencial, aquest problema té una complexitat de $O(|V| + |E|)$. L'algorisme seqüencial utilitza DFS; per tant, la paral·lelització s'ha de fer utilitzant una altra metodologia. Per resoldre el problema en paral·lel, necessitem identificar cada component connex de manera única, i ho farem mitjancant l'identificador més petit entre els vèrtexs que pertanyen al component.

El graf no dirigit $G = (V, A)$ on $V = \{1, 2, \ldots, n\}$ vindrà donat per la seva matriu d'adjacència M, que recordem és booleana i simètrica. Al començament cada vèrtex té com a identificador el seu propi natural. L'algorisme construeix pas a pas un vector C amb longitud n tal que $C[i] = C[j] = k$ si i només si el vèrtex i és al mateix component connex que el vèrtex j i, a més, k és el vèrtex amb etiqueta més petita en aquest component connex.

[1] En el sentit de col·locar a la classe NC.

L'algorisme va construint conjunts de vèrtexs connexos, cada cop més grans (mentres s'hi puguin afegir elements) i recalculant l'identificador.

Primer necessitem algunes definicions. Un **pseudovèrtex** és un conjunt de vèrtexs $\{i, j, k, l \cdots\}$ tals que tots tenen associat el mateix valor s, on s és el vèrtex més petit entre els vèrtexs del conjunt. El vèrtex s s'anomena **p-vèrtex**. A cada iteració de l'algorisme, doblem la grandària dels pseudovèrtexs que es poden augmentar, unificant dos pseudovèrtexs que es puguin unir. El nou p-vèrte, serà el més petit dels dos p-vèrtexs. Una **estrella** és un arbre arrelat on tots els nusos apunten cap a l'arrel. Representem un pseudovèrtex com una estrella, on l'arrel correspondrà amb el seu p-vèrtex. Un **pseudoarbre** és un arbre dirigit amb un únic cicle de longitud 1 o 2.

La idea de l'algorisme és crear un **pseudobosc** (un bosc de pseudoarbres) i contraure cada arbre fins a tenir una estrella. Un cop tenim un bosc d'estrelles obtenim un nou graf amb un nus per a cada estrella. Iterem el procés fins arribar a un graf sense arestes.

La correcció de l'algorisme es dedueix del lema següent, que es deixa com a exercici **10:2**.

Lema 10.1. *Donat un graf $G = (V, A)$ per a cada vèrtex $u \in V$ sigui $n(u)$ el veí de u amb identificador més petit si existeix; altrament definim $n(u) = u$. Llavors, la funció n defineix un pseudobosc.*

L'algorisme CCONNEXOS (algorisme 28) té dues fases. A la primera es calcula un pseudobosc, d'acord amb el lema 10.1. Després es contrau en un bosc d'estrelles i cada estrella esdevé un pseudovèrtex. A la figura 10.1 es presenta un exemple de la primera

Algorisme 28 Components connexos.

```
    CCONNEXOS(G = (V, E))
    k := 0; G_0 = G
1   mentre G té arestes fer
2       per a tot v ∈ V_k parfer
3           si v té veins
4               llavors C[v] := min{u | (u, v) ∈ A}
5               altrament C[v] := v
            fsi
        fper
6       SALT-PUNTER(sobreC)
7       per a tot v ∈ V parfer
8           C[v] := min{C[v], C[C[v]]
        fper
        k:=k+1
9       Contraure els pseudovèrtexs per obtenir el graf G_k
    fmentre
10  Expandir els representants als nusos originals.
```

fase de l'algorisme 28. Per implementar la segona fase, hem de tenir en compte que al

pas 9, quan per calcular el nou graf assignem nous identificadors a cada pseudovèrtex, hem de tenir cura que els nous identificadors preservin l'ordre relatiu al graf precedent. A més, cal mantenir informació sobre l'identificador corresponent en el graf original. Per poder implementar fàcilment la segona fase, mantenim una estructura de dades, formada per punters que es crearan a mesura que apareguin noves estrelles; cada cop que un nus esdevé part d'una estrella i no n'és el representant, com que al pas previ el nus era el representant d'una estrella, modificarem el seu autobucle en el pas anterior, per tal que apunti al seu representant en aquesta fase. Com a resultat, quan finalitzi la primera fase, a més dels identificadors tindrem una estructura de dades, formada per punters, que és un bosc arrelat, amb un arbre per a cada component connex. A la figura 10.2 es presenta l'estructura de dades creada a la primera fase. Observem que al bosc arrelat, cada arrel té l'identificador del component. Per tant, només cal fer $O(\log h)$ iteracions de salt de punter perquè tots els vèrtexs tinguin l'identificador del seu component connex, on h és l'alçada de l'arbre.

Direm que una estrella és **aïllada** si cap dels seus vèrtexs té veïns fora de l'estrella. Per analitzar el cost de la primera fase només cal analitzar l'evolució entre fases del nombre d'estrelles aïllades.

Lema 10.2. *Si n_k és el nombre d'estrelles no aïllades al final de la k-èsima iteració de l'algorisme 28, tenim que per a tot $k \geq 1$, $n_k \leq n_{k-1}/2$.*

Demostració. Les noves estrelles o són estrelles aïllades a la iteració anterior, o es creen en el pas previ per fusió de dues o més estrelles no aïllades. Descomptant si cal l'aparició de noves estrelles aïllades, tenim que $n_k \leq n_{k-1}/2$. □

Per tant, tenim que la primera fase de l'algorisme necessita com a molt $\log n$ iteraccions. Cada iteració necessita $O(\log n_k)$ passos i $O(n_k)$ operacions per fer el salt de punter al pas 6. Però al pas 4 s'ha de calcular el valor mínim d'una llista, per al qual calen $O(\log n_k)$ passos i $O(n_k)$. La segona fase (pas 10) es pot implementar amb les mateixes fites i es deixa com a exercici. Hem demostrat el teorema següent.

Teorema 10.1. *L'algorisme* CCONNEXES *troba els components connexos d'un graf no dirigit en $O(\log^2 n)$ passos i $O(n^2)$ operacions i necessita una* PRAM-CRCW *comuna.*

10.2 Arbre d'expansió mínim d'un graf

Donat un graf $G = (V, A)$ no dirigit i amb pesos $w : A \to \mathbb{R}$, representat per la seva matriu d'adjacència amb pesos W, on $W(i,j) = w(i,j)$, si $(i,j) \in A$ o $W(i,j) = 0$, si $(i,j) \notin A$. Un **arbre d'expansió** (spanning tree) és un subgraf $T = (V, A')$ de G tal que és un arbre. El pes $w(T)$ d'un arbre d'expansió T és la suma dels pesos de les arestes a T. Donat un graf $G = (V, A)$, un arbre d'expansió és mínim (MST) si el seu pes és mínim respecte a tots els possibles arbres d'expansió del graf. El problema de trobar l'arbre mínim d'expansió d'un graf és un problema molt estudiat, ja que és el primer pas per resoldre molts altres problemes de la teoria dels grafs. Existeixen múltiples algorismes seqüencials

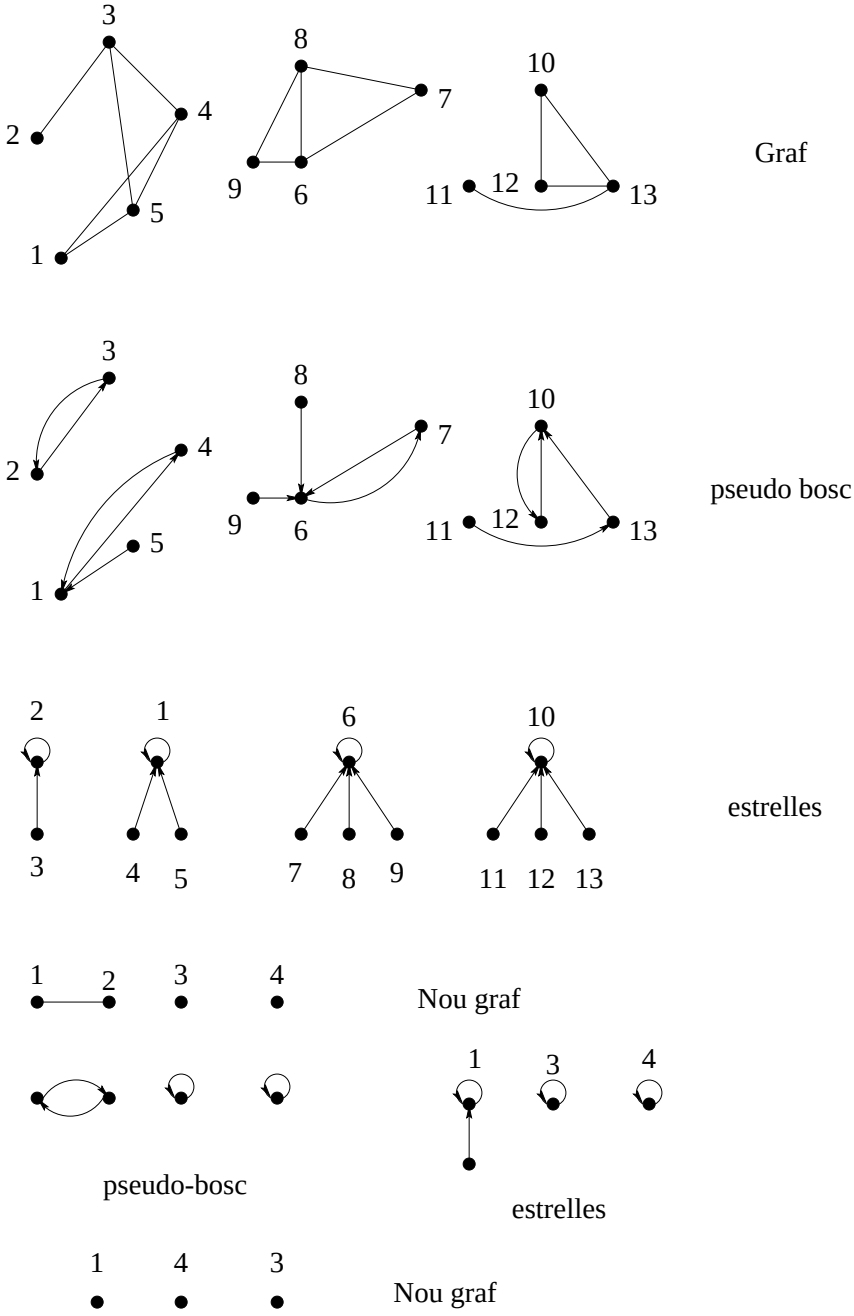


Figura 10.1: Un graf i l'aplicació de la fase I de l'algorisme CCONNEXOS.

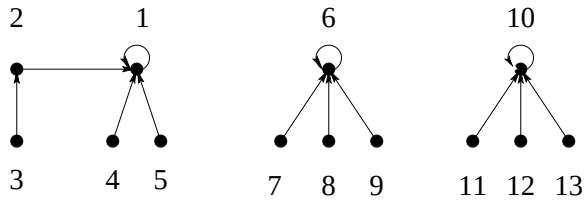


Figura 10.2: L'estructura de dades creada durant la primera fase de l'algorisme CCONNEXOS, amb les etiquetes corresponents al graf original.

del tipus "voraç" per resoldre el problema: l'algorisme de Prim, el de Kruskal i el de Sollin són els mètodes més utilitzats . Normalment se suposa que totes les arestes tenen pesos diferents, cosa que implica que només tenim un arbre d'expansió mínim. Aquesta hipòtesi no presenta cap restricció ja que sempre podem incorporar els extrems de l'aresta al seu pes (és a dir considerar la terna $(w(u,v),u,v)$ quan $u<v$) i classificar-les.

La base teòrica de les estratègies seqüencials és el lema següent,

Lema 10.3. *Sigui $G=(V,A)$ un graf connex amb pesos reals w (tots diferents), i sigui $\{V_i\}_1^k$ una partició de V. Per a cada i, sigui a_i l'aresta de pes mínim que connecta un vèrtex de V_i a un vèrtex de $V \Leftrightarrow V_i$. Aleshores, totes les arestes $\{a_i\}_1^k$ pertanyen a l'arbre d'expansió mínim de G; a més, aquest* MST *és únic per G.*

L'estratègia que seguirem per trobar un algorisme paral·lel és paral·lelitzar l'algorisme de Sollin. Aquest algorisme comença amb una col·lecció de n arbres, cadascun consistent en un únic vèrtex de V, i va reunint a cada iteració diferents subarbres, mantenint l'estructura d'arbre, fins a arribar al MST de G. El procés és molt semblant a la construcció de components connexos que hem vist a la secció anterior, i es dóna com a algorisme 29. A cada iteració seleccionem l'aresta amb pes mínim que surt d'un vèrtex. Aquesta selecció fa que aparegui una sèrie de pseudoarbres que col·lapsem a estrelles. A més, com que volem obtenir l'arbre, marquem aquestes arestes i en finalitzar l'algorisme, el MST estarà format per les arestes marcades. La correcció de l'algorisme ve del lema 10.3 i del lema següent, que es deixa com a exercici.

Lema 10.4. *Sigui $G=(V,A)$ un graf connex. Definim $c(u)=v$ si (u,v) és l'aresta amb pes mínim que surt de u. Llavors*

1. *$(u,c(u))$ pertany al* MST.
2. *c defineix un pseudobosc.*

La diferència fonamental respecte de l'algorisme de components connexos és que hem de calcular el mínim sobre el pes de les arestes, en lloc de fer-ho sobre els identificadors dels veïns. Per tant, hem de mantenir algun tipus d'estructura de dades per poder implementar fàcilment el pas 10 de l'algorisme 29. De la mateixa manera que hem fet per al problema dels components connexos, podem demostrar el teorema següent, els detalls del qual es deixen com a exercici.

Algorisme 29 Arbre d'expansió mínim.

```
    MINST(G = (V, E))
    k := 0; G_0 = G
1   mentre G té arestes fer
2       per a tot v ∈ V_k parfer
3           Sigui (v, u) l'aresta amb pes mínim sortint de v
4           Marcar l'aresta (v, u)
5           C[v] := u
        fper
6       SALT-PUNTER(sobreC)
7       per a tot v ∈ V parfer
8           C[v] := min{C[v], C[C[v]]
        fper
        k:=k+1
9       Contraure els psedovèrtexs per obtenir el graf G_k
    fmentre
10  Tornar marques a les arestes originals.
```

Teorema 10.2. *Donat un graf $G = (V, A)$ no dirigit i connex. Amb pesos w sobre A, que ve donat per la seva matriu de pesos W, l'algorisme* MINST *troba un* MST *en $O(\log^2 n)$ passos, amb $O(n^2)$ operacions, i es pot implementar amb una* PRAM-CREW. *L'algorisme és òptim.*

10.3 Camins mínims entre els vèrtexs d'un graf

Sigui $G = (V, A)$ un graf dirigit, amb pesos $w : A \to \mathbb{N}$, donat per la seva matriu d'adjacència amb pesos W. Recordem que si $|V| = n$, aleshores W és una matriu de $n \times n$, on $W(i, j) = k$ si i només si existeix una aresta de i cap a j amb $w(i, j) = k$. En aquest cas, ens interessa que $W(i, j) = \infty$ quan no hi ha aresta entre els vèrtexs i i j. Considerarem que G no té bucles.

El problema de trobar la distància mínima consisteix a trobar per a qdos vèrtexs ualsevol $i, j \in V$ un camí de i a j, on la suma dels pesos de les arestes sigui mínima. L'algorisme que presentem produirà com a sortida una matriu S amb dimensió $n \times n$ tal que $S(i, j) = 0$ si $i = j$, altrament $S(i, j)$ serà el cost del camí mínim entre i i j. Les modificacions addicionals per tal de calcular també un camí mínim es deixen com a exercici.

L'algorismeEl següent és una implementació paral·lela de l'estratègia seqüencial.

L'algorisme utilitza una matriu auxiliar B amb dimensió $n \times n$ i una taula Q amb dimensió $n \times n \times n$. La matriu B evita la destrucció de W. A cada pas de la iteració, per a cada parell de vèrtexs l'algorisme 30 cerca un camí amb menys cost que l'existent fins al moment. Aquesta cerca es fa considerant tots els camins possibles que utilitzen

Algorisme 30 Càlcul de les distàncies mínimes.

```
      CMIN(M, S)
1     per a tot i, j ∈ V parfer
2           B[i, j] := M[i, j]
3     Repetir log n cops
4           per a tot i, j, k ∈ V(i ≠ j ≠ k) parfer
5                 Q[i, j, k] := B[i, j] + B[j, k]
            fper
6           per a tot i, j ∈ V(i ≠ j) parfer
7                 B[i, j] := min{B[i, j], Q[i, 1, j], ··· , Q[i, n, j]}
            fper
8     per a tot i, j ∈ V parfer
9           si i ≠ j
                  llavors S[i, j] := B[i, j]
                  altrament S[i, j] := 0
            fsi
      fper
```

com a màxim dues vegades el nombre d'arestes considerades al pas anterior. Per tant, a cada iteració, com a màxim doblem el nombre d'arestes utilitzades per descobrir un camí amb cost més petit. Com a màxim tenim que qualsevol camí entre dos vèrtexs pot tenir n arestes; llavors el nombre d'iteracions necessàries serà $O(\log n)$.

Anàlisi Per analitzar-ne la complexitat, observem que les instruccions 2 i 9 es poden executar en $O(1)$ passos i $O(n^2)$ operacions. La instrucció 5 es pot executar en temps $O(1)$ i amb $O(n^3)$ operacions. Trobar el màxim de n elements es pot fer en $O(\log n)$ passos i $O(n)$ operacions, i la instrucció 7 necessita $O(\log n)$ passos i $O(n^3)$ operacions. L'algorisme 30 té una complexitat de $T(n) = O(\log^2 n)$ i $W(n) = O(n^3 \log n)$. Com al pas 5, necessitem escriptura concurrent, tot l'algorisme ha de ser implementat amb una PRAM-CREW. Per tant, hem demostrat el teorema següent.

Teorema 10.3. *Sigui $G = (V, A)$ un graf dirigit amb pesos w sobre les arestes A, que ve donat per la seva matriu de pesos W. L'algorisme* CMIN *troba totes les distàncies mínimes entre vèrtexs, en $O(\log^2 n)$ passos, amb $O(n^3 \log n)$ operacions, i es pot implementar amb una* PRAM-CREW.

10.4 Referències bibliogràfiques

Les definicions i els conceptes bàsics sobre grafs s'expliquen en qualsevol llibre d'algorísmica seqüencial, per exemple [CLR89]. El problema de trobar els components connexos d'un graf és un problema molt estudiat. L'algorisme clàssic seqüencial té complexitat de $O(|V|+|E|)$

[Eve79]. En aquesta referència també ve demostrat el lema 10.3. Històricament, el primer algorisme paral·lel va ser proposat a [HCS79] amb complexitat $T(n) = O(\log^2 n)$ i $W(n) = O(n^2 \log n)$ i model de màquina PRAM-CRCW. Chin, Lam i Chen van modificar aquest algorisme de manera que tingués $O(n^2)$ operacions [CLC82]. Un algorisme òptim per a la PRAM-EREW és a [NM82]. Per a grafs esparsos existeixen algorismes amb temps $O(\log n)$ [AS87].

Una bona referència per al MST en seqüencial és el capítol 24 de [CLR89]. L'algorisme presentat per trobar l'arbre d'expansió mínim està tret bàsicament de [SJ81] i el de camins mínims de [Kuc82].

El problema de camins mínims per a grafs també ha estat extensament tractat. En seqüencial, aquest problema té una complexitat de $O(n|A| + n^2 \log n)$ [EK72], o $O(n^3)$ [Rom89]. L'algorisme paral·lel que presentem es pot trobar al [JáJ92].

Existeixen més tècniques per solucionar altres problemes de grafs, com la descomposició orella, o les tècniques per a grafs planaris. Als llibres [Rei93, JáJ92] es poden trobar bones exposicions d'aquestes tècniques.

Exercicis

10:1 Demostreu el teorema 10.2.

10:2 Demostreu el lema 10.1.

10:3 Modifiqueu l'algorisme 28 per tal que també calculi el nombre de components connexos del graf.

10:4 Dissenyeu l'estructura de dades que s'ha de crear en la primera fase de l'algorisme 29 per tal d'identificar ràpidament les arestes del MST.

10:5 Modifiqueu l'algorisme 29 de manera que, donat un graf no dirigit, no necessàriament connexe, trobi el bosc d'arbres d'expansió de G.

10:6 Modifiqueu l'algorisme 30 de manera que, a més, per a cada parell de vèrtexs connectats ens doni un camí de llongitud mínima.

10:7 Demostreu que si tenim dues matrius booleanes, A i B, amb dimensions $n \times n$, el seu producte es pot computar amb una PRAM-CRCW comuna en $O(1)$ passos i $O(n^3)$ operacions.

10:8 Utilitzeu l'exercici **10:7** per demostrar que la **clausura transitiva** d'un digraf sense pesos i amb n vèrtexs es pot computar amb una PRAM CRCW-comuna en $O(\log n)$ passos i $O(n^3 \log n)$ operacions. (Recordeu que la clausura transitiva d'un graf amb n vèrtexs ve donada per $(I + M)^{\log n}$, on I és la matriu identitat de $n \times n$ i M és la matriu (booleana) d'adjacència del digraf.)

Capítol 11

Problemes P-complets

Recordem que el nostre concepte d'algorisme paral·lel eficient és aquell que ens permet resoldre problemes de manera molt ràpida, en temps polilogarítmic, utilitzant una quantitat no gaire elevada de processadors, un nombre polinòmic. Els problemes que poden ser resolts per algorismes dintre d'aquesta quantitat de recursos pertanyen a la classe NC. Aquesta definició de paral·lelisme captura l'essència de la descomposició estructural d'un problema paral·lelitzable, descomposició donada per la sentzillesa del model de màquina SIMD, i ens en restringeix l'àmbit d'exploració als problemes que ja coneixem que són a la classe P. Recordem que a tot el monògraf hem utilitzat el terme algorisme NC per denotar algorismes SIMD per a la PRAM que troben la solució en un nombre polilogarítmic de passos utilitzant un nombre polinòmic de processadors.

Des del punt de vista relacional, les classes P i NC tenen moltes similituds amb P i NP. Evidentment, de la simulació seqüencial d'una PRAM, utilitzant el lema de Brent (lema 6.1) amb un sol processador, tenim la relació NC$\subseteq$P, però com al cas P i NP, no s'en coneix la relació exacta i únicament es conjectura que P$\neq$NC. Gran part de la fonamentació d'aquesta conjectura es basa en l'existència de problemes "complets" per a la classe P que no han pogut ser paral·lelitzats. Per definir aquesta classe completa, necessitem el concepte de **reduïbilitat** entre llenguatges. Igual que a la teoria de la NP-completesa, considerarem els problemes en versió decisional i codificats com a llenguatges sobre un alfabet $\Sigma = \{0, 1\}$.

11.1 La classe P-completa

Donats dos llenguatges L_1 i L_2, direm que L_1 és **NC reduïble** a L_2 ($L_1 \leqslant^{\text{NC}} L_2$) si existeix un algorisme NC que transforma una entrada x_1 de L_1 en una entrada x_2 de L_2 de manera que $x_1 \in L_1$ si i sols si $x_2 \in L_2$. El lema següent es demostra com a la teoria de la NP-completesa,

Lema 11.1. *Siguin L_1, L_2, L_3 llenguatges tals que $L_1 \leqslant^{\text{NC}} L_2$ i $L_2 \leqslant^{\text{NC}} L_3$, aleshores tenim*

1. $L_1 \leqslant^{\text{NC}} L_3$
2. *si $L_3 \in$ NC, llavors L_1 i L_2 pertanyen a* NC.

Amb el concepte de reducció NC, podem definir la classe P-completa. Un llenguatge L és a la classe **P-completa** si,

1. $L \in \mathrm{P}$, i
2. per a tot $L' \in \mathrm{P}$, $L' \leqslant^{\mathrm{NC}} L$

El lema següent és immediat, i ens diu que si partim de la premissa que $\mathrm{P} \neq \mathrm{NC}$, llavors la classe P-completa està formada pels problemes que no poden ser paral·lelitzats (en el sentit NC).

Lema 11.2. *Si per a un llenguatge L P-complet es demostra que $L \in \mathrm{NC}$ aleshores tindrem que $\mathrm{NC} = \mathrm{P}$.*

Per demostrar l'existència real d'aquesta classe, necessitem un teorema equivalent al teorema de Cook en la teoria de la NP-completesa que doni un primer problema P-complet. Considerem el problema següent, anomenat problema **d'avaluació d'un circuit booleà** (circuit value problem), al qual ens referirem per l'acrònim CVP. Donat un circuit booleà i una assignació de valors a les entrades, és a dir, $C = (g_1, g_2, \cdots, g_n)$, on cada g_i pot ser o una entrada (amb valor 0 o 1) o una porta del tipus $\{\neg, \wedge, \vee\}$, i g_n és la sortida del circuit, el problema consisteix a decidir si la sortida del circuit és 1.

Teorema 11.1. CVP *és* P*-complet.*

Demostració. Donats un circuit i una assignació $C = (g_1, \cdots, g_n)$, els valors de cada porta g_i es poden determinar consecutivament, començant per la primera i acabant per la darrera, la qual cosa es pot fer en temps $O(n)$. Per tant, $\mathrm{CVP} \in \mathrm{P}$.

Per veure que és P-complet, construirem una reducció d'un problema genèric a la classe P a CVP. La reducció segueix l'argument de la reducció donada per Cook [Coo71] per demostrar que el problema SAT és NP-complet. Considerem un llenguatge arbitrari $L \in \mathrm{P}$ això implica que existeix una màquina de Turing amb una cinta M tal que, donada una entrada e per a L, $|e| = n$, M acceptarà e en un temps polinòmic, $p(n)$ per algun polinòmi p. Sigui $Q = \{q_1, \ldots, q_s\}$ el conjunt d'estats de M i $\delta : Q \times \{0, 1, B\} \to Q \times \{0, 1, B\} \times \{E, D\}$ la funció de transició de M, on B és un símbol especial "blanc", E indica que el capçal de M es mou cap a l'esquerra i D indica que el capçal de M es mou cap a la dreta. L'alfabet de M també serà $\Sigma = \{0, 1\}$. Assumirem que al començament M té l'entrada e col·locada en cel·les 1 fins a n, la resta de les cel·les són en blanc. L'estat de començament és q_1 i el capçal és sobre la cel·la 1. En finalitzar després de $p(n)$ passos, el resultat (0 o 1) s'escriu a la cel·la 1.

Per construir el circuit booleà C, començarem per definir uns predicats.

1. $H(i, t)$ tal que $H(i, t) = 1$ si i només si el capçal explora la cel·la i al pas t, on $1 \leq i \leq p(n)$ i $0 \leq t \leq p(n)$.
2. $C(i, j, t)$ tal que $C(i, j, t) = 1$ si i només si la cel·la i conté a_j al temps t, $1 \leq i \leq p(n)$, $1 \leq j \leq 3$, $0 \leq t \leq p(n)$.

3. $S(k,t)$ tal que $S(k,t) = 1$ si i només si l'estat de M al pas t és q_k, $1 \leq k \leq s$, $0 \leq t \leq p(n)$.

El circuit tindrà $p(n)+1$ nivells, un per cada pas de la computació de M sobre e. El nivell corresponent al pas t tindrà tots els subcircuits que ens permeten calcular els predicats $H(i,t)$, $C(i,j,t)$ i $S(k,t)$ per a tots els possibles valors $1 \leq i \leq p(n)$, $1 \leq j \leq 3$, $1 \leq k \leq s$. El nombre de subcircuits a cada nivell és $p(n) + 2p(n) + s = O(p(n))$, que és un polinomi.

El nivell 0 està format per les entrades al circuit:

$$\begin{aligned} H(i,0) = 1 &\Leftrightarrow i = 0; \\ C(i,j,0) = 1 &\Leftrightarrow \text{ la cel·la } i \text{ conté inicialment } a_j; \\ S(k,0) = 1 &\Leftrightarrow k = 1. \end{aligned}$$

Hem d'especificar com podem calcular tots els predicats del nivell $t+1$ $H(i,t+1), C(i,j,t+1), S(k,t+1)$ en termes dels predicats als nivells anteriors.

Considerem els conjunts:

$$\begin{aligned} D &= \{(k,j) | \delta(q_k, a_j) = (*,*,D)\}, \\ E &= \{(k,j) | \delta(q_k, a_j) = (*,*,E)\}. \end{aligned}$$

La posició del capçal necessita la posició al nivell anterior i el moviment:

$$\begin{aligned} H(i,t+1) = H(i \Leftrightarrow 1, t) \wedge \left(\bigvee_{(k,j)\in D} C(i \Leftrightarrow 1, j, t) \wedge S(k,t)) \right) \\ \vee H(i+1,t) \wedge \left(\bigvee_{(k,j)\in E} C(i+1,j,t) \wedge S(k,t) \right) \end{aligned}$$

Per tant, $H(i,t+1)$ pot ser expressada com una seqüència de $\vee$ i $\wedge$ amb longitud constant $O(2s)$, que es pot generar en temps $O(1)$ utilitzant $O(T^2(n))$ processadors.

Per a cada símbol j $(1 \leq j \leq 3)$ definim

$$W(j) = \{(k,j') | \delta(q_k, a_{j'}) = (*, a_j, *)\}$$

llavors els continguts de la cinta es poden calcular com

$$\begin{aligned} C(i,j,t+1) = (\neg H(i,t) \wedge C(i,j,t)) \\ \vee H(i,t) \bigvee_{(k,j')\in W(j)} (C(i,j',t) \wedge S(k,t)) \end{aligned}$$

Per tant, també les $C(i,j,t+1)$ es poden expressar com una seqüència constant de $\vee, \wedge$ i es poden generar en temps $O(1)$ utilitzant $O(T^2(n))$ processadors.

Finalment, per a cada estat k, $1 \leq k \leq s$ definim

$$M(k) = \{(k', j) | \delta(q_{k'}, a_j) = (q_k, *, *)\}.$$

Llavors tenim

$$S(k, t+1) = \bigvee_{(k',j) \in S(k)} \bigvee_{1 \leq i \leq p(n)} (S(k', t) \vee H(i, t) \vee C(i, j, t))$$

Cada $S(k, t+1)$ es pot expressar com una seqüència de $\vee, \wedge$ de longitud $O(p(n))$ que es pot generar en $O(\log n)$ passos utilitzant $O(T^2(n))$ processadors.

Per tant, tots els subcircuits es poden calcular en $O(\log n)$ passos utilitzant un nombre polinòmic de processadors. A més, per construcció M accepta e, si i nomeés si la sortida de C és 1. □

11.2 Altres problemes P-complets

Un cop tenim el primer problema P-complet, es pot utilitzar aquesta informació per incloure-hi altres problemes. Donat un problema A a P, el qual es vol demostrar que és P-complet, hem de demostrar que un problema B que és P-complet es pot reduir a A.

Demostrarem la P-completesa del problema de la **programació lineal** (LP). Donada una matriu d'enters A amb dimensió $d \times n$, un vector d'enters b amb dimensió d i un vector d'enters c amb dimensió n, el problema consisteix a trobar un vector racional x amb dimensió n tal que $Ax \leq b$ i que, a més, maximitzi (minimitzi) $c^T x$.

És conegut, però no trivial, que aquest problema és a P. L'algorisme del mètode el-lipsoïdal degut a Khachian i modificat per Karmarkar dóna una solució polinòmica per al problema de la programació lineal. La reducció de CVP a LP és la següent.

Teorema 11.2. CVP$\leqslant^{\text{NC}}$LP.

Demostració. Donat un circuit booleà i una assignació, codificat com $C = (g_1, \ldots, g_n)$, construim la següent entrada específica per a LP:

1. Generar les desigualtats $x_k \leq 1$ i $x_k \geq 0$ per a $0 \leq k \leq n$.
2. Si g_k és una entrada de C i $g_k = 1$, aleshores generar l'equació $x_0 = 1$; si g_k és una entrada de C i $g_k = 0$, generar $x_k = 0$.
3. Si $g_k = \neg g_j$, generar l'equació $x_k = 1 \Leftrightarrow x_j$
4. Si $g_k = g_i \wedge g_j$, generar les desigualtats $x_k \leq x_i$, $x_k \leq x_j$ i $x_i + x_j \Leftrightarrow 1 \leq x_k$.
5. Si $g_k = g_i \vee g_j$, generar les desigualtats $x_i \leq x_k$, $x_j \leq x_k$ i $x_k \leq x_i + x_j$.
6. Generar $c_i = 0$ per a tot i entre 1 i $n \Leftrightarrow 1$; per a $i = n$ generar $c_n = 1$.

Es deixa com a exercici la demostració de que aquesta reducció es pot fer en temps $O(\log n)$ utilitzant $O(n^2)$ operacions, i que l'entrada construïda de LP té solució si i únicament el circuit original C s'avalua a 1. □

Hi ha una classe de problemes sobre grafs que en general són P-complets. Són aquells problemes que s'obtenen per la utilització d'un algorisme voraç (greedy). Aquests problemes es denominen *lexicogràfics*.

Un primer problema d'aquest tipus és el de la **cerca ordenada amb profunditat prioritària en un graf** (DFS): Donat un digraf $G = (V, A)$ representat per la seva llista d'adjacència i donats tres vèrtexs, $u, v, z \in V$, decidir si començant de u i utilitzant DFS arribarem primer al vèrtex v o al vèrtex z. Recordem que aquesta tècnica té un paper molt important en el disseny d'algorismes seqüencials eficients. El problema es redueix a, donats un graf G i un vèrtex s, enumerar tots els vèrtexs del graf d'acord amb l'algorisme recursiu 31

Algorisme 31 Càlcul de l'ordre DFS lexicogràphic

```
        DFS(G)
        k:=0
1       per i = 1 fins n fer
2           m[i] := 0;
        fper
2       k:=k+1; m[s] = k
3           mentre hi hagi algun veí de s, amb m[u] = 0 fer
4           sigui u el primer vèrtex a
            la llista de s amb m[u] = 0
5           VISITAR(u)
        fmentre


        VISITAR(u)
1       k:=k+1;m[u]:=k;
2           mentre hi hagi algun veí de u, amb m[v] = 0 fer
4           sigui v el primer vèrtex a
            la llista de u amb m[v] = 0
5           VISITAR(v)
        fmentre
```

L'algorisme utilitza un nombre polinòmic de passos i, per tant, el problema de la DFS és a la classe P. Per veure que és P-complet reduirem un cas particular de CVP, NOR-CVP que demana l'avaluació d'un circuit que només té portes NOT i OR, en el que totes les entrades són uns (aquest problema també és P-complet) a DFS.

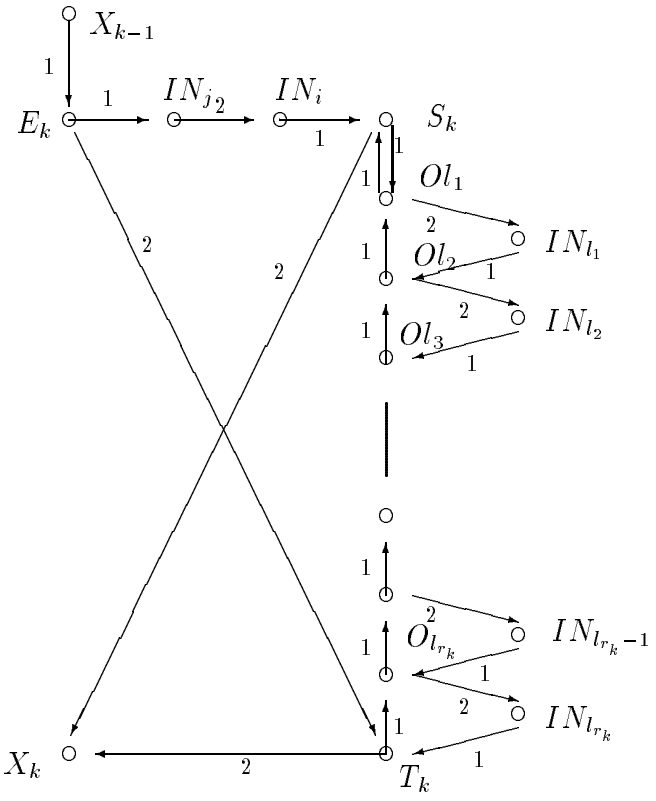


Figura 11.1: El subgraf corresponent a la porta $g_k = \neg(g_i \vee g_j)$ del circuit.

Descriurem ara la reducció. Sigui $g = (g_1, \ldots, g_n)$ un circuit booleà. Assumirem que $b_0 = 0$ i que, per $k = 1, \ldots, n$, $g_k = 1$ (input) o $g_k = \neg(g_i \vee g_j)$, per valors i, j anteriors a k. La reducció consisteix en substituir cada porta per un graf, i després connectar els vèrtexs corresponents a les diferents portes i obtenir un graf final G.

Per a cada k, sigui $I(k) = \{l_1, \ldots, l_{r_k}\}$ el conjunt de portes amb índex major que k que tenen la porta g_k com a entrada. A la figura 11.1 tenim el subgraf G_k que correspon a $g_k = \neg(g_i \vee g_j)$. Aquest graf té un conjunt de vèrtexs

$$V_i = \{E_k, IN_j, IN_i, S_k, O_{l_1}, \ldots, O_{l_{r_k}}, T_k, X_k\}$$

amb les arestes que es donen a la figura. Cada aresta té un nombre què indica l'ordre en que apareix l'aresta a la llista d'adjacència corresponent. Com que estem treballant amb un problema lexicogràfic l'ordre és rellevant. Amb les mateixes convencions, el graf associat a una porta $g_k = 1$ ve a la Figura 11.2. El graf final G és la unió dels grafs G_k amb les arestes adicionals representades en les figures.

La manera com l'algorisme visita cada subgraf depèn dels vèrtexs que s'hagin visitat abans. Suposem que entrem al subgraf G_k des del vèrtex E_k i que cap vèrtex no ha estat visitat abans. Llavors, l'algorisme visita els vèrtexs d'acord amb l'esquema de la figura 11.3. Quan IN_j o IN_i ja han estat visitats, la seqüència és la que ve a la figura 11.4.

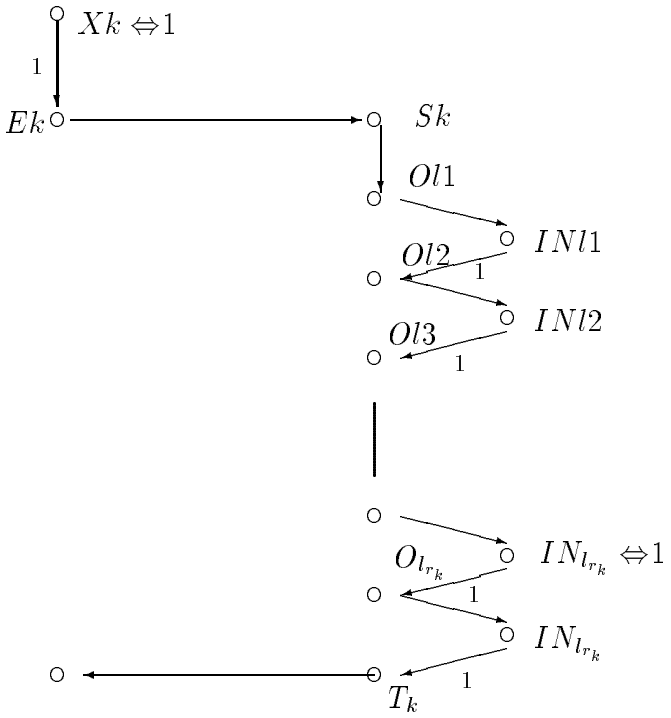


Figura 11.2: El subgraf corresponent a una entrada $g_k = 1$ del circuit.

Per demostrar que la reducció funciona, primer demostrarem un lema. Suposem que comencem el recorregut de G pel vèrtex E_0.

Lema 11.3. *El vèrtex S_n és abans de T_n si i només si el circuit s'avalua a cert.*

Demostració. La demostració és per inducció. Fixem i, $(1 \leq i \leq n)$ i suposem que hem visitat el vèrtex E_i per primera vegada. La nostra hipòtesi d'inducció és que per a tot k, $1 \leq k < i$ es compleix Volem demostrar que per a tot k es compleix:

1. Si g_k s'avalua a cert, llavors els vèrtexs en G_k es visiten d'acord amb la figura 11.3.

2. Altrament, d'acord amb la figura 11.4.

Quan $k = 1$ la porta és una entrada, que val 1, i es compleix la primera part.

Fixem i, $(1 \leq i \leq n)$ i suposem que hem visitat el vèrtex E_i per primera vegada. La nostra hipòtesi d'inducció és que per a tot k, $1 \leq k < i$ es compleix la condició. Anem a veure que també es compleix per k.

Supossem que $g_k = \neg(g_i \vee g_j)$. Quan g_k s'avalua a cert, tant g_i com g_j s'avaluen a fals llavors cap dels vèrtexs IN_i o IN_j han estat visitats abans, i així el recorregut correspon a la figura 11.3. Però quan g_k s'avalua a fals, g_i o g_j s'avaluen a cert, llavors un dels vèrtexs IN_i o IN_j ha estat visitat abans, i així el recorregut correspon a la figura 11.4. □

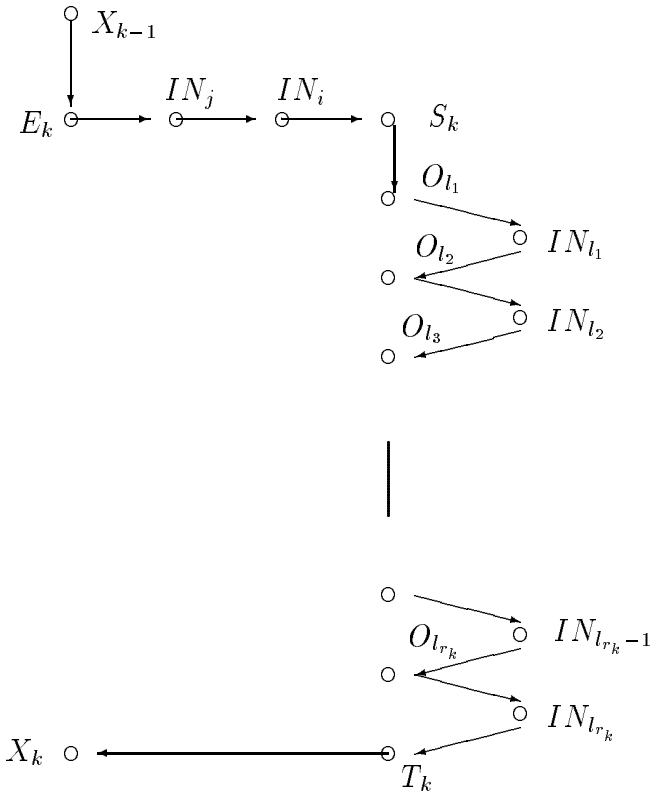


Figura 11.3: Recorregut de G_k sense vèrtexs visitats.

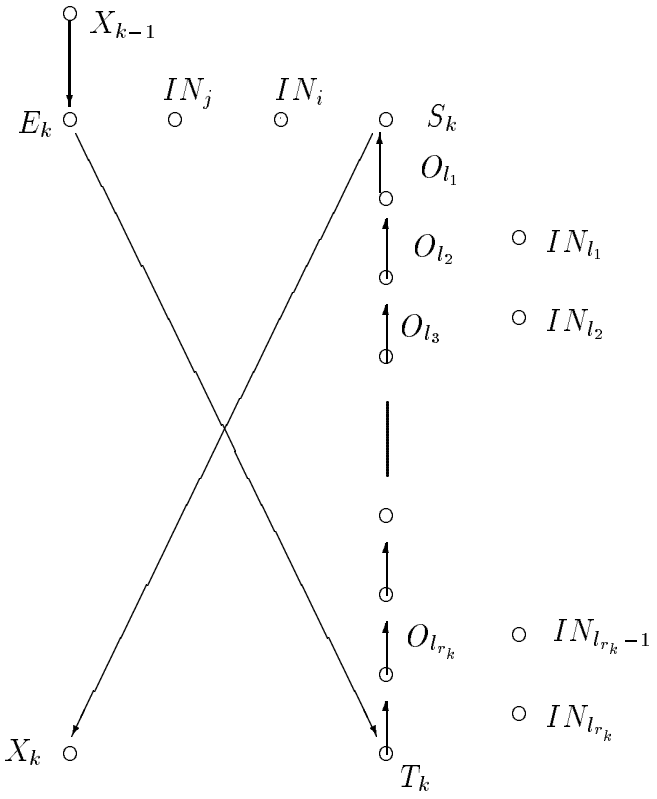


Figura 11.4: Recorregut de G_k amb vèrtexs visitats.

Per finalitzar la reducció, per a cada circuit g construïm el graf G com hem descrit i demanem si el vèrtex S_n apareix abans del vèrtex T_n. Utilitzant el lema 31 i el fet que la reducció es pot calcular amb una PRAM en temps $O(1)$ amb un nombre polinòmic de processadors, hem demostrat el teorema següent.

Teorema 11.3. NOR-CVP $\leqslant^{\text{NC}}$ DFS.

Un altre problema lexicogràfic, P-complet, és el problema de trobar el **primer clique màxim en l'ordre lexicogràfic** d'un graf donat G, (LFMC). Aquest problema es defineix com donat $G = (V, A)$ amb $|V| = n$ calcular el resultat de aplicar l'algorisme voraç següent.

Algorisme 32 Càlcul del primer *clique* a l'ordre lexicogràfic

```
      LFMC
1     U:= ∅
2     per i = 1 fins n fer
3         si v_i és adjacent a tots els vèrtexs a U
4         U := U ∪ {v_i}
      fper
```

L'algorisme utilitza un nombre polinòmic de passos; per tant, el problema del LFMC és a P. Deixem com a exercici la demostració de la seva P-completesa (exercici **11:3**).

Molts problemes han estat classificats a la classe P-completa. L'exemple més paradigmàtic de problema classificat com a P-complet és trobar el **flux màxim** d'un graf: donat un digraf $G = (V, A)$ amb una funció capacitat $c : A \to \mathbb{N}$ i vèrtexs $s, t \in V$, determinar si el flux màxim de s a t és un enter senar.

11.3 Referències bibliogràfiques

Hi ha un llibre dedicat a la teoria dels problemes P-complets que conté una llista exhaustiva de problemes [GHR95]. Aquest llibre es la versió P del famós [GJ79]. També hi ha bons articles d'exposició, un particularment bo és [Tor93]. Per a una exposició més aprofun dels conceptes i resultats bàsics de la teoria de la complexitat, vegeu per exemple, el llibre [BDG88]. En particular, per aprofundir l'estudi de les propietats estructurals de la classe NC, veugeu el capítol 4 de [BDG90]. Part del present capítol ha estat inspirat en la tesi de M.J. Serna [Ser90].

Per a una exposició molt clara sobre el mètode el·lipsoïdal, vegeu el capítol 8 de [PS82]. El resultat de la P-completesa de Max Flow es pot trobar als llibres de paral·lelisme síncron [JáJ92, GR90, Akl89].

Exercicis

11:1 Completeu la demostració del teorema 11.2.

11:2 Demostreu que el problema de **l'avaluació d'un circuit monòton** (MCVP) és a dir, un circuit que només conté negacions de les entrades, és també P-complet.

11:3 Demostreu que existeix una reducció NC de MCVP a LFMC.

11:4 Considereu el problema de les **desigualdats lineals** (LI). Donada una matriu d'enters A amb dimensions $n \times d$, un vector n-dimensional d'enters b, trobeu si existeix un vector racional x amb dimensió d tal que $Ax \leq b$. Demostreu que aquest problema és P-complet.

Capítol 12

Simulació d'una PRAM amb xarxes d'interconnexió

En el model PRAM, tots els problemes de comunicació entre processadors són ignorats. El fet de tenir una memòria comuna fa que sigui relativament senzill dissenyar algorismes per la PRAM, però si volem una implementació real necessitem simular els passos d'una PRAM amb algun tipus de xarxa d'interconnexió.

En aquest capítol veurem com simular els passos d'un algorisme per la PRAM-EREW amb una papallona. Utilitzarem un algorisme randomitzat, és a dir, un algorisme que en un cert moment de la computació pot llançar una moneda i depenent que surti cara o creu, continuar la computació per un camí o per altre. La fita és simular una PRAM amb n processadors utilitzant una xarxa d'interconnexió amb el mateix nombre de processadors, de manera que. amb probabilitat molt gran el cost de la simulació en nombre de passos, és molt petit.

12.1 Un pas PRAM a una papallona

Vegem ara com simular un pas d'escriptura (lectura) de la PRAM-EREW. Utilitzarem una xarxa papallona tancada, en la qual la primera i la darrera columnes estan identificades, i tots els processadors tenen el mateix grau. Considerarem una PRAM-EREW, M', que utilitza $n = k2^k$ processadors $\{p'_0, \ldots, p'_{n-1}\}$, i que té m posicions de memòria comuna $\{\gamma_0, \ldots, \gamma_{m-1}\}$, assumirem que $m \geq n$ (vegeu l'exercici **12:1**). Sigui M una xarxa papallona tancada amb n processadors $p_0, \ldots, p_{n-1}$. Els processadors de la xarxa són idèntics als de la PRAM. Cada processador té una instrucció que genera en una unitat de temps un bit aleatori amb probabilitat 1/2 de ser 0 o 1, els successius bits generats han de ser independents els uns dels altres (i també han de ser independents del bit generat per altres processadors). Com hem suposat, cada processador coneix el seu propi identificador i en el cas dels processadors a la papallona, cada processador pot computar la seva fila i columna en $O(\log n)$ passos (vegeu l'exercici **12:2**).

La papallona tancada es pot representar com un graf $G_n = (V_n, E_n)$, amb conjunt de

vèrtexs $V_n = \{0, \ldots, 2^k \Leftrightarrow 1\} \times \{0, \ldots, k \Leftrightarrow 1\}$, per a cada element $(b, i) \in V_n$ el conjunt E_n conté les arestes:

$$((b, i), (b, (i + 1) \mod k)) \text{ i } ((b \oplus 2^i, (i + 1) \mod k))$$

on $\oplus$ denota la "o-exclusiva" lògica.

En lloc de fer una simulació de la PRAM per la papallona, el que farem és una **emulació** de la memòria global de la PRAM. En el nostre esquema tindrem una xarxa amb $2n$ processadors, $\{p_0, \ldots, p_{n-1}\}$ i $\{q_0, \ldots, q_{n-1}\}$. Els processadors $p_0, \ldots, p_{n-1}$ estan connectats mitjançant la xarxa papallona, i cada un d'ells tindrà en custòdia una part de la memòria global de la PRAM a la seva memòria local, d'acord amb un patró que descriurem més endavant. Modifiquem M' reemplaçant la seva memòria local per M. Després, per a cada $i, 0 \leq i \leq n \Leftrightarrow 1$, la connexió que abans servia per unir p'_i a la memòria global ara serveix per unir q_i a p_i. Direm que M **emula** la memòria comuna de M' si qualsevol programa que funciona a M' també funciona a la màquina modificada, encara que possiblement amb un cost extra en el nombre de passos.

Considerem que cada processador p'_i de la PRAM té dues instruccions per comunicar-se amb la memòria comuna: la instrucció LOAD j que substitueix els continguts de l'acumulador a p'_i pels continguts del valor a γ_j, la segona instrucció és STORE j, que substitueix els continguts de γ_j pels continguts a l'acumulador a p'_i.

L'emulació funciona de la manera següent. Al començament de cada cicle de computació de la PRAM, cada processador p_i rep de q_i una de les tres entrades següents:

1. Un enter j, si la instrucció actual de p'_i és LOAD j. En aquest pas, q_i vol llegir de γ_i,
2. Un parell $< j, a >$, si la instrucció actual de p'_i és STORE j, i a és el contingut a l'acumulador de p'_i. En aquest pas, q_i vol escriure a a γ_j,
3. Un senyal que indica que al pas actual, p'_i no accedeix a la memòria compartida.

En finalitzar l'emulació de cada pas de la PRAM, cada p_i que ha llegit una posició de la memòria global γ_j ha computat el valor actual de γ_i i ho ha pasat a q_i. El valor actual de γ_j es defineix com el segon component de $< j, a >$, on j es la dada entrada més recentment, en un pas anterior a l'actual, per algun processador. Assumirem que almenys un processador de M coneix els valors m i n, i en $O(\log n)$ passos els transmet als altres processadors. També permetrem que cada cel·la de memòria sigui capaç d'emmagatzemar enters de grandària menor o igual que m, i que cada processador de M tingui $\Theta(n)$ cel·les de memòria local.

12.2 L'algorisme

Siguin h i ϕ dues funcions que més endavant definirem. Representarem cada γ_j a la memòria comuna de la PRAM per una cel·la $\phi(j)$ a la memòria local d'un processador $p_{h(j)}$ a la papallona. Després de simular un pas, quan un valor s'emmagatzema per primer

cop a γ_j, $\phi(j)$ conté el valor actual de $\gamma(j)$, i direm que el processador $h(j)$ té γ_j sota la seva custòdia. Com veurem, h i ϕ poden canviar en el transcurs de l'emulació, però al començament de cada pas de la simulació tots els processadors coneixen h i poden computar $h(j)$ en $O(\log n)$ passos.

Suposem que tenim un processador de la papallona que ha d'actualitzar amb un cert valor a una cel·la de la memòria global γ_j. Aleshores comença la simulació del pas, avaluant $h(j)$ i demanant al processador $p_{h(j)}$ que substitueixi a pel valor actual de la cel·la $\phi(j)$. Aquesta sol·licitud va fins a $p_{h(j)}$. Com que cada processador té $\Theta(n)$ cel·les de memòria, $\phi(j)$ serà l'element $\phi(j)$ d'un vector amb m elements a la memòria local de $p_{h(j)}$; per tant, l'actualització es pot realitzar en temps constant.

De la mateixa manera, un processador p_i que ha de llegir de γ_j demana al processador $p_{h(j)}$ quin és el valor actual de la cel·la j-èsima, i $p_{h(j)}$ ha d'enviar cap a p_i el valor a.

Per tant, a més de com s'escolleix el valor de h, el problema de l'emulació es redueix a un problema d'encaminament de missatges (paquets). Considerarem dos tipus de paquets, els generats per sol·licituds d'escriptura i els generats per sol·licituds de lectura. A més, assumirem que cada processador origina com a màxim un paquet i que no hi ha més d'una petició de (lectura/escriptura) per a cada posició de memòria, ja que es tracta d'una PRAM-EREW.

En general, hi pot haver més d'un paquet que vol anar al mateix processador. Per tractar de minimitzar aquest problema, al començament, de manera aleatòria, cada paquet elegeix un identificador (entre 0 i $2^k \Leftrightarrow 1$), que interpretarem com la seva prioritat. A cada fila de la xarxa, li assignem un líder (pot ser el processador central o qualsevol altre). L'encaminament de paquets es fa en quatre fases:

1. A cada fila, permutar els paquets que hi ha a la fila.

2. A cada fila, els paquets que hi són s'encaminen cap al líder de la fila.

3. Els paquets amb destinació una fila determinada s'encaminen cap al líder d'aquesta fila.

4. Cada paquet (a la fila correcta) s'encamina des del líder cap a la seva destinació final.

L'objectiu de la primera fase és classificar els paquets que són en una fila determinada en un ordre de prioritats, de manera que quan comenci la segona fase i els paquets s'encaminin cap al líder de la fila, aquest els rebi en ordre de prioritat. Observem que la fase 1 es pot implementar utilitzant qualsevol algorisme de classificació per al vector lineal, ja que mai no s'utilitza cap de les connexions que connecten dues files diferents.

Cal remarcar que les fases 1, 2 i 4 utilitzen únicament les arestes que connecten els processadors a la mateixa fila. A més, el fet de tenir una papallona tancada fa que tot el flux de dades vagi cap endavant (columnes amb índex més gran) excepte a la fase 2, on l'algorisme de classificació s'utilitza amb dues direccions. A més veiem que les fases 1 i 2 es poden implementar en temps $O(k) = O(\log n)$, i si cada líder encamina paquets a la seva fila a mesura que li arriben, la fase 4 afegirà un altre $O(\log n)$ al càlcul total.

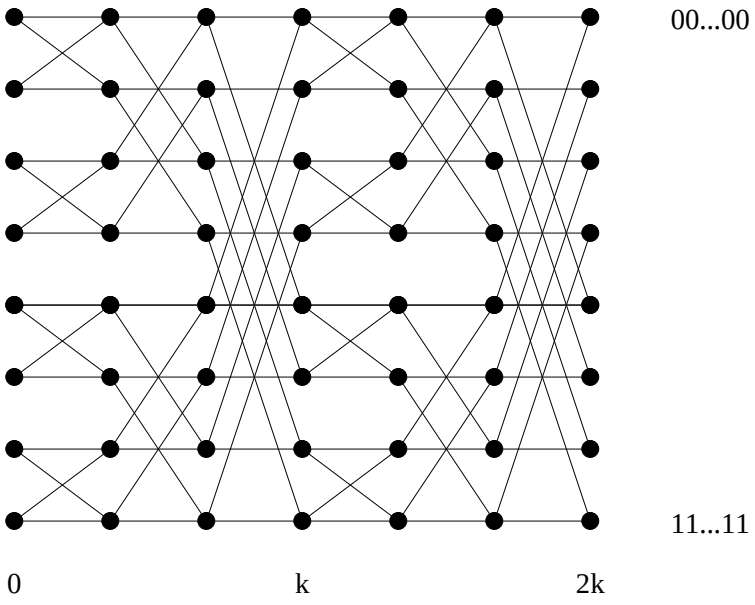


Figura 12.1: Xarxa G'_n

En finalitzar la fase 2, cada líder d'una fila conté com a màxim k paquets que ha d'enviar als líders de les altres files. És convenient descriure l'operació d'encaminament com si tingués lloc en una xarxa G'_n que consisteix en dues xarxes papallona, una darrera l'altra (vegeu la figura 12.1). Cada paquet ha d'anar des de la columna 0 a la columna 2k. Qualsevol estratègia d'encaminament a G'_n es pot simular a G_n, de manera que cada processador simuli 2 o 3 processadors de G'_n (3 en el cas de processadors a la columna 0).

A la fase 3, per enviar els paquets s'utilitza un conjunt de camins pels quals s'envien els paquets. Per determinar el camí, s'utilitza un altre cop l'identificador assignat aleatòriament al començament de l'algorisme. Així s'utilitza l'únic camí que hi ha des de l'origen del paquet a la fila 0, fins a la seva destinació a la fila $2k$, que passa per l'element c de la fila k, on c és l'identificador.

L'algorisme d'encaminament a la fase 3 assumeix que cada processador a G'_n manté una cua FIFO per a cadascuna de les arestes d'entrada. A més dels paquets reals per la xarxa, hi ha uns paquets, anomenats paquets *fantasmes*, que transmeten informació sobre els valors dels identificadors que es poden rebre per la connexió. Cada processador fa el següent:

1. Mira els caps de les dues cues i tria el paquet x amb identificador més petit.
2. Envia el paquet x per la connexió de sortida que correspon al camí que ha de seguir x.
3. Per l'altra connexió, envia un paquet fantasma amb el mateix identificador que x.
4. Els paquets fantasmes que no es poden enviar es descarreguen de les cues.

La figura 12.2 és un exemple de quatre passos consecutius d'encaminament.

Els processadors a les columnes 0 y $2k$ es comporten d'una manera una mica diferent, ja que els de la fila 0 mai no reben missatges i els de la fila $2k$ mai mai envien res. Per tal de mantenir el fluxe de dades, quan un dels processadors a la columna 0 acaba amb els seus paquets, envia contínuament un paquet fantasma per cadascuna de les línies amb identificador ∞, que és més gran que qualsevol altre identificador. Observem que aquest valor sempre es pot calcular, ja que la grandària dels identificadors és fitada i varia entre 0 i $2^k \Leftrightarrow 1$.

Observem que el fet que els paquets arribin al líder classificats per identificador fa que aquesta propietat es mantingui sempre per a tots els processadors, i que l'ús de cues FIFO sigui correcte i permeti mantenir aquesta propietat com a invariant.

12.3 La distribució de memòria

Descriurem ara com assignem les posicions de memòria de la PRAM als diferents processadors. Sigui $\mathbb{H}_0$ el conjunt de totes les funcions del conjunt $\{0,\ldots,m \Leftrightarrow 1\}$ al conjunt $\{0,\ldots,n \Leftrightarrow 1\}$. La nostra assignació de memòria és una funció $h \in \mathbb{H}_0$. Recordem que el processador $p_{h(j)}$ té a la seva memòria local la posició γ_j. Sigui , el conjunt de posicions de memòria en què, en un pas de PRAM, els processadors volen escriure o llegir. Observem que , pot ser qualsevol subconjunt amb menys de n posicions.

El fet de seleccionar un element h de $\mathbb{H}_0$ és molt costós; per això s'utilitza un subconjunt de funcions, $\mathbb{H}$ en el qual es pot fer una selecció aleatòria de h.

Primer construïm un nombre primer $p \geq m$ però $p = O(m)$ (això sempre és possible) i després seleccionem un polinomi Q amb grau $\vartheta \Leftrightarrow 1$ a $\mathbb{Z}_p[x]$, i h serà $h(x) = Q(x) \mod n$. Observem que per seleccionar un polinomi només cal triar aleatòriament els seus coeficients. Identificant cada polinomi amb els seus coeficients, tenim que

$$\mathbb{H} = \{Q \mod n | Q = (q_0,\ldots,q_{\vartheta-1}) \forall i\ 0 \leq i \leq p\}.$$

La propietat següent dóna la probabilitat que més de q elements vagin al mateix processador.

Lema 12.1. *Sigui $q > \vartheta$ un enter, sigui $R \subseteq \{0,\ldots,n \Leftrightarrow 1\}$ i sigui*

$$\mathbb{H}' = \{h \in \mathbb{H} :|\ x \in ,\ : h(x) \in R\}\ | \geq q\}.$$

Aleshores,

$$\frac{|\mathbb{H}'|}{|\mathbb{H}|} \leq \left(\frac{2|R|}{q \Leftrightarrow \vartheta}\right)^{\vartheta}$$

D'aquest lema, prenent els valors adequats tenim:

Lema 12.2. *Per qualsevol $\alpha > 0$, si $\vartheta = \lceil(\alpha+1)\log n\rceil$, el nombre de posicions de memòria en un processador és $O(\log n)$ amb probabilitat $1 \Leftrightarrow n^{\alpha}$.*

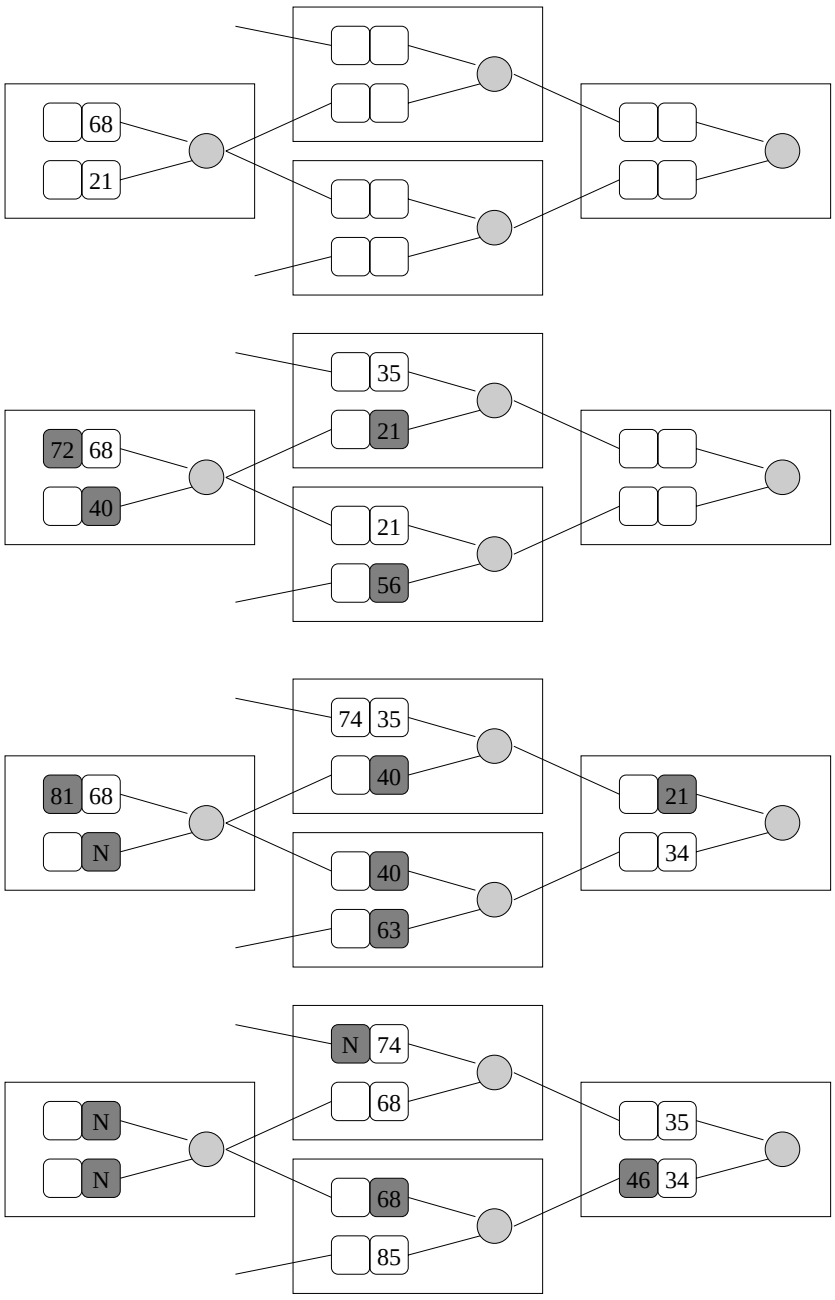


Figura 12.2: Quatre passos consecutius d'encaminament (els paquets fantasmes hi són ombrejats)

12.4 Anàlisi

Analitzem ara el nombre de camins que passen per una aresta fixada de G'_n. Suposant que el nombre de posicions de memòria assignades a cada processador és $O(\log n)$, aquest fet es donarà amb alta probabilitat si utilitzem l'esquema proposat d'assignació de memòria. Per fer l'anàlisi cal recordar el resultat següent de la teoria de probabilitat.

Lema 12.3. *Sigui S una variable aleatòria amb distribució binomial amb mitjana λ. Aleshores, per a qualsevol $a \geq 6\lambda$ tenim*

$$Prob(S \geq a) \leq 2^{-a}.$$

Donat un conjunt de camins $C = \{p_1, \ldots, p_r\}$ a G'_n i una aresta e, definim la **congestió** de e relativa a C com el nombre de camins que contenen e. La *congestió* de C és el màxim de les congestions de les arestes que apareixen a C.

Lema 12.4. *Sigui $r \in \mathbb{N}$ i $x_1, y_1, \ldots, x_r, y_r \in \{0, \ldots, 2^k \Leftrightarrow 1\}$ i sigui*

$$c = \max\{\max_{0 \leq l < 2^k} |\{i \in \{1, \ldots, r\} \mid x_i = l\}|, \max_{0 \leq l < 2^k} |\{i \in \{1, \ldots, r\} \mid y_i = l\}|\}.$$

Per $i = 1, \ldots, r$ seleccionem un valor z_i amb distribució uniforme de $\{0, \ldots, 2^k \Leftrightarrow 1\}$. Definim el camí p_i com l'unic camí a G'_n que conté els vèrtexs $(0, x_i)$, (k, x_i) i $(2k, y_i)$. Sigui $C = \{p_1, \ldots, p_r\}$. Llavors tenim que per a qualsevol $\alpha \in \mathbb{R}$, amb probabilitat almenys $1 \Leftrightarrow n^{-\alpha}$, la congestió de C és $O(c + \log n)$.

Demostració. Per a tot $i = 1, \ldots, r$ denotem per p'_i la primera part del camí p_i, és a dir. la part que va de $(0, x_i)$ a (k, x_i). Veurem que amb probabilitat almenys $1 \Leftrightarrow n^{-\alpha}$, la congestió a $\{p'_1, \cdots p'_n\}$ és $O(c + \log n)$. Per simetra obtenim el resultat.

Fixem un vèrtex $u = (i, b) \in V'_n$, amb $0 \leq i < k$, i una aresta $e = (u, v) \in E'_n$ sortint d'u. Definim una variable aleatòria indicadora X_j per cada valor de $j = 1, \ldots, r$,

$$X_j = \begin{cases} 1 & \text{si } p'_j \text{ conté } e; \\ 0 & \text{altrament .} \end{cases}$$

$X_1, \ldots, X_r$ són independents, i $S = X_1 + \cdots + X_r$ és la congestió de l'aresta e amb relació al conjunt de camins $\{p'_1, \cdots p'_n\}$. Representem per Q_j l'esdeveniment:

> *les expansions binàries de x_j i b són les mateixes, o només difereixen a l'i-èsim bit més significatiu.*

Observem que

$$\Pr(X_j = 1) = \begin{cases} 2^{-i-1} & \text{si } Q_j; \\ 0 & \text{altrament .} \end{cases}$$

Per tant, S segueix una distribució binomial i, tenint en compte que com a molt 2^i adreces només difereixen de b en l'i-èsim bit, llavors tenim que per almenys $2^i c$ valors j Q_j és cert.

Llavors el valor esperat de S es pot fitar per $2^i c 2^{-i-1} = c/2$. Utilitzant el lema 12.3 amb $a = 3c + (\alpha+1)\log n$ tenim

$$\Pr(S \geq a) \leq 2^{-3c} n^{-\alpha-1} \leq \frac{1}{2} n^{-\alpha-1}.$$

Donat que e es pot escollir de $2n$ maneres, la congestió del conjunt de camins $\{p_1, \cdots p_n\}$ és $O(c + \log n)$ amb probabilitat d'almenys $1 \Leftrightarrow 2n\frac{1}{2}n^{-\alpha-1} = 1 \Leftrightarrow n^{-\alpha}$. □

Aplicant el lema anterior, prenent com $(0, x_i)$, (k, x_i) i $(2k, y_i)$: el node que envia, el vertèx del mig i el receptor del paquet i-èsim, el valor c és el nombre de posicions de memòria a un processador, tenim

Lema 12.5. *Per a qualsevol $\alpha \in \mathbb{R}$ prefixat, si el nombre de posicions de memòria en un processador és $O(\log n)$, la congestió en la simulació d'un pas és també $O(\log n)$.*

Volem demostrar que si la congestió del pas actual és $O(\log n)$, amb alta probabilitat l'últim paquet arribarà a la seva destinació en $O(\log n)$ passos d'encaminament. L'estratégia és demostrar que quan un paquet s'endarrereix molt, l'encaminament presenta una estructura combinatòria molt particular, i llavors podem fitar la seva probabilitat. Primer introduirem una notació més acurada pel procés d'encaminament. Imaginem que a temps $0, 1, \ldots$ la xarxa és estable, i canvia d'estat en transicions de t a $t+1$. En una transició, cada processador mou un paquet d'una cua, transmet un paquet als seus veïns i es descarrega, si cal, dels paquets fantasmes, com ja hem descrit abans. Definim el **retard** d'un paquet x al temps t com $t \Leftrightarrow i$, on i és la columna del processador que té x al temps t. Direm que un paquet x *espera* el processador u en la transició de t a $t+1$ si x és a u a t i a $t+1$. Direm que un vèrtex v és un *predecessor* del vèrtex u si tenim un camí en G'_n de v a u.

Lema 12.6. *Suposem que a temps t, algun paquet real x té retard $l > 0$ i resideix en un processador a nivell $< 2k$. Llavors, en algun instant abans de t, un altre paquet real amb clau menor va tenir retard $l \Leftrightarrow 1$ i va residir en un predecessor d'u.*

Demostració. Com que inicialment tots els paquets tenen retard 0 i a mesura que passa el temps el seu retard només s'incrementa quan el paquet espera en un processador, x ha hagut d'esperar en algun vèrtex. Si refem la història de x pas a pas cap enrere, arribarem a un temps t' de manera que, passant de t' a $t'+1$ x, va esperar en un vèrtex v. És clar que v és un predecessor d'u; a més, el retard de x al temps t' era $l \Leftrightarrow 1$.

L'única possibilitat que x s'esperi a v és que en aquesta transició v passi un altre paquet y amb clau menor que la de x. Observem que el paquet y ha de tenir també retard $l \Leftrightarrow 1$. Si y és un paquet real, ja hem acabat. Si y és un paquet fantasma, continuem cap enrere fins al temps t'' on el paquet y es va crear (no duplicat) i va deixar un vèrtex w en la transició de t'' a $t''+1$. Observem que w també és un predecessor d'u. A més, en la mateixa transició, w va enviar un paquet real z per l'altra connexió i, com que un paquet fantasma mai no espera, z va tenir retard $l \Leftrightarrow 1$ a temps t''. □

Imaginem que algun paquet té retard L quan arriba a la columna $2k$. Utilitzant el lema anterior podem construir un camí a G'_n, format per paquets en ordre creixent de claus. Per formalitzar aquesta idea primer direm que un camí és **maximal** si té longitud $2k$.

Definició 12.1. *Donats $r, L \in \mathbb{N}$, i r camins maximals $p_1, \ldots, p_r$ a G'_n, una* **seqüència de retards** *de longitud L és una seqüència $s_1, \ldots, s_L$ d'enters diferents dos a dos, amb $1 \leq s_i \leq r$ per $i = 1, \ldots, L$, tal que per algun camí maximal $p = v_0, \ldots, v_k$ es pot construir una seqüència ordenada d'enters $0 \leq t_1 \leq \cdots \leq t_L \leq 2k$, de manera que p_{s_i} conté v_{t_i} per $i = 1, \ldots, L$.*

Com que un paquet que no hagi arribat a la seva destinació a temps $2k+L$ té un retard de L, aplicant reiteradament el lema anterior tenim el resultat següent.

Lema 12.7. *Siguin $x_1 \ldots, x_n$ els paquets actuals encaminats al pas 3, i siguin κ_i la clau de x_i i p_i el camí pel qual s'encamina x_i, per $i = 1, \ldots, r$. Suposem que algun paquet no ha arribat a la seva destinació a temps $2k + L$, per $L \in \mathbb{N}$. Llavors existeix una seqüència de retard $s_1, \ldots, s_L$ per $p_1, \ldots, p_r$ tal que $\kappa_{s_i} < \kappa_{s_{i+1}}$, per $i = 1, \ldots, L \Leftrightarrow 1$.*

Com que les etiquetes dels paquets s'eligeixen independentment, una seqüència de retard $s_1, \ldots, s_L$ donada per al conjunt de camins seleccionats per l'algorisme d'encaminament té probabilitat $1/(L!)$ de satisfer, a més, les condicions addicionals que les claus siguin diferents. Per tant, la probabilitat que la fase 3 necessiti més de $2k + L$ passos es pot fitar per $1/(L!)$ multiplicat pel nombre de seqüències de retard. Un simple exercici de comptar ens dóna el següent.

Lema 12.8. *Siguin $r, L \in \mathbb{N}$, i $p_1, \ldots, p_r$ camins maximals a G'_n amb congestió c. Llavors el nombre de seqüències de retard amb longitud L per $p_1, \ldots, p_r$ és, com a molt,*

$$2^{3k} \binom{2k + L}{L} (2c)^L.$$

Per al resultat següent basta tenir en compte les desigualtats elementals $L! \geq (L/e)^L$, per a tot L, i $\binom{T}{L} \leq (T\,e/L)^L$ per a tot $T, L \in \mathbb{N}$ i tenim el lema següent.

Lema 12.9. *Si la congestió en el pas actual és $O(\log n)$, per a qualsevol constant $\alpha > 0$, si $K = n^{\lceil \alpha+3 \rceil}$, la fase 3 es completarà després de $O(\log n)$ passos amb probabilitat almenys $d'1 \Leftrightarrow n^{-\alpha}$.*

Posant-ho tot junt obtenim el resultat desitjat.

Teorema 12.1. *Sigui M' una* PRAM-EREW *amb n processadors i $m \geq n$ cel·les de memòria compartida; a més, $n = k2^k$ per algun $k \in \mathbb{N}$ i m és polinòmic en n. Llavors, si les variabeles de M' es distribueixen als processadors d'una papallona amb n processadors, d'acord amb una funció d'assignació h que segueix l'esquema presentat i que és coneguda per tots els n processadors de M. Un pas de M' es pot simular amb $O(\log n)$ passos de M amb probabilitat almenys $d'1 \Leftrightarrow n^{-\alpha}$, per a qualsevol $\alpha > 0$. A més, l'espai local que necessitem és de $O(m)$ cel·les.*

12.5 La simulació d'un algorisme

Per finalitzar, l'esquema general de la simulació es dóna com a algorisme 33. Observem que anirem simulant passos de la PRAM; si detectem que una simulació triga massa temps, entrem en un procés d'emergència en el què es recalcula tota l'assignació de memòria. És clar que aquest tipus de recomputació ens portarà molt temps, però com que l'emulació d'un pas s'efectua amb probabilitat alta, és d'esperar que no s'hagi de fer massa sovint.

Algorisme 33 Emulació d'una PRAM a una xarxa papallona.

SIMULA

1. El processador P_1 selecciona un funció de *hashing* $h \in \mathbb{H}$ i la distribueix a tots els processadors.

   repetir

2. El pas següent de la PRAM, anomenat pas actual, se simula per ck passos de la xarxa, com s'ha descrit abans.

3. Cada processador, mitjançant el còmput d'una funció OR global, determina si la simulació del pas actual s'ha completat. En cas que això sigui cert, es continua al punt 2 amb el pas següent de la PRAM.

4. En cas contrari, P_1 torna a seleccionar una funció de *hash* i es retorna al pas 2, amb el mateix pas de la PRAM.

   fins que tots els passos de la PRAM s'hagin simulat.

12.6 Referències bibliogràfiques

La simulació de la PRAM per la papallona segueix fidelment les notes manuscrites dels professors Hagerup i Rüb per al curs d'Algorismes Paral·lels a la Universitat des Saarlandes [HR92], on es poden trobar els detalls que manquen en aquest llibre. La simulació presentada és deguda a Ranade [Ran87]. El camp de la simulació de PRAM per diferents models de xarxes d'interconnexió ha estat molt desenvolupat, des de la *mesh* fins a la xarxa de Bennes. El llibre [Lei93] presenta molts resultats (entre ells, la simulació de Ranade). Un bon article "survey" és [Har94]. Per veure una simulació de PRAM per circuits, consulteu [SV84].

Exercicis

12:1 Què passa amb l'emulació si el nombre de processadors n és més gran que el nombre de cel·les d la memòria comuna m?

12:2 Donada una xarxa papallona com la descrita a l'emulació, descriviu com un processador donat pot conèixer el seu nombre de fila i de columna en $O(\log n)$ passos?

12:3 L'emulació, tal com ha estat explicada utilitza $2n$ processadors. Seguint el mateix esquema, és possible utilitzar exactament n processadors, sense incrementar el nombre de passos en més d'un factor constant?

Bibliografia

[ADKP89] K. Abrahamson, N. Dadoun, D. Kirkpatrick, i K. Przytycka. "A simple parallel tree contraction algorithm". *Journal of Algorithms*, 10:287–302, 1989.

[Akl89] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Nova York, 1989.

[AKS83] M. Ajtai, J. Komlos, i E. Szemeridi. "Sorting in $c \log n$ parallel steps". *Combinatorica*, 3:1–19, 1983.

[AS87] B. Awerburch i Y. Shiloach. "New connectivity and MSF algorithms for ultracomputer and PRAM". *IEEE Transactions on Computers*, 36:1258–1263, 1987.

[BDG88] J. L. Balcázar, J. Díaz, i J. Gabarró. *Structural Complexity I.* Springer-Verlag, Heidelberg, 1988.

[BDG90] J. L. Balcázar, J. Díaz, i J. Gabarró. *Structural Complexity II.* Springer-Verlag, Heidelberg, 1990.

[Bre73] R.P. Brent. "The parallel evaluation of general arithmetic expressions". *Journal of the ACM*, 21:201–208, 1973.

[CKP+93] D. Culler, R.M. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, T. Subramonian, i T. von Eicken. "Logp: Towards a realistic model of parallel computation", a: *ACM SIGPLAN Symposium on Principles i Practice of Parallel Programming*, pàg. 478–491. ACM Press, 1993.

[CLC82] F.Y. Chin, J. Lam, i I. Chen. "Efficient parallel algorithms for some graph problems". *Comunications of the ACM*, 25:659–665, 1982.

[CLR89] T. H. Cormen, Ch. Leiserson, i R. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1989.

[Col88] R. Cole. "Parallel merge sort". *SIAM Journal on Computing*, 17:770–785, 1988.

[Coo71] S. Cook. "The complexity of theorem-proving procedures", a: *3rd. ACM Symposium on the Theory of Computing*, pàg. 151–158. ACM Press, 1971.

[CV86] R. Cole i U. Vishkin. "Approximate and exact parallel scheduling with applications to list, tree and graphs problems", a: *27th. IEEE Symposium on Foundations of Computer Science*, pàg. 478–491, 1986.

[EK72] J. Edmonds i R. M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". *Journal of the ACM*, 19:248–267, 1972.

[Eve79] S. Even. *Graph Algorithms.* Computer Science Press, Potomac, Md, 1979.

[FK80] M.J. Foster i H.T. Kung. "The design of special-purpose VLSI". *Computer*, 71:26–40, 1980.

[FW78] S. Fortune i J. Wyllie. "Parallelism in random access machines", a: *10th ACM Symposium on Theory of Computation*, pàg. 114–118. ACM Press, 1978.

[GGK+83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, i M. Snir. "The NYU Ultracomputer: Designing an MIMD shared memory parallel computer". *IEEE Transactions on Computers*, C-32:175–189, 1983.

[GHR95] R. Greenlaw, H.J. Hoover, i W.L. Ruzzo. *Limits to parallel computation: P-completeness theory.* Oxford University Press, 1995.

[GJ79] M. R. Garey i D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[GR88] A. Gibbons i W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[GR89] A. Gibbons i W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context free recognition. *Information and Computation*, 81(1):32–45, abril de 1989.

[GR90] A. Gibbons i W. Rytter. Optimal edge-colouring outerplanar graphs is in NC. *Theoretical Computer Science*, 71:401–411, 1990.

[Hag88] T. Hagerup. *Parallel algorithms on planar graphs.* Universitat des Saarlandes, 1988. [Informe tècnic]

[Har94] T.J. Harris. "A survey of PRAM simulation techniques". *ACM Computer Surveys*, 26:187–205, 1994.

[HCS79] D.S. Hirschberg, A.K. Chandra, i D.V. Sarwate. "Computing connected components on parallel computers". *Communications of the ACM*, 22:461–464, 1979.

[HR92] T. Hagerup i C. Rub. *Randomized simulation of PRAMSs on processor networks.* Universitat des Saarlandes, 1992. [Informe tècnic]

[JáJ92] J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley, Reading, Mass., 1992.

[JGD87] L.H. Jemieson, D. Gannon, i R.J. Douglas. *The characteristics of Parallel Algorithms.* MIT Press, Cambridge, 1987.

[KR90] R.M. Karp i V. Ramachandran. "Parallel algorithms for shared memory machines", a: Jan van Leewen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pàg. 869–942. Elsevier Science Publishers, Amsterdam, 1990.

[Kuc82] L. Kučera. "Parallel computation and comflicts in memory access". *Information Processing Letters*, 14:93–96, 1982.

[Kun89] H.T. Kung. "The structure of parallel algorithms", a: Yovitz, editor, *Advances in Computer*, pàg. 65–112, New York, 1989. Academic Press.

[Lei93] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, Calif., San Mateo, CA., 1993.

[MC82] T.A. Marsland i M. Campbell. "Parallel search of strongly ordered game trees". *Computer Surveys*, 14:533–551, 1982.

[McC93] W.F. McColl. "General purpose parallel computers", a: A. Gibbons i P. Spirakis, editors, *Lectures on Parallel computation*, pàg. 337–391. Cambridge University Press, 1993.

[MRK88] G.L. Miller, V. Ramachandran, i E. Kaltofen. "Efficient parallel evaluation of straight-line code and arithmetic circuits". *SIAM Journal on Computing*, 17:687–695, 1988.

[NM82] D. Nath i S.N. Maheshwari. "Parallel algorithms for the connected components and minimal spanning trees". *Information Processing Letters*, 14:7–11, 1982.

[PS82] C. Papadimitriou i K. Steiglitz. *Combinatorial Optimizations, Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Qui87] M.J. Quinn. *Designing efficient algorithms for Parallel Computers.* McGraw-Hill, Nova York, 1987.

[Ran87] A.G. Ranade. "How to emmulate shared memory", a: *28th IEEE Symposium on Foundations of Computer Science*, pàg. 185–194, 1987.

[Rei93] J. Reif. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, Nova York, 1993.

[Rom89] F. Romani. "Shortest path problem is not harder than matrix multiplication". *Information Processing Letters*, 32:199–204, 1989.

[Ser90] M. J. Serna. *The parallel approximability of P-complete problems*. Departament LSI, Universitat Politècnica de Catalunya, 1990. [Tesi doctoral]

[SJ81] J. Savage i J. JáJá. "Fast efficient parallel algorithms for some graph problems". *SIAM Journal on Computing*, 10:682–691, 1981.

[Sni85] M. Snir. "On parallel searching". *SIAM Journal on Computing*, 14:688–707, 1985.

[SV84] L. Stockmeyer i U. Vishkin. "Simulation of parallel random access machines by circuits". *SIAM Journal of Computing*, 13:409–422, 1984.

[TK91] P. de la Torre i C.P. Kruskal. "Towards a single model of efficient computation in real parallel machines", a: J. van Leewen, editor, *PARLE-91*, volume 506 of *Lecture Notes in Computer Science*, pàg. 6–24. Springer-Verlag, 1991.

[Tor93] J. Toran. "P-completeness", a: A. Gibbons i P. Spirakis, editors, *Lectures on Parallel Computation*, pàg. 177–196, Cambridge, 1993. Cambridge University Press.

[TV85] R.E. Tarjan i U. Vishkin". "Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing*, 14:862–874, 1985.

[Val75] L.G. Valiant. "Parallelism in comparison problems". *SIAM Journal on Computing*, 4:348–355, 1975.

[Val90] L.G. Valiant. "A bridging model for parallel computation". *Communications of the ACM*, 33:103–111, 1990.

[Val91] L.G. Valiant. "General purpose parallel architectures", a: Jan van Leewen, editor, *Handbook of Theoretical Computer Science. Vol. A*, pàg. 945–971. North-Holland, 1991.

[VSBR83] L.G. Valiant, S. Skyum, S. Berkowitz, i C. Rackoff. "Fast parallel computation of polynomial using few processors". *SIAM Journal on Computing*, 12:641–644, 1983.

Índex de figures

Índex d'algorismes

Índex de matèries