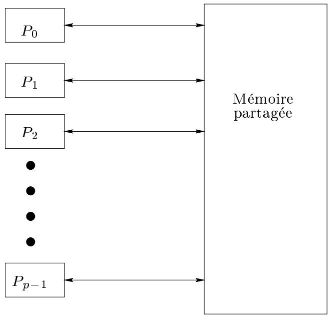
Machines P-RAM et Réseaux de tri

Plan

1 Introduction

Un modèle pour le parallélisme ?
- Machine de Turing → coût d'un algorithme séquentiel
- Grande variété des architectures parallèles
- Comment définir le coût d'une communication ?
- **Ignorer les communications ! !**
- Modèle imparfait mais classification des algorithmes parallèles
- Déterminer le parallélisme **maximal** contenu dans un algorithme

Machine P-RAM



- Mémoire centrale partagée (de taille non bornée)
- Nombre de processeurs non borné
- Les processeurs accèdent à n'importe quelle case de la mémoire centrale en une unité de temps
- Accès simultané de plusieurs processeurs à autant de cases mémoire distinctes possible en un seul pas de temps.

Accès concurrent à la même case

CREW (Concurrent Read Exclusive Write)
- Modèle le plus usuel
- Nombre de lectures simultanées d'une même case non borné
- Accès en écriture par un seul processeur à la fois.

CRCW (Concurrent Read Concurrent Write)
- Modèle le plus puissant
- Nombre d'écritures simultanées dans une même case non borné
- Variantes :

1. mode consistant : tous les proc. écrivent la même valeur
2. mode arbitraire : valeur du dernier processeur qui écrit
3. mode fusion : calcul au vol d'une fonction commutative et associative (ET ou un OU booléens, maximum ou somme d'entiers, etc)

EREW (Exclusive Read Exclusive Write)
- Modèle le plus restrictif
- Le plus proche des machines réelles.

2 Techniques de pointeurs

(A) Calcul du rang

Problème

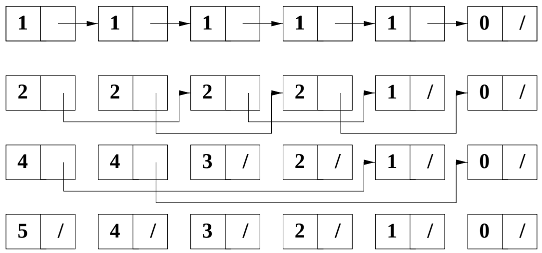
– Liste chaînée L de taille n
– Calculer pour chaque élément i sa distance $d[i]$ à la fin de la liste :

$$d[i] = \begin{cases} 0 & \text{if } next[i] = \text{NIL} \\ d[next[i]] + 1 & \text{if } next[i] \neq \text{NIL} \end{cases}$$

– Séquentiel : propager les valeurs d, temps $O(n)$
– P-RAM : un processeur P_i par élément i, temps $O(\log n)$

Algorithme

Initialisation
  for each processor i in // **do**
    if $next[i] = \text{NIL}$ **then** $d[i] = 0$ **else** $d[i] = 1$
Boucle principale
  while ($\exists$ objet i t.q. $next[i] \neq \text{NIL}$) **do**
    for each processor i in // **do**
      if $next[i] \neq \text{NIL}$ **then** $\begin{cases} d[i] \leftarrow d[i] + d[next[i]] \\ next[i] \leftarrow next[next[i]] \end{cases}$



Correction

– Principe : chaque pas dédouble les listes courantes en listes des objets en positions paires – positions impaires
  $\Rightarrow$ Taille de chaque liste réduite de moitié à chaque étape
  $\Rightarrow$ Nombre d'itérations égal à $\lceil \log n \rceil$

– A chaque étape, les mises à jour doivent être synchronisées
– Sémantique d'une boucle **forall** parallèle :

$$\begin{array}{lcl} \textbf{forall } i \textbf{ do} & & \textbf{for } i = 1 \textbf{ to } n \textbf{ do} \\ \quad A(i) = B[i] & \Longleftrightarrow & \quad A(i) = temp[i] \\ & & \textbf{for } i = 1 \textbf{ to } n \textbf{ do} \\ & & \quad temp[i] = B[i] \end{array}$$

– Test de la condition globale :
  Remplacer : **while** ($\exists$ objet i t.q. $next[i] \neq$ NIL) **do**
  par : **for** étape $= 1$ **to** $\lceil \log n \rceil$

Type de P-RAM : EREW

– Chaque processeur P_i écrit dans $d(i]$ et $next[i]$, en mode exclusif
– P_i et P_j, où $next[j] = i$, accèdent simultanément à $d[i]$ en lecture
– Solution : remplacer $d[i] \leftarrow d[i] + d[next[i]]$
  par :
$$temp[i] \leftarrow d[next[i]]$$
$$d[i] \leftarrow d[i] + temp[i]$$

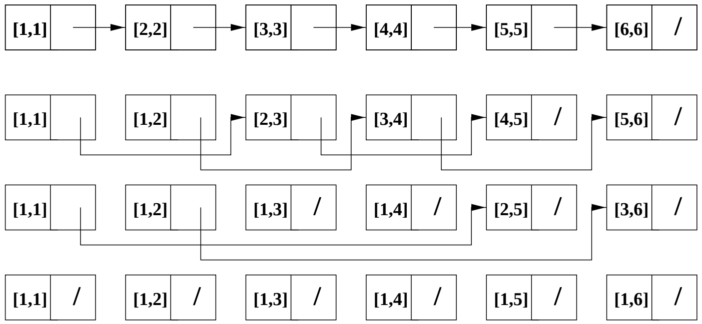
(B) Calcul des préfixes

Problème

– Suite donnée $(x_1, x_2, \ldots, x_n)$
– $\otimes$ opération binaire associative quelconque
– Calculer $(y_1, y_2, \ldots, y_n)$, $y_k == x_1 \otimes x_2 \otimes \ldots \otimes x_k$

Algorithme

for each processor i in // **do**
  $y[i] \leftarrow x[i]$
  while ($\exists$ objet i t.q. $next[i] \neq$ NIL) **do**
    for each processor i in // **do**
      if $next[i] \neq$ NIL **then** $\begin{cases} y[next[i]] \leftarrow y[i] \otimes y[next[i]] \\ next[i] \leftarrow next[next[i]] \end{cases}$



$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ pour $i \leq j$

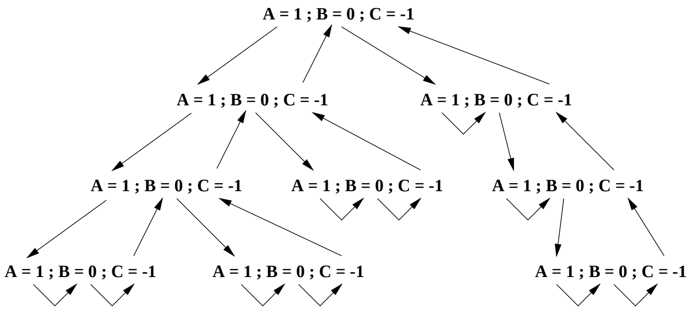
(C) Tour d'Euler

Problème

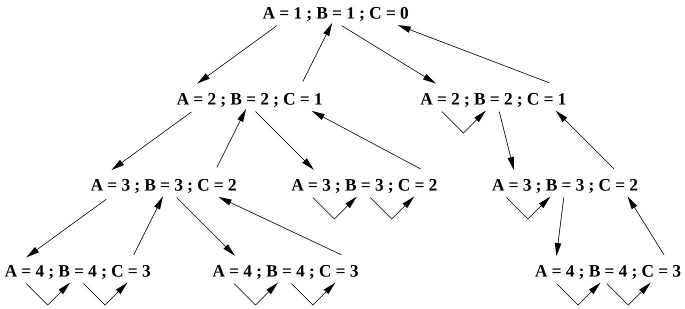
– Dans un arbre binaire à n sommets, calculer la hauteur de chaque sommet, i.e. sa distance à la racine
– Chaque sommet i à trois champs de données $parent[i]$, $left[i]$ et $right[i]$

Solution

– Tour d'Euler
– Coût en $O(\log n)$, quelle que soit la hauteur de l'arbre.



(a) Initialisation



(b) Après calcul des préfixes

3 Simulation

Calcul du maximum

– Calcul en temps constant du maximum d'un tableau A de n éléments
– P-RAM CRCW avec $O(n^2)$ processeurs, mode consistant

for all $i = 1$ **to** n in // **do**
  $m[i] \leftarrow$ TRUE
for all $i, j = 1$ **to** n in // **do**
  if $A[i] < A[j]$ **then** $m[i] \leftarrow$ FALSE
for all $i = 1$ **to** n in // **do**
  if $m[i] =$ TRUE **then** $max \leftarrow A[i]$

Séparation des modèles CRCW et CREW

– Une P-RAM CRCW plus *puissante* qu'une P-RAM CREW ?
– Trouver un algorithme tel que, à nombre de processeurs égal, temps(P-RAM CRCW) < temps(P-RAM CREW)
– P-RAM CRCW de n processeurs : calcul en temps constant de la fusion $\bigotimes_{1 \leq i \leq n} e_i$ de n éléments $(e_1, \ldots, e_n)$, où $\otimes$ est une loi associative
– P-RAM CREW de n processeurs : au moins $O(\log n)$ étapes : en temps constant fusion d'au plus un nombre constant d'éléments en une seule valeur.

Séparation des modèles CREW et EREW

– Déterminer si un élément e donné se trouve dans un ensemble de n éléments $(e_1, \ldots, e_n)$ tous distincts.

– Temps constant sur une P-RAM CREW avec n processeurs :
(i) initialiser un booléen à faux : $res \leftarrow$ FALSE
(ii) en parallèle, chaque processeur i compare e à e_i (lectures simultanées) et positionne $res \leftarrow$ TRUE si égalité
NB : les e_i sont différents deux à deux $\rightarrow$ mode écriture exclusive.

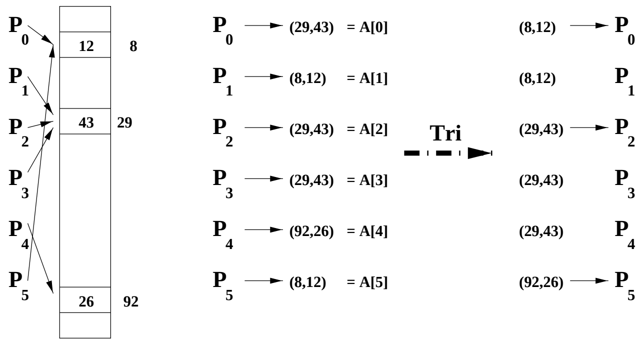
– Temps $\Omega(\log n)$ sur une P-RAM CREW avec n processeurs :
comparaisons en temps constant $\rightarrow$ nombre de copies de e de l'ordre de $n \rightarrow O(\log n)$ pas de calcul.

– Diffusion d'une information à n processeurs
$\rightarrow O(\log n)$ étapes en EREW :
le nombre de processeurs "au courant" au plus "double" à chaque étape.

Théorème de simulation

– Tout algorithme sur une machine P-RAM CRCW à p processeurs ne peut pas être plus de $O(\log p)$ fois plus rapide que le meilleur algorithme P-RAM EREW à p processeurs pour le même problème.

Preuve



(a) mode CRCW **(b) simulation EREW**

– Simulation en mode exclusif d'écritures concurrentes, mode consistant
– P_i mode CRCW écrit x_i à l'adresse l_i
$\Rightarrow P_i$ mode EREW effectue l'écriture (exclusive) $A[i] = (l_i, x_i)$
– Tri de p données avec $O(p)$ processeurs EREW en temps $O(\log p)$
$\Rightarrow$ tri de A suivant la première coordonnée
– P_i mode EREW inspecte $A[i] = (l_j, x_j)$ et $A[i-1] = (l_k, x_k)$
et écrit (exclusif) x_j à l'adresse l_j si $l_j \neq l_k$ ou $i = 0$.

Théorème de Brent

– Soit A algorithme comportant un nombre total de m opérations et qui s'exécute en temps t sur une P-RAM (avec un nombre de processeurs indéterminé)
– Alors on peut simuler A en temps $O(\frac{m}{p}+t)$ sur une P-RAM de même type avec p processeurs.
– Preuve : à l'étape i, A effectue $m(i)$ opérations ($\sum_{i=1}^{n} m(i) = m$)
Simulation avec p processeurs en temps $\lceil \frac{m(i)}{p} \rceil \leq \frac{m(i)}{p} + 1$.

Exemple d'application

– Calcul du maximum sur P-RAM EREW en temps $O(\log n)$
– Idée : arbre binaire, $O(n)$ processeurs
– $n = 2^m$, A de taille $2n$, maximum des n éléments de A en position $A[n], A[n+1], \ldots, A[2n-1]$, résultat dans $A[1]$ après exécution :

for $k = m-1$ **downto** O **step** -1 **do**
  for all j, $2^k \leq j \leq 2^{k+1}-1$ in // **do**
    $A[j] \leftarrow \max(A[2j], A[2j+1])$

– Avec p processeurs, simulation en temps $O(\frac{n}{p} + \log n)$
– Choisir $p = \frac{n}{\log n}$: même temps d'exécution, moins de processeurs !

Travail et efficacité

– P problème de taille n à résoudre
– $T_{seq}(n)$ temps du meilleur algorithme séquentiel (connu)
– Algorithme P-RAM en temps $T_{par}(p)$ avec p processeurs :
Speed-up $S_p = \frac{T_{seq}(n)}{T_{par}(p)}$

Efficacité $e_p = \frac{T_{seq}(n)}{p.T_{par}(p)}$

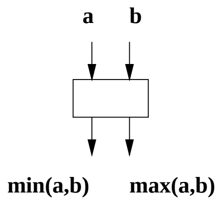
Travail $W_p = p.T_{par}(p)$

– Le travail se conserve par simulation
– A algorithme en temps t sur une P-RAM avec p processeurs
$\rightarrow$ simuler A sur une P-RAM de même type avec $p' \leq p$ processeurs, en temps $O(\frac{t.p}{p'})$.
– Algorithme *efficace* $\Longleftrightarrow$ travail de l'ordre de la complexité séquentielle

4 Tri sur réseau linéaire

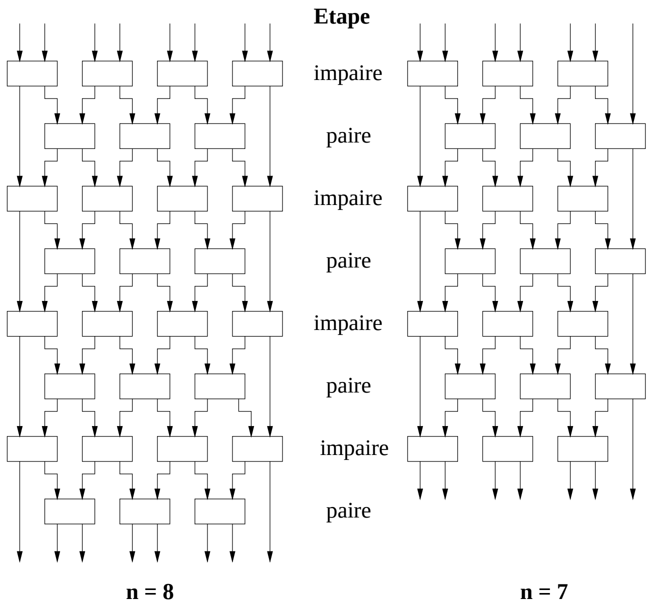
Réseaux de tri

– Modèle de calcul plus réaliste, mais moins général, que les P-RAM
– Disposer et interconnecter des comparateurs afin de trier des listes



– Architecture qui ne dépend que de la taille des listes, et non des valeurs de leurs éléments.

Réseau de transposition pair-impair



– Implante correctement le tri !
– Temps de calcul $t_n = n$
– Comparateurs $p_n = n(n-1)/2$
– Travail $W_n = O(n^3)$

Tri pair-impair sur réseau linéaire de processeurs

– Tri-fusion pair-impair sur réseau linéaire de p processeurs
– Chaque processeur trie sa sous-suite de taille n/p en temps $O(\frac{n}{p}\log n)$
– p étapes d'échanges alternés avec fusion, chacune de temps $O(n/p)$

	P_1		P_2		P_3		P_4		P_5		P_6
init	{8,3,12}		{10,16,5}		{2,18,9}		{17,15,4}		{1,6,13}		{11,7,14}
tri local	{3,8,12}		{5,10,16}		{2,9,18}		{4,15,17}		{1,6,13}		{7,11,14}
impair	{3,5,8}	$\leftrightarrow$	{10,12,16}		{2,4,9}	$\leftrightarrow$	{15,17,18}		{1,6,7}	$\leftrightarrow$	{11,13,14}
pair	{3,5,8}		{2,4,9}	$\leftrightarrow$	{10,12,16}		{1,6,7}	$\leftrightarrow$	{15,17,18}		{11,13,14}
impair	{2,3,4}	$\leftrightarrow$	{5,8,9}		{1,6,7}	$\leftrightarrow$	{10,12,16}		{11,13,14}	$\leftrightarrow$	{15,17,18}
pair	{2,3,4}		{1,5,6}	$\leftrightarrow$	{7,8,9}		{10,11,12}	$\leftrightarrow$	{13,14,16}		{15,17,18}
impair	{1,2,3}	$\leftrightarrow$	{4,5,6}		{7,8,9}	$\leftrightarrow$	{10,11,12}		{13,14,15}	$\leftrightarrow$	{16,17,18}
pair	{1,2,3}		{4,5,6}	$\leftrightarrow$	{7,8,9}		{10,11,12}	$\leftrightarrow$	{13,14,15}		{16,17,18}

– Temps de calcul $O(\frac{n}{p}\log n + n)$
– Travail $O(n(p + \log n))$: **optimal** pour $p \leq \log n$.

Machines P-RAM et Réseaux de tri
TD – Enoncé

Plan

1. Tri-fusion de Batcher
2. Principe du 0-1
3. Tri sur une grille 2D
4. Machine à trier P-RAM de Cole

1 Tri-fusion de Batcher

Réseau de fusion

– On note $\textsc{tri}(c_1, c_2 \ldots, c_n)$ la liste triée des c_i
– On écrit $\textsc{triée}(c_1, c_2 \ldots, c_n)$ pour $c_1 \leq c_2 \leq \cdots \leq c_n$
– Si $\textsc{triée}(a_1, \ldots, a_n)$ et $\textsc{triée}(b_1, \ldots, b_n)$, on note

$$\textsc{fusion}((a_1, \ldots, a_n), (b_1, \ldots, b_n)) = \textsc{tri}(a_1, \ldots, a_n, b_1, \ldots, b_n) .$$

1. Concevoir un réseau $\underline{\textsc{fusion}_1}$, ($\underline{\textsc{fusion}_2}$) à partir de comparateurs pour la fusion de deux listes triées de longueur 2 (2^2).
2. Montrer la propriété suivante :
   $\textsc{triée}(a_1, \ldots, a_{2n})$ et $\textsc{triée}(b_1, \ldots, b_{2n})$
   $$\begin{array}{rcl} (d_1, \ldots, d_{2n}) & = & \textsc{fusion}((a_1, a_3, \ldots, a_{2n-1}), (b_1, b_3, \ldots, b_{2n-1})) \\ (e_1, \ldots, e_{2n}) & = & \textsc{fusion}((a_2, a_4, \ldots, a_{2n}), (b_2, b_4, \ldots, b_{2n})) \end{array}$$
   $\Rightarrow$
   $$\textsc{triée}(d_1, \min(d_2, e_1), \max(d_2, e_1), \ldots, \min(d_{2n}, e_{2n-1}), \max(d_{2n}, e_{2n-1}), e_{2n})$$
3. Définir par récurrence un réseau $\underline{\textsc{fusion}_n}$ pour la fusion de deux listes de longueur 2^n. Combien de comparateurs comprend ce réseau, et quel en est le temps de calcul ?

Réseau de tri

– Construire par récurrence un réseau $\underline{\textsc{tri}_n}$ pour trier 2^n éléments
– Analyser la complexité (temps de calcul, nombre de processeurs, coût).

2 Principe du 0-1

Enoncé

– Montrer qu'un réseau de comparateurs implante correctement le tri si et seulement si il calcule bien cette fonction pour toute suite de type 0-1 (i.e. constituée uniquement de 0 et de 1) en entrée
– En déduire une nouvelle preuve de la correction du réseau de fusion pair-impair de Batcher

3 Tri sur une grille 2D

Définitions

– On s'intéresse au tri par transposition pair impair sur une grille

– Un tableau carré $A = ((a_{i,j}))$ de taille $n \times n$, $n = 2^m$ est ordonné en serpent si les éléments du tableau sont ordonnés comme suit :

$$a_{2i,j} \leq a_{2i,j+1}, \quad \text{si} \quad 1 \leq j \leq n-1$$

$$a_{2i+1,j+1} \leq a_{2i+1,j}, \quad \text{si} \quad 1 \leq j \leq n-1$$

$$a_{2i-1,n} \leq a_{2i,n}, \quad \text{si} \quad 1 \leq i \leq 2^{m-1}$$

$$a_{2i,1} \leq a_{2i+1,1}, \quad \text{si} \quad 1 \leq j \leq 2^{m-1}-1$$

– Ce serpent induit un réseau linéaire à l'intérieur de la grille

– Un *shuffle* transforme la séquence de $n = 2p$ éléments $< z_1, \ldots, z_n >$ en la séquence $< z_1, z_{p+1}, \ldots, z_p, z_{2p} >$.

Algorithme de fusion

– Données : 4 tableaux de taille $2^{m-1} \times 2^{m-1}$ ordonnés en serpent à fusionner en un tableau de taille $2^m \times 2^m$ ordonné en serpent

– 1. *Shuffle* de chaque ligne du tableau en utilisant les transpositions pair-impair sur chaque ligne (ce qui revient à appliquer la transformation *shuffle* sur les colonnes)

– 2. Trier les paires de colonnes, c'est à dire les tableaux de taille $n \times 2$ en ordre serpent, en utilisant $2n$ étapes de transposition pair-impair sur le réseau linéaire induit sur chaque serpent de longueur $2n$

– 3. Appliquer $2n$ étapes de transposition pair-impair sur le réseau linéaire induit par le serpent de taille n^2.

Questions

1. Faire tourner l'algorithme de tri induit avec $n = 4$ et $a_{i,j} = 21 - 4i - j, \quad 1 \leq i, j \leq 4$

2. Montrer que la première étape de l'algorithme peut s'effectuer en temps $2^{m-1}-1$, l'unité étant un échange entre voisins. En déduire que l'algorithme global de fusion s'effectue en temps $\leq 4.5n$

3. En supposant l'algorithme de fusion correct, construire un algorithme qui trie une séquence de longueur 2^{2m} sur une grille $2^m \times 2^m$. Estimer sa complexité

4. Montrer que le tri par transposition pair-impair sur une grille est correct : il s'agit de montrer que $2n$ étapes de transposition pair-impair dans la troisième phase de l'algorithme de fusion suffisent à obtenir un serpent correctement ordonné.

4 Machine à trier P-RAM de Cole

Enoncé

– Lire le polycopié sur la machine à trier P-RAM de Cole
– Comprendre le principe du fonctionnement en pipeline
– Comprendre comment fusionner des listes en temps constant à l'aide de listes auxiliaires
– Admirer la beauté algorithmique de la construction :-)

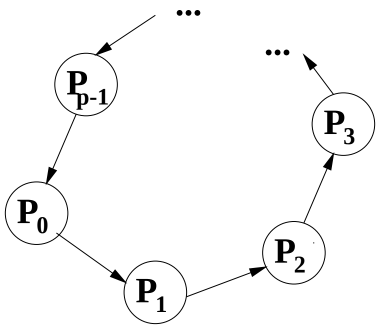
Algorithmique sur anneau de processeurs

Plan

1. Macro-communications sur un anneau
2. Produit matrice-vecteur
3. Algorithmes de balayage d'image

1 Macro-communications sur un anneau

Architecture



– p processeurs arrangés en un anneau orienté, numérotés de 0 à $p-1$
– Numéro logique : q ← my_num()
– Nombre total de processeurs : tot_proc_num().

Fonctionnement SPMD
– Tous les processeurs exécutent le même programme
– Ils opèrent sur des données présentes dans leur mémoire locale
– Envoyer un message à son successeur :

send(adr, L)

adr adresse mémoire du processeur émetteur de la première valeur
L est la longueur du message
– Recevoir un message : receive(adr, L)

– Il doit y avoir un receive "en face de" chaque send
– Adresses locales :

```
q ← mynum();
if (q=0) then send(adr1, L);
if (q=1) then receive(adr2, L);
```


Sémantique
– Hypothèse restrictive : send et receive bloquants
– Hypothèse classique : receive bloquant, send non bloquant
– Hypothèse récente : les deux primitives sont non bloquantes
→ envoyer, recevoir et calculer simultanément, si les données ne sont pas partagées en écriture
→ trois *threads* de contrôle, une pour le calcul, et deux dédiées aux communications asynchrones

Coût d'une communication
– Difficile à modéliser
– Formule simple : envoyer/recevoir un message de longueur L coûte $\beta + L\tau$
– β coût d'initialisation (configuration du matériel, mise en oeuvre logicielle)
– τ inverse de la bande passante, mesure la vitesse du lien en régime permanent
– Distance de communication : message de P_0 à P_q → coût $q(\beta+L\tau)$

Diffusion

– Envoi par P_k d'un message de longueur L (stocké à l'adresse adr) à tous les autres processeurs
– Programme du processeur numéro q :

```
broadcast(k, adr, L) {
q ← my_num();
p ← tot_proc_num();
if q = k then
   send(adr, L);
else
   if q = k − 1 mod p then
     receive(adr, L);
   else
     receive(adr, L);
     send(adr, L);
}
```

– receive bloquant : on transmet le message dès réception
– temps de la diffusion d'un message de longueur L : $(p-1)(\beta+L\tau)$.

Diffusion personnalisée

– Envoi par P_k d'un message différent à tous les autres processeurs
– Message pour P_q à l'adresse adr[q]
– A la fin, chaque processeur a son message à l'adresse adr
– Pipeline des messages sur l'anneau
– Programme du processeur numéro q :

```
scatter(k, adr, L) {
q ← my_num();
p ← tot_proc_num();
if q = k then
   adr ← adr[k];
else
   receive(adr, L);
   for i = 1 to k − 1 − q mod p
     {{ send(adr, L) // receive(temp, L) }};
     adr ← temp;
}
```

– send non bloquant
– receive bloquant (sinon ajouter des points de synchronisation)
– temps total comme pour la diffusion, $(p-1)(\beta+L\tau)$.

2 Produit matrice-vecteur

Données

- Multiplication $y = Ax$
- Matrice A de dimension $n \times n$
- Vecteur x à n composantes (numérotées de 0 à $n-1$)
- Anneau orienté de p processeurs, $r = n/p$

Distribution

- Ensemble de n produits scalaires :

```
for i = 1 to n do
  for j = 1 to n do
    y[i] = y[i] + a[i,j] * x[j] ;
```

- Distribuer le calcul des produits scalaires aux processeurs.
- Chaque processeur a en mémoire r lignes de la matrice A, rangées dans une matrice a de dimension $r \times n$
- P_q contient les lignes qr à $(q+1)r-1$ de la matrice, et les composantes de même rang des vecteurs x et y :

```
var a : array[0..r-1,0..n-1] of real ;
var x, y : array[0..r-1] of real ;
```


Principe

$$
\begin{array}{c|c}
P_0 & \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\ \hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}
\end{array}
$$

(a) Distribution initiale des données

$$
\begin{array}{c|c}
P_0 & \begin{pmatrix} A_{00} & A_{01} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{10} & A_{11} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\ \hline
P_1 & \begin{pmatrix} \bullet & \bullet & A_{22} & A_{23} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{32} & A_{33} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \hline
P_2 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{44} & A_{45} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{54} & A_{55} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \hline
P_3 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{66} & A_{67} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{76} & A_{77} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}
\end{array}
$$

(b) Première étape

$$
\begin{array}{cccc}
P_0 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{06} & A_{07} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{16} & A_{17} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{30} & A_{31} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} \bullet & \bullet & A_{42} & A_{43} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{52} & A_{53} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{64} & A_{65} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{74} & A_{75} & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}
\end{array}
$$

(c) Deuxième étape

$$
\begin{array}{cccc}
P_0 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{04} & A_{05} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{14} & A_{15} & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{26} & A_{27} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{36} & A_{37} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{50} & A_{51} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} \bullet & \bullet & A_{62} & A_{63} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{72} & A_{73} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}
\end{array}
$$

(c) Troisième étape

$$
\begin{array}{cccc}
P_0 & \begin{pmatrix} \bullet & \bullet & A_{02} & A_{03} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{12} & A_{13} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{24} & A_{25} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{34} & A_{35} & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{46} & A_{47} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{56} & A_{57} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{70} & A_{71} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}
\end{array}
$$

(c) Quatrième étape

Algorithme

```
matrice-vecteur(A, x, y) {
q ← my_num();
p ← tot_proc_num();
for step = 0 to p − 1 do
   {{ send(x, r) // receive(temp, r) //
     for i=0 to r-1 do
       for j=0 to r-1 do
         y[i] ← y[i]+ a[i,(q - step mod p)r+j]*x[j] }};
   x ← temp;
}
```

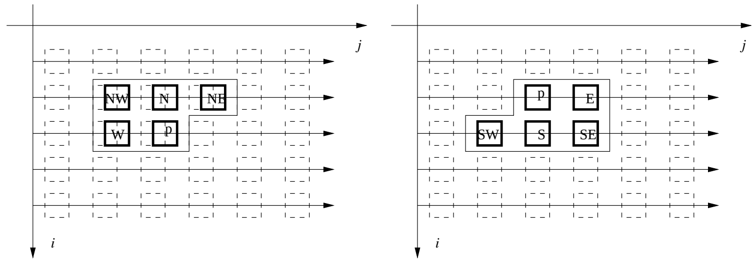
– Temps d'exécution $T(p) = p\max\{r^2\tau_a, \beta + r\tau_c)$
– Efficacité égale à 1 si n assez grand ($\frac{n^2}{p^2}\tau_a \geq \beta + \frac{n}{p}\tau_c$).

3 Algorithmes de balayage d'image

Algorithme séquentiel
– P une grille de $n \times n$ points
– Voisinage :

$$\begin{array}{ccc} NW & N & NE \\ W & \boxed{\text{p}} & E \\ SW & S & SE \end{array}$$

– Balayage en passe avant : de haut en bas et de gauche à droite :
(PAV) p := mise_à_jour_avant(p, W, NW, N, NE)
– Balayage en passe arrière : de bas en haut et de droite à gauche :
(PAR) p := mise_à_jour_arrière(p, E, SE, S, SW)



(a) balayage avant (b) balayage arrière

Distance au contour

- P image binaire : figure $F = \{+\infty\}$, entièrement entourée par son complémentaire $P \setminus F = \{0\}$
- Calcul de la distance (CD) au contour : image copie de F
  chaque pixel $\rightarrow$ sa distance (selon une certaine métrique) à $P \setminus F$
- Axe médian = lieu des centres des disques maximaux

- Balayer l'image deux fois :
  (PAV) $\quad p := \min(p, W + t_1, NW + t_2, N + t_1, NE + t_2)$
  (PAR) $\quad p := \min(p, E + t_1, SE + t_2, S + t_1, SW + t_2)$
- Coefficients t_1 et t_2 :
  $t_1 = 1$ et $t_2 = +\infty \rightarrow$ distance de Manhattan d_4
  $t_1 = t_2 = 1 \rightarrow$ distance de l'échiquier d_8
  $t_1 = 3$ et $t_2 = 4 \rightarrow$ approximation de la distance Euclidienne.

Calcul de la trajectoire optimale

- Voir polycopié

Parallélisation de PAV : Algorithme glouton

- Cas $p = n$: au top $2i + j$, P_i reçoit le pixel $(i-1, j+1)$ de P_{i-1}
  calcule (i, j) et le transmet à P_{i+1}

$$\begin{array}{l}
P_0 \ \boxed{0}\,\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}\,\boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\ \cdots \\
P_1 \ \boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}\,\boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\ \cdots \\
P_2 \ \boxed{4}\,\boxed{5}\,\boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\ \cdots \\
P_3 \ \boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\ \cdots \\
P_4 \ \boxed{8}\,\boxed{9}\ \cdots \\
\cdots\ \cdots
\end{array}$$

- Cas $p < n$: allocation entrelacée
  la ligne j est allouée au processeur j modulo p
  P_0 a besoin de valeurs calculées par P_{p-1}
  P_0 reçoit $(p-1, 0)$ au temps $2p - 1$ et $(p-1, 1)$ au temps $2p$
  $\rightarrow n \geq 2p$, sinon P_0 est inactif

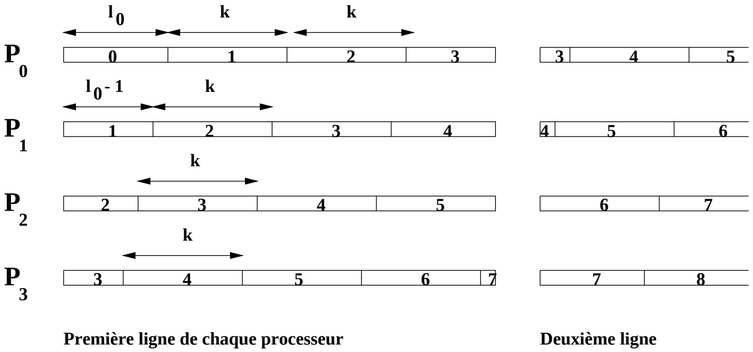
- Point fort : le délai entre deux processeurs voisins est petit
- Point faible : nombreuses communications de petite taille

Augmenter la granularité

– Utiliser de plus longs messages

– Calcul de k pixels consécutifs à chaque étape

– Condition pour le régime permanent : $n \geq (k+1)p$



– Point fort : communications moins coûteuses

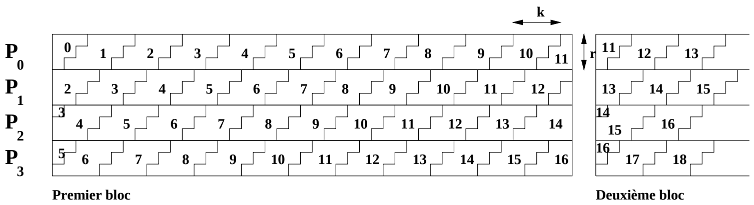
– Point faible : délai d'attente entre deux processeurs adjacents

D'autres stratégies d'allocation

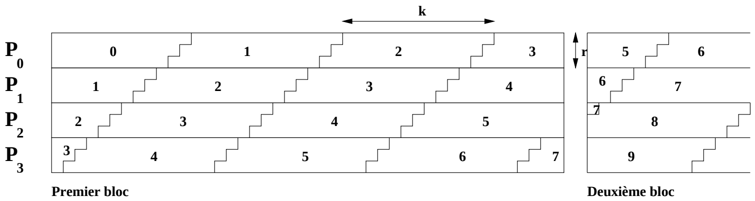
– Allouer des blocs de r lignes consécutives de manière entrelacée :

P_0	P_1	P_2	P_3
$0, 1, 2$	$3, 4, 5$	$6, 7, 8$	$9, 10, 11$
$12, 13, 14$	$15, 16, 17$	$18, 19, 20$	$21, 22, 23$
$24, 25, 26$	$27, 28, 29$	$30, 31, 32$	$33, 34, 35$
$36, 37, 38$	$39, 40, 41$	$42, 43, 44$	

– Condition pour le régime permanent : $n \geq (k+r)p$

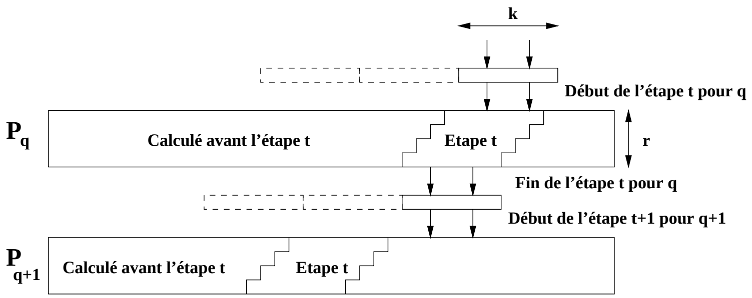


$n = 44$, $p = 4$, $r = 3$ et $k = 4$



$n = 44$, $p = 4$, $r = 3$ et $k = 13$

Analyse de complexité



- Temps séquentiel $T_{seq} = n^2 \tau_a$
- Temps d'exécution d'une étape : $\tau_{step} = \beta + k\tau_c + rk\tau_a$
- Calculer t_q, début du calcul de P_q ?
  1. P_0 calcule l_0 pixels à $t_0 = 0$
  2. P_1 calcule $l_1 = (l_0 - r) \bmod k$ pixels à $t_1 = 1 + \lfloor \frac{r - l_0}{k} \rfloor$
  3. P_q calcule $l_q = (l_0 - q * r) \bmod k$ pixels à $t_q = q + \lfloor \frac{qr - l_0}{k} \rfloor$
- Nombre total d'étapes $T_p = t_{p-1} + \lfloor (n^2/(p * r) - l_q + r - 1)/k \rfloor$
- Temps d'exécution de l'algorithme $T_{//} = \tau_{step} T_p$.
- Condition de validité ?
  1. P_0 effectue sa première réception au temps t_p
  2. A cet instant, il a déjà calculé $l_0 + k(t_p - 1)$ pixels
  3. Condition : $n - (l_0 + k(t_p - 1)) + l_p \geq k$
  4. On retrouve : $n \geq p(r + k)$

Optimisation

- Problème de taille n
- Anneau de p processeurs
- Allocation des lignes par blocs de taille r, avec $1 \leq r \leq \frac{n}{p}$
- Segments de taille k, avec $1 \leq k \leq \frac{n}{p} - r$

$$T_{//} = (\beta + k\tau_c + rk\tau_a) \left((p-1)(1 + \frac{r}{k}) + \frac{n^2}{prk} \right).$$

- $k_{opt}(r) = \min(k_{max}(r), k_{//}(r))$
- $k_{max}(r) = \frac{n}{p} - r$
- $k//(r) = \sqrt{\frac{\beta}{\tau_c + r\tau_a} \left(\frac{n^2}{p(p-1)r} + r \right)}.$

Algorithmique sur anneau de processeurs
TD – Enoncé

Plan

1. Echange total
2. Diffusion pipelinée
3. Factorisation LU

1 Echange total sur l'anneau

Enoncé

– Chaque processeur `k` veut envoyer un message à tous les autres : p diffusions simultanées

– Hypothèses :

1. Tous les messages ont même longueur L
2. Au départ, chaque processeur dispose d'un message stocké à l'adresse `my_adr`
3. A la fin, chaque processeur a en mémoire le même tableau, contenant à la case k le message diffusé par P_k.

2 Diffusion pipelinée

Enoncé

- Diminuer le temps de la diffusion pour des messages de grande longueur
- Découper le message en r morceaux (de taille égale, en supposant L divisible par r)
- Le processeur émetteur envoie successivement les r morceaux, qui circulent simultanément sur l'anneau
- Au début du programme, les r morceaux du message sont stockés aux adresses adr[1],...,adr[r] du processeur k
- A la fin du programme, le message est stocké dans tous les processeurs aux mêmes adresses.

1. Ecrire le programme
2. Donner le temps d'exécution en fonction de r
3. Calculer la meilleure valeur de r.

3 Factorisation LU

Enoncé

- Algorithme séquentiel, matrice $n \times n$ indicée de 0 à $n-1$:

  for $k = 0$ to $n-2$ do
      /* Tâche T_{kk} : préparation de la colonne k
      prep(k) : for $i = k+1$ to $n-1$ do $a_{ik} = -a_{ik}/a_{kk}$
      for $j = k+1$ to $n-1$ do
        /* Tâche T_{kj} : mise à jour de la colonne j
        update(k,j) : for $i = k+1$ to $n-1$ do $a_{ij} = a_{ij} + a_{ik} * a_{kj}$

- Paralléliser cet algorithme sur un anneau de p processeurs :
  1. Distribuer les colonnes de A aux processeurs : fonction d'allocation alloc
  2. Supposer d'abord alloc donnée et écrire le code
  3. Comment choisir une bonne fonction d'allocation ?
  4. Proposer des améliorations.

Algorithmique parallèle

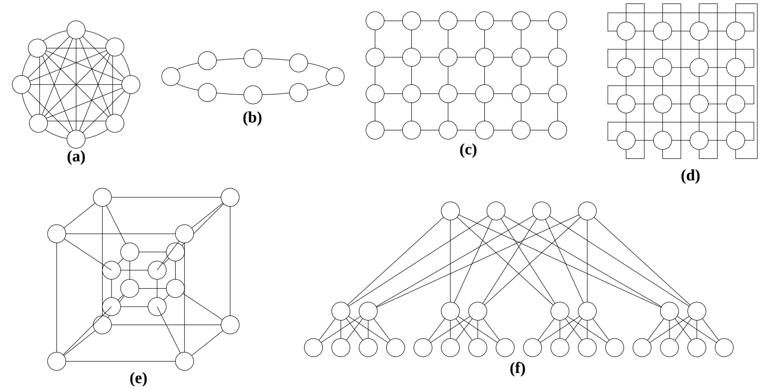
Plan

1. Communications et routage
2. Produit de matrices sur grille 2D
3. Algorithmique hétérogène

1 Réseaux d'interconnexion

Topologies

– Topologie statique : réseau d'interconnexion fixe
– Topologie dynamique : modifiée en cours d'exécution (configurer un *switch*).



– Borne de Moore sur le nombre de processeurs p d'une topologie statique de degré Δ :

$$p = 1 + \Delta + \Delta(\Delta - 1) + \Delta(\Delta - 1)^2 + \ldots + \Delta(\Delta - 1)^{D-1}$$

Topologie	Nbre de proc.	Degré	Diamètre	Nbre de liens
Réseau complet	p	$p-1$	1	$p(p-1)/2$
Anneau	p	2	$\lfloor p/2 \rfloor$	p
Grille 2D	$\sqrt{p}\sqrt{p}$	$2 \rightarrow 4$	$2(\sqrt{p}-1)$	$2p - 2\sqrt{p}$
Tore 2D	$\sqrt{p}\sqrt{p}$	4	$2\lfloor \sqrt{p}/2 \rfloor$	$2p$
Hypercube	$p = 2^d$	$d = \log(p)$	d	$p\log(p)/2$

2 Routage

Modèle "store-and-forward" (SF)

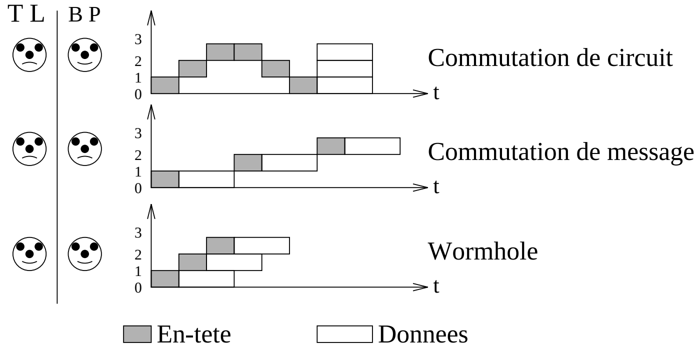
- Chaque processeur intermédiaire reçoit et stocke le message M avant de le ré-émettre en direction du processeur destination
- Modèle à commutation de message
- Première génération de machines
- Coût : $d(x, y)(\beta + L\tau)$
- Pipeline $\rightarrow (\sqrt{(d(x, y) - 1))\beta} + \sqrt{L\tau})^2 = L\tau + o(\sqrt{L})$.

Modèle "cut-through" (CT)

- Co-processeur de communication
- Coût $\beta + (d(x, y) - 1)\delta + L\tau$
- $\delta \ll \beta$: chemin calculé par matériel, partie logicielle dans β
- Deux protocoles :
  1. *Circuit-switching (CC)*
     - Chemin créé avant émission des premiers octets
     - Puis message acheminé directement entre source et destination
     - Immobilisation des liens
     - iPSC d'Intel
  2. *Wormhole (WH)*
     - Adresse du destinataire placée dans l'en-tête du message
     - Routage sur chaque processeur
     - Message découpé en petits paquets appelés *flits*
     - Blocage $\rightarrow$ flits stockés dans les registres internes des routeurs intermédiaires
     - Paragon d'Intel.

Comparaison

- CC et WH plus efficaces que SF : masquer la distance entre les processeurs communiquants
- WH construit sa route tandis que le message avance dans le réseau
- CC construit son chemin (et l'acquitte) avant l'envoi des premiers paquets de données.



TL = Temps de latence
BP = Bande passante.

3 Produit de matrices sur grille 2D

Allocation des données sur les processeurs

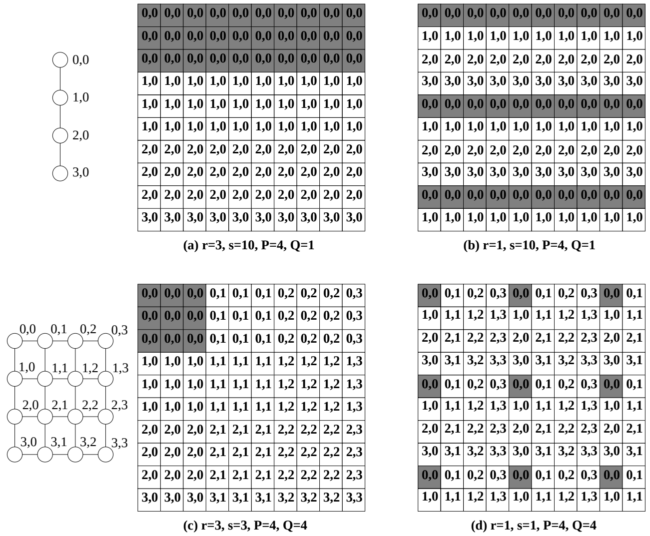
– Algorithmes par blocs→ augmenter le grain de calcul et améliorer l'utilisation des mémoires hiérarchiques

– Distribuer les données de manière cyclique → équilibrer la charge

– Distribution cyclique par blocs pour un vecteur de M composantes sur p processeurs, $0 \leq m < M$ et $0 \leq q < p$:

$$m \mapsto (q, b, i) = \left(\lfloor \frac{m \mod T}{r} \rfloor, \lfloor \frac{m}{T} \rfloor, m \mod r \right)$$

où r est la taille de bloc et $T = rp$.



(a) r=3, s=10, P=4, Q=1 (b) r=1, s=10, P=4, Q=1

(c) r=3, s=3, P=4, Q=4 (d) r=1, s=1, P=4, Q=4

(a) et (b) réseau linéaire, (c) et (d) grille

Implémentation dans une bibliothèque

1. **Version centralisée** Routines appelées en supposant les données sur la machine hôte. Résultats stockés sur la machine hôte.
   + minimise le nombre de routines à écrire
   + choisir la distribution des données la plus adaptée au noyau de calcul
   - coût prohibitif si on enchaîne plusieurs appels

2. **Version distribuée** Données déjà distribuées sur les processeurs, résultats distribués également.
   + *scalable*
   + routines de redistribution des données
   - compromis à trouver entre *coût de redistribution + coût de l'algorithme avec rangement adapté* et *coût de l'algorithme avec rangement non adapté*.

Produit de matrices sur grille 2D

- $C = C + AB$, avec A, B et C de taille $N \times N$
- $p = q^2$, grille cible = tore de taille $q \times q$
- Distribution des matrices par blocs : P_{ij} stocke A_{ij}, B_{ij} et C_{ij}.

Algorithme de Cannon

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} = \begin{matrix} A00 & A01 & A02 & A03 \\ A11 & A12 & A13 & A10 \\ A22 & A23 & A20 & A21 \\ A33 & A30 & A31 & A32 \end{matrix} \; x \; \begin{matrix} B00 & B11 & B22 & B33 \\ B10 & B21 & B32 & B03 \\ B20 & B31 & B02 & B13 \\ B30 & B01 & B12 & B23 \end{matrix}$$

```
/* mouvements de données avant les calculs */
Rotations de A et B

/* calcul du produit de matrice */
pour k = 1 à √P en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
   Rotation horizontale de A
finpour

/* mouvements de données après les calculs */
Rotations de A et B
```


Algorithme de Fox

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} += \begin{matrix} A00 & A00 & A00 & A00 \\ A11 & A11 & A11 & A11 \\ A22 & A22 & A22 & A22 \\ A33 & A33 & A33 & A33 \end{matrix} \; x \; \begin{matrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{matrix}$$

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} += \begin{matrix} A01 & A01 & A01 & A01 \\ A12 & A12 & A12 & A12 \\ A23 & A23 & A23 & A23 \\ A30 & A30 & A30 & A30 \end{matrix} \; x \; \begin{matrix} B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \\ B00 & B01 & B02 & B03 \end{matrix}$$

```
/* pas de mouvements de données avant les calculs */

/* calcul du produit de matrices */
Diffuser la première diagonale de A
pour k = 1 à √P − 1 en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
   Diffuser la diagonale suivante de A
finpour
en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
fin

/* pas de mouvements de données après les calculs */
```

Algorithme de Snyder

$$\begin{array}{cccc} \mathbf{C00} & C01 & C02 & C03 \\ C10 & \mathbf{C11} & C12 & C13 \\ C20 & C21 & \mathbf{C22} & C23 \\ C30 & C31 & C32 & \mathbf{C33} \end{array} = \begin{array}{cccc} A00 & A01 & A02 & A03 \\ A10 & A11 & C12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array} \mathrm{x} \begin{array}{cccc} B00 & B10 & B20 & B30 \\ B01 & B11 & B21 & B31 \\ B02 & B12 & B22 & B32 \\ B03 & B13 & B23 & B33 \end{array}$$

$$\begin{array}{cccc} C00 & \mathbf{C01} & C02 & C03 \\ C10 & C11 & \mathbf{C12} & C13 \\ C20 & C21 & C22 & \mathbf{C23} \\ \mathbf{C30} & C31 & C32 & C33 \end{array} = \begin{array}{cccc} A00 & A01 & A02 & A03 \\ A10 & A11 & C12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array} \mathrm{x} \begin{array}{cccc} B01 & B11 & B21 & B31 \\ B02 & B12 & B22 & B32 \\ B03 & B13 & B23 & B33 \\ B00 & B10 & B20 & B30 \end{array}$$

```
/* mouvements de données avant les calculs */
transposer la matrice B

/* calcul du produit de matrices */
en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
finpour
pour k = 1 à √P − 1 en // faire
   Somme globale sur les lignes pour C_{i,(i+k−1) mod √P}
   LocalProdMat(A, B, C)
   Rotation verticale de B
finpour
Somme globale sur les lignes pour C_{i,(i+√P−1) mod √P}

/* mouvements de données après les calculs */
transposer la matrice B
```


4 Algorithmique hétérogène

Allocation statique de tâches

– $t_1, t_2, \ldots, t_p$ temps de cycle des processeurs
– B tâches identiques et indépendantes
– Principe : $c_i \times t_i = Constante$

$$c_i = \frac{\frac{1}{t_i}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times B$$

– Allocation optimale :

Distribue$(B, t_1, t_2, \ldots, t_p)$
{ Initialisation : calcule c_i tels que $c_i \times t_i \approx Constante$
et $c_1 + c_2 + \ldots + c_p \leq B$ }
do $i = 1, p$
   $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times B \right\rfloor$
{ Incrémenter itérativement les c_i
qui minimisent le temps d'exécution tant que $\sum_{k=1}^{p} c_i < B$}
while $\sum_{i=1}^{p} c_i < B$ do
   trouve $k \in \{1, \ldots, p\}$ tel que $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$
   $c_k = c_k + 1$
Retourne$(c_1, c_2, \ldots, c_k)$

Algorithme incrémental

– Allocation optimale pour tout nombre de tâches entre 1 et B

– Programmation dynamique avec $t_1 = 3$, $t_2 = 5$, et $t_3 = 8$:

Nombre d'atomes	c_1	c_2	c_3	Coût	Proc. sélectionné	Allocation σ
0	0	0	0		1	
1	1	0	0	3	2	$\sigma[1] = 1$
2	1	1	0	2.5	1	$\sigma[2] = 2$
3	2	1	0	2	3	$\sigma[3] = 1$
4	2	1	1	2	1	$\sigma[4] = 3$
5	3	1	1	1.8	2	$\sigma[5] = 1$
6	3	2	1	1.67	1	$\sigma[6] = 2$
7	4	2	1	1.71	1	$\sigma[7] = 1$
8	5	2	1	1.87	2	$\sigma[8] = 1$
9	5	3	1	1.67	3	$\sigma[9] = 2$
10	5	3	2	1.6		$\sigma[10] = 3$

Distribue$(B, t_1, t_2, \ldots, t_p)$

{ Initialisation : aucune tâche à distribuer. $m = 0$ }

do $i = 1, p$

  $c_i = 0$

{ Construit itérativement l'allocation σ}

do $m = 1, B$

  trouve $k \in \{1, \ldots, p\}$ tel que $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$

  $c_k = c_k + 1$

  $\sigma[m] = k$

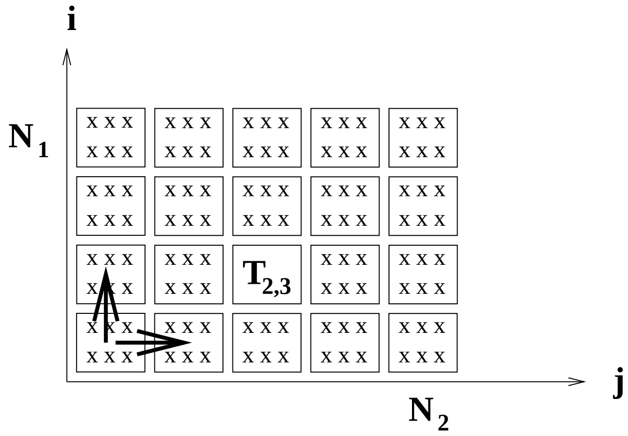
Retourne$(\sigma, c_1, c_2, \ldots, c_k)$

Exemple d'application

– Domaine de calcul rectangulaire

– Vecteurs de dépendance

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$



Allocation pérodique

– p processeurs de temps de cycle $t_1, t_2, \ldots, t_p$

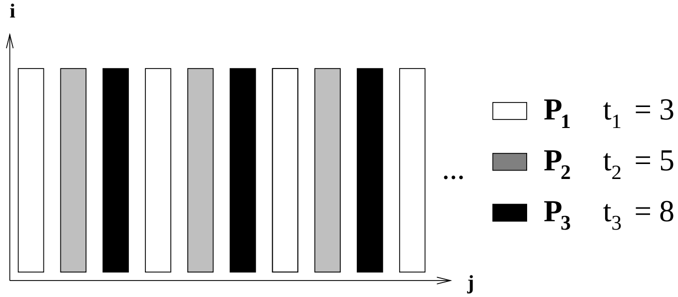
– Allocation de N colonnes par tranches de B colonnes

– Calculer les c_i et trier les $c_i t_i$

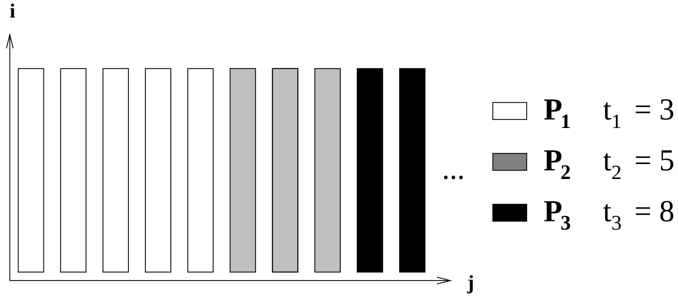
– Allouer, dans cet ordre, c_i colonnes consécutives à P_i

Illustration

– 3 processeurs, $t_1 = 3$, $t_2 = 5$, et $t_3 = 8$, $B = 10$
– $T_{seq} \approx 3N$
– Glouton $\rightarrow$ solution cyclique de période B, $T \approx \frac{8}{3}N$



– Périodique $\rightarrow$ $c_1 = 5$, $c_2 = 3$ et $c_3 = 2$, $T \approx 1.6N$



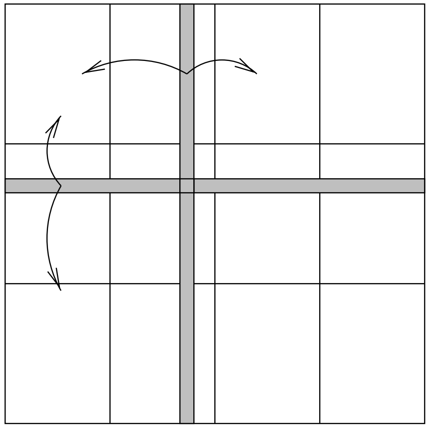
– $T_{opt} \approx \frac{120}{79}N = 1.52N$:
$B = 79 \rightarrow c_1 = 40$, $c_2 = 24$, $c_3 = 15 \rightarrow c_1t_1 = c_2t_2 = c_3t_3$
$\Rightarrow$ équilibrage parfait.

Retour au balayage d'image

– 3 processeurs de temps de cycle $t_1 = 1$, $t_2 = 2$ and $t_3 = 4$
– Période $B = 7$ calculée pour un équilibrage de charge optimal.

P_1	0	1	3	6	10	14	18	22	26	30	$\cdots$
P_1	2	4	7	11	15	19	23	27	31	$\cdots$	
P_1	5	8	12	16	20	24	28	32	$\cdots$		
P_1	9	13	17	21	25	29	33	$\cdots$			
P_2	14	18	22	26	30	34	$\cdots$				
P_2	20	24	28	32	36	$\cdots$					
P_3	26	30	34	38	$\cdots$						
$\cdots$	$\cdots$										

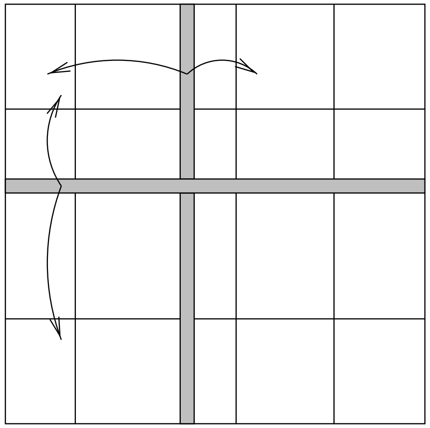
Multiplication de matrices sur grille homogène



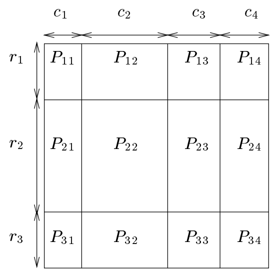
- Algorithme par blocs de ScaLAPACK
- Double diffusion horizontale et verticale
- S'adapte au cas de matrices et grilles
- Aucune redistribution initiale des données.

Multiplication de matrices sur grille hétérogène

- Allouer des rectangles de tailles différentes aux processeurs, en fonction de leur vitesse relative

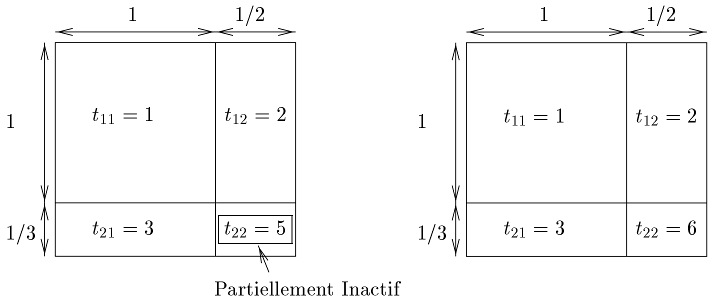


- $p \times q$ processeurs $P_{i,j}$ de temps de cycle $t_{i,j}$



Equilibrage de charge parfait

– Matrice $T = (t_{ij})$ de rang 1



Optimisation

– Objectif Obj_1 : $\displaystyle\min_{(\sum_i r_i = 1; \sum_j c_j = 1)} \max_{i,j} \{r_i \times t_{ij} \times c_j\}$

– Objectif Obj_2 : $\displaystyle\max_{r_i \times t_{ij} \times c_j \leq 1} \{(\sum_i r_i) \times (\sum_j c_j)\}$

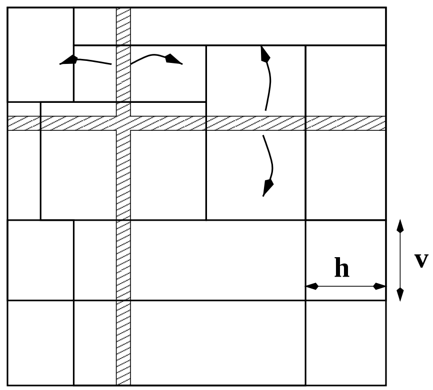
– La position des processeurs dans la grille n'est pas une donnée du problème : toutes les permutations sont possibles, et il faut chercher la meilleure

– Problème est NP-complet

– Difficulté de l'équilibrage de charge 2D !

Partitionnement libre

– Avec 13 processeurs ?



– p processeurs de vitesses $s_1, s_2, \ldots, s_p$ normalisées : $\sum_{i=1}^{p} s_i = 1$

– Partitionner le carré unité en p rectangles de surfaces $s_1, s_2, \ldots, s_p$

– Surface des rectangles $\Longleftrightarrow$ vitesses relatives des processeurs

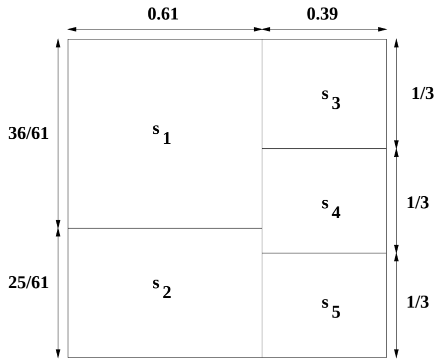
– Forme des rectangles $\Longleftrightarrow$ minimiser les communications

– Géométriquement : partitionner le carré unité en p rectangles d'aires fixées $s_1, s_2, \ldots, s_p$ (tels que $\sum_{i=1}^{p} s_i = 1$) afin de minimiser
  – soit la somme des demi-périmètres des rectangles dans le cas de communications séquentielles
  – soit le plus grand des demi-périmètres des rectangles dans le cas de communications parallèles

– Problèmes NP-complets, of course.

Exemple

– $p = 5$ rectangles $R_1, \dots, R_5$
– Aires $s_1 = 0.36$, $s_2 = 0.25$, $s_3 = s_4 = s_5 = 0.13$



– Demi-périmètre maximal pour R_1, approximativement 1.2002
$\rightarrow$ borne inférieure absolue $2\sqrt{s_1} = 1.2$ atteinte lorsque le plus grand rectangle est un carré (pas possible ici)

– Somme des demi-périmètres = 4.39
$\rightarrow$ borne absolue inférieure $\sum_{i=1}^{p} 2\sqrt{s_i} \approx 4.36$ atteinte lorsque tous les rectangles sont des carrés (pas possible ici).

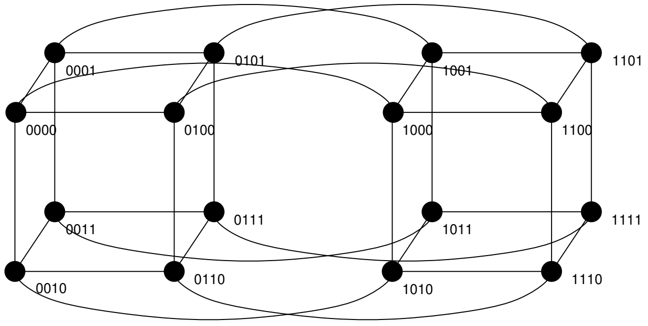
Algorithmique parallèle
TD – Enoncé

Plan

1 Etude de cas : l'hypercube

Définition et numérotation des sommets

– Les hypercubes se construisent par duplication
– Etant donnés deux m-cubes identiques (i.e. avec 2^m sommets), on construit un $m+1$-cube en reliant deux à deux les sommets correspondants



– Autre définition (équivalente) : un m-cube est composé de 2^m sommets numérotés de 0 à 2^m-1, et il existe un lien entre deux sommets $\Longleftrightarrow$ leur écriture binaire ne diffère que par un seul bit.
– Construction récursive $\rightarrow$ étiquetage des sommets :
étant donnés deux m-cubes, on renumérote les sommets du premier en $0a_i$ et ceux du second en $1a_i$, où a_i est l'écriture binaire représentant les deux sommets correspondants du m-cube.
– Un m-cube est de diamètre $D = m$ et de degré $\Delta = m$

Chemins dans l'hypercube

– A et B deux sommets du m-cube
– $H(A, B)$ distance de Hamming entre A et B : nombre de bits qui diffèrent dans l'écriture binaire de A et B.
– *Router* un message de A à B = trouver un chemin dans le m-cube qui mène de A à B, de longueur minimale.
– Questions :
  1. Montrer que H est bien une distance
  2. Construire un chemin de A à B
  3. Combien y-a-t-il de chemins indépendants de longueur minimum entre A et B ? Deux chemins sont dits indépendants s'ils n'ont aucun sommet en commun à l'exception de A et de B.

Routage

1. Comment implanter le routage d'un message de A à B
2. Peut-on trouver un algorithme non statique, i.e. capable de choisir entre plusieurs routes ?

Plongements d'anneaux et de grilles

– G_m code de Gray de dimension $m \geq 2$ défini récursivement comme la concaténation

$$G_m = 0G_{m-1}|1G_{m-1}^{rev}$$

  – xG énumère les éléments de G en rajoutant un x en tête de leur écriture binaire.
  – G^{rev} énumère les éléments de G dans l'ordre renversé (en partant de la fin).
  – On initie la construction avec $G_1 = \{0, 1\}$

– Questions

1. Construire G_2, G_3 et G_4
2. Définir un anneau de 2^m processeurs dans le m-cube
3. Définir une grille torique de taille $2^r \times 2^s$ dans un m-cube, avec $r + s = m$.

Diffusion dans l'hypercube

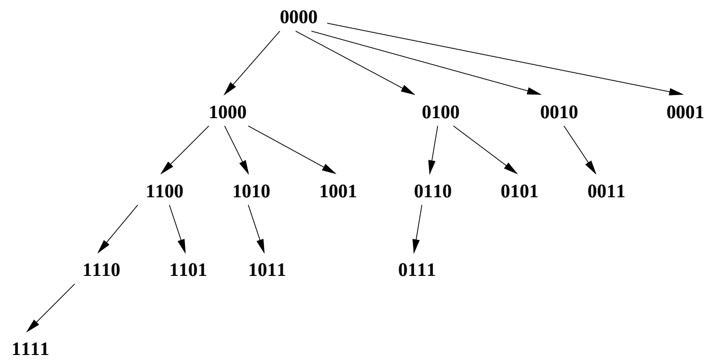
(A) Diffusion simple

– Primitives de communication

send(cube-link, send-adr, L)

receive(cube-link, recv-adr, L)

– Diffusion selon l'arbre couvrant ($m = 4$) :



– Ecrire la procédure de diffusion à partir du processeur k :

broadcast(k, adr, L)

– Quel est le coût en modèle *store and forward*?

(B) Diffusion m-ports

– Construction de m arbres couvrants à arêtes indépendantes :

1. Prendre l'arbre couvrant précédent, enraciné en 0, noté A_0
2. Effectuer une rotation (à gauche) des écritures binaires des sommets de A_0 pour construire les autres arbres : A_i est l'image de A_0 par l'opérateur de rotation $R^{(i)}$.
   Si $s = s_{m-1}s_{m-2}\dots s_1s_0$, on définit $R(s) = s_{m-2}s_{m-3}\dots s_0s_{m-1}$.
   On pose $R^{(1)} = R$ et $R^{(i)} = R \circ R^{(i-1)}$ pour $2 \leq i \leq m-1$
3. On inverse le i-ème bit (modulo m) des sommets de A_i
4. Après cette opération XOR, le noeud 0 est une feuille dans chacun des m arbres A_i. On fusionne ceux-ci en un seul arbre de racine 0, et de hauteur m

– Faire la construction pour $m = 3$

– Quel est le coût en modèle *store and forward*?

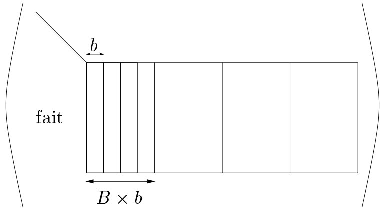
2 Algorithmique hétérogène

LU hétérogène

– A chaque étape, le processeur qui possède le bloc pivot le factorise et le diffuse

– Les autres processeurs mettent à jour les colonnes restantes

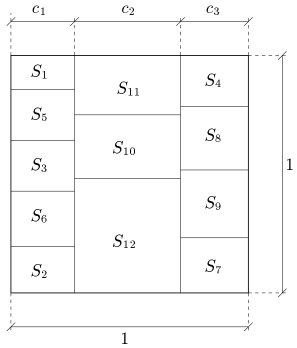
– A l'étape suivante le bloc des b colonnes suivantes devient le pivot



Question : Trouver une allocation statique de période B qui équilibre au mieux les tâches lors de chaque étape de mise à jour.

Partitionnement en colonnes

– n réels positifs $s_1, s_2, \ldots, s_n$ de somme 1
– Partitionner le carré unité en n rectangles de surfaces $s_1, s_2, \ldots, s_n$
– On cherche un partitionnement en colonnes



– Fonction objective à minimiser : somme des traits dessinés, i.e. (à une constante près) la somme des demi-périmètres des n rectangles.

Questions

1. Résoudre si $n = 4$ et $s_1 = s_2 = s_3 = s_4 = 0.25$.
2. Résoudre si $n = 6$, $s_1 = 0.5$ et $s_2 = s_3 = s_4 = s_5 = s_6 = 0.1$
3. Dans le cas général, on ne connaît pas le nombre de colonnes qu'on va utiliser. Mais :
   (a) Montrer qu'on peut se ramener à placer les s_i dans l'ordre croissant
   (b) Résoudre le problème par programmation dynamique