INF 560
Calcul Parallèle et Distribué
Cours 7

Eric Goubault et Sylvie Putot

Ecole Polytechnique

2 février 2015

Plan

- Communications et routage
  - Généralités
  - Cas de l'hypercube (diffusion simple)
  - Cas du tore 2D (multiplication de matrices)
- Algorithmique sur ressources hétérogènes
  - Allocation et équilibrage de charges statique/dynamique
  - Cas de l'algorithme LU: sur anneau (1D) puis tore (2D)

Généralités

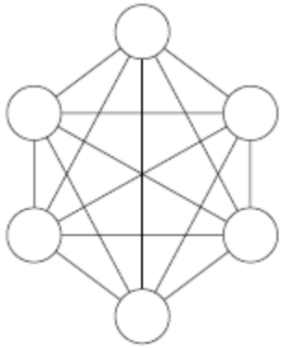
- Topologie statique: réseau d'interconnexion fixe:
  - anneau, tore 2D,
  - hypercube, graphe complet etc.
- Topologie dynamique: modifiée en cours d'exécution (par configuration de *switch* pour établir une connexion)

Caractéristiques d'un réseau d'interconnexion statique

Représentable par un graphe dont les sommets sont les processeurs et les arêtes des liens de communication (les liens sont le plus souvent bidirectionnels)

- Le degré (nombre d'arêtes partant d'un noeud): nombre de voisins immédiats
- Le diamètre: distance (= longueur du plus court chemin) maximale entre deux noeuds
- Le nombre de liens
- Largeur de bisection: nombre minimal de connexions qu'il faut retirer pour avoir deux parties égales non connectées (mesure la capacité du réseau à transmettre des messages simultanément)

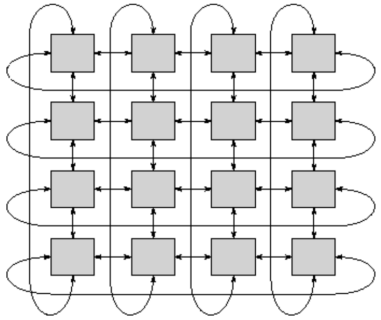
Réseau complet



- Diamètre = 1: idéal pour le temps de communications
- Mais degré = $p-1$ et nombre de liens $p(p-1)/2$: passage à l'échelle difficile! (prix du cablage, rajouter des nouveaux processeurs?)
- Largeur de bisection = $(p/2)^2$: $p/2$ noeuds qui ont une connection à $p/2$ autres noeuds pour couper le réseau en 2

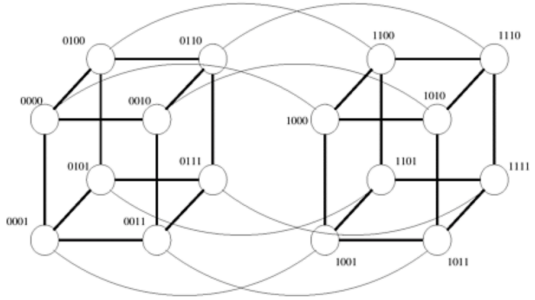
Grille et tore (ici 2D)

- Anneau: pas cher et simple, mais communications lentes (diamètre $p/2$), et tolérance aux pannes des liens faible (largeur bisection 2)
- Grille 2D: degré 2 à 4, diamètre en $2\sqrt{p}$, largeur bisection $\sqrt{p}$, facile à étendre, mais manque de symétrie (bords)
- Tore 2D: bon compromis dérivé de la grille, degré 4, diamètre en $\sqrt{p}$, largeur bisection $2\sqrt{p}$
- Connectivité encore augmentée en 3D (ex Crays T3D, T3E)



Hypercube

- Hypercube: un autre compromis intéressant
- Définition récursive du cube de dimension m ($p = 2^m$ noeuds) à partir de 2 cubes de dimension m en reliant les sommets correspondants de chaque cube
- Degré m, diamètre faible (m), largeur bisection 2^{m-1}
- Mais nombre de liens croit rapidement avec le nombre de processeurs $m2^{m-1}$



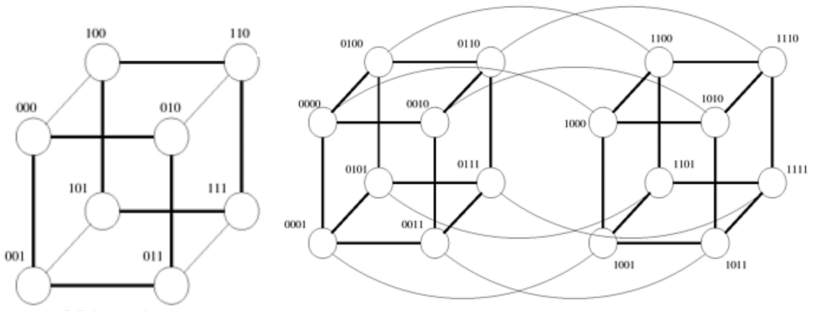
Communications dans un hypercube

- Chemins et routage dans un hypercube
- Plongement d'anneaux et de grilles dans un hypercube
- Broadcast dans un hypercube

Numérotation des sommets

Un m-cube est la donnée de:

- Sommets numérotés de 0 à $2^m - 1$ (représentation binaire de longueur m)
- Il existe une arête d'un sommet à un autre si les deux diffèrent seulement d'un bit dans leur écriture binaire



Chemins et routage

Chemins entre deux sommets/processeurs A et B:

- Routage: trouver un chemin de longueur minimale
- Distance de Hamming $H(A, B)$ entre deux sommets = le nombre de bits qui diffèrent dans l'écriture
- Distance de Hamming entre deux sommets adjacents est 1
- Il existe un chemin de longueur $H(A, B)$ entre A et B (récurrence facile sur $H(A, B)$)
- Il existe $H(A, B)!$ chemins entre A et B, dont seuls $H(A, B)$ sont indépendants (n'ont aucun sommet en commun exceptés A et B)
- Des chemins indépendants permettent l'acheminement simultané de plusieurs messages de A à B

Chemins et routage

Un routage: on construit un chemin $A, A_1, A_2, \ldots, B$ en corrigeant à chaque étape le bit de poids faible des bits qui diffèrent.

Implémentation:

- Le message circule avec un en-tête, initialement égal à (A xor B) calculé bit à bit
- Le routeur du processeur du noeud courant examine l'en-tête
  - s'il est nul, le message est pour lui
  - sinon il met à 0 le bit non nul de poids le plus faible de l'en-tête et envoie le message sur le lien correspondant

Exemple:

- $A = 1011$, $B = 1101$, A xor $B = 0110$ (ou exclusif bit à bit)
- A envoie donc son message vers 1001 avec entête 0100
- Puis 1001 renvoie vers $1101 = B$ avec entête 0000.

Aussi algorithmes dynamiques qui prennent les liens disponibles

Plongements d'anneaux et de grilles

Pourquoi des plongement d'anneaux et grilles dans des hypercubes:

- Algorithmes conçus sur un anneau ou une grille
- Utiliser la connectivité de l'hypercube pour les communications globales comme les diffusions

Principe:

- Préserver la proximité des voisins
- On cherche des plongements qui minimisent la distance entre les processeurs image de processeurs voisins dans le réseau de départ

Plongements d'anneaux et de grilles

On se limite aux anneaux et grilles de dimensions 2^m

Codes de Gray

Un code de Gray = une suite ordonnée de codes binaires tels que l'on passe au code suivant en ne modifiant qu'un seul bit

- Le code de Gray G_m de dimension $m \geq 2$ est défini récursivement:

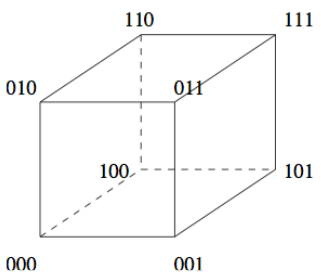
$$G_m = \{0G_{m-1}, 1G_{m-1}^{rev}\}$$

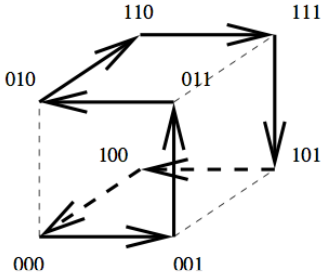
  - xG énumère les éléments de G en rajoutant x en tête de leur écriture binaire
  - G^{rev} énumère les éléments de G dans l'ordre renversé

$G_1 = (0, 1)$, $G_2 = (00, 01, 11, 10)$,
$G_3 = (000, 001, 011, 010, 110, 111, 101, 100)$,
$G_4 = (0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1011, 1010, 1001, 1000)$.

Codes de Gray: intérêt

- Permet de définir un anneau de 2^m processeurs dans le m-cube grâce à G_m en projetant le processeur i de l'anneau sur le i^{eme} élément, noté $g_i^{(m)}$, de la suite G_m.
  Exemple $G_3 = (000, 001, 011, 010, 110, 111, 101, 100)$





- Permet de définir un réseau torique de taille $2^r \times 2^s$ dans un m-cube avec $r + s = m$ (utiliser $G_r \times G_s$: le processeur (i, j) de la grille a pour image $(g_i^{(r)}, g_j^{(s)})$ dans le m-cube)

Diffusion simple dans l'hypercube

On suppose que le processeur 0 veut envoyer un message à tous les autres

- Algorithme naïf 1: le processeur 0 envoie à tous ses voisins, puis tous ses voisins à tous leurs voisins etc.: très redondant (et gérer tous les send/receive...)

On voudrait également que chaque processeur reçoive le message une seule fois:

- Algorithme naïf (un peu moins) 2: on utilise le code de Gray et la diffusion sur l'anneau

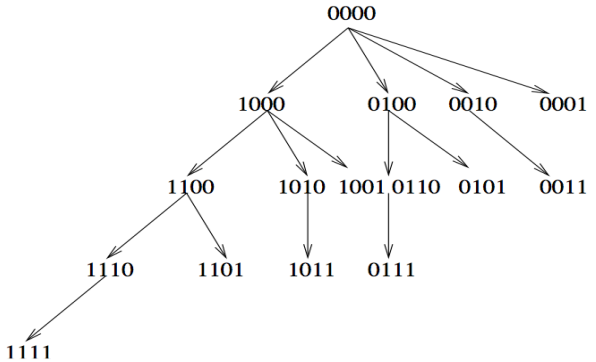
Arbres couvrants de l'hypercube

- Primitives: `send(cube-link,send-adr,L)`, `receive(cube-link,recv-adr,L)`
- On construit à la fois l'arbre couvrant et l'algo de diffusion, qui opère en m phases, numérotées de $m-1$ à 0
- Les processeurs vont recevoir le message sur le lien correspondant à leur premier 1 (à partir des poids faibles)
- Et vont propager sur les liens qui précèdent ce premier 1
- Le processeur 0 est supposé avoir un 1 fictif en position m

Arbres couvrants de l'hypercube

Exemple ($m = 4$):

- phase 3: 0000 envoie le message sur le lien 3 à 1000
- phase 2: 0000 et 1000 envoient le message sur le lien 2, à 0100 et 1100 respectivement
- ainsi de suite jusqu'à la phase 0



Si processeurs m-ports: message découpé et algo pipeliné sur arbres couvrants à arêtes indépendantes

Algorithmique sur grille torique 2D

Trois algorithmes pour le produit de matrices carrées denses:

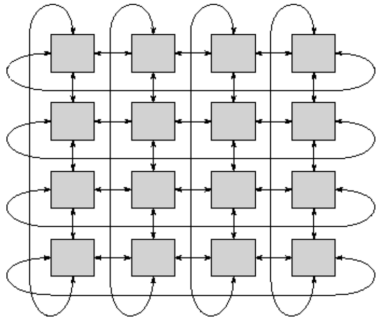
- Algorithme de Cannon
- Algorithme de Fox
- Algorithme de Snyder

Objectifs:

- Réduire le ratio communication/calcul
- Minimiser l'occupation mémoire (un bloc de chaque matrice par processeur)

Produit de matrices sur grille 2D

- $C = C + AB$, avec A, B et C de taille $N \times N$
- $p = q^2$: on a une grille de processeurs en tore de taille $q \times q$



- Distribution des matrices par blocs: P_{ij} stocke A_{ij}, B_{ij} et C_{ij}

Principe de l'algorithme de Cannon

- Distribution des données au départ:

$$\begin{matrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{matrix} = \begin{matrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{matrix} \times \begin{matrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{matrix}$$

- 'Preskewing":
  - Pour $i = 0$ à $q - 1$, décalage circulaire de la ligne i de la matrice A de i positions à gauche
  - Pour $j = 0$ à $q - 1$, décalage circulaire de la colonne j de la matrice B de j positions vers le haut

$$\begin{matrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{matrix} = \begin{matrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{11} & A_{12} & A_{13} & A_{10} \\ A_{22} & A_{23} & A_{20} & A_{21} \\ A_{33} & A_{30} & A_{31} & A_{32} \end{matrix} \otimes \begin{matrix} B_{00} & B_{11} & B_{22} & B_{33} \\ B_{10} & B_{21} & B_{32} & B_{03} \\ B_{20} & B_{31} & B_{02} & B_{13} \\ B_{30} & B_{01} & B_{12} & B_{23} \end{matrix}$$

La diag. de A est sur sa 1ère colonne, celle de B sur sa 1ère ligne

Principe de l'algorithme de Cannon (2)

- Initialisation $C_{ij} = 0, \forall i, j \in 0, \ldots, q-1$, $A^{(0)}$ et $B^{(0)}$ les matrices "preskewees"
  - $A^{(0)}_{ij} = A_{i,(i+j) \bmod q}$, $B^{(0)}_{ij} = B_{(i+j) \bmod q,j}$
- Puis q étapes de calcul, de $k = 0$ à $q-1$: multiplication locales aux processeur, et décalage par communications entre voisins
  - $C_{ij} = C_{ij} + A^{(k)}_{ij} \times B^{(k)}_{ij} = C_{ij} + A_{i,(i+j+k) \bmod q} \times B_{(i+j+k) \bmod q,j}$ (produit de matrice local à chaque bloc) en parallèle
  - $A^{(k+1)} = A^{(k)}$ dont chaque ligne a été décalée circulairement d'une position vers la gauche
  - $B^{(k+1)} = B^{(k)}$ dont chaque colonne a été décalée circulairement d'une position vers le haut

Etape 1:

$$\begin{matrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{matrix} += \begin{matrix} A_{01} & A_{02} & A_{03} & A_{00} \\ A_{12} & A_{13} & A_{10} & A_{11} \\ A_{23} & A_{20} & A_{21} & A_{22} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{matrix} \otimes \begin{matrix} B_{10} & B_{21} & B_{32} & B_{03} \\ B_{20} & B_{31} & B_{02} & B_{13} \\ B_{30} & B_{01} & B_{12} & B_{23} \\ B_{00} & B_{11} & B_{22} & B_{33} \end{matrix}$$

Algorithme de Cannon

```
/* diag(A) sur col 0, diag(B) sur ligne 0 */
Rotations(A,B); /* preskewing */

/* calcul du produit de matrice */
forall (k=0; k<=q-1) {
  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards);
  HorizontalRotation(A,leftwards); }

/* mouvements des donnees apres les calculs */
Rotations(A,B); /* postskewing */
```


Algorithme de Fox

Algorithmes de Fox et Snyder:

- D'autres façons de réorganiser les matrices A et B pour toujours pouvoir faire des produits entrée par entrée sur chaque processeur
- 3 algos difficiles à comparer, les coûts de communication dépendent des paramètres du réseau

Principe de l'algorithme de Fox:

- Pas de mouvements de données initiales
- Diffusions horizontales des éléments de A
- Rotations verticales de B (de bas en haut)
- Comme pour Cannon: à chaque étape, produit matriciel par blocs sur chaque entrée

Principe de l'algorithme de Fox

- Initialisation $C_{ij} = 0, \forall i, j \in 0, \ldots, q-1$, pas de mouvement de données initial
- Puis q étapes de calcul, de $k = 0$ à $q-1$:
  - Diffusion horizontale sur chaque ligne de A: dans la ligne i, $A^{(k)}_{ij} = A_{i,(i+k) \bmod q}$, $\forall j \in \{0, \ldots, q-1\}$
  - Rotation verticale de B (chaque colonne est décalée circulairement d'une position vers le haut): $B^{(0)}_{ij} = B$, $B^{(k+1)}_{ij} = B^{(k)}_{(i+1) \bmod q,j} = B_{(i+k) \bmod q,j}$, $\forall i, j \in \{0, \ldots, q-1\}$
  - $C_{ij} = C_{ij} + A^{(k)}_{ij} \times B^{(k)}_{ij} = C_{ij} + A_{i,(i+k) \bmod q} \times B_{(i+k) \bmod q,j}$ (produit de matrice local à chaque bloc) en parallèle

Algorithme de Fox: étapes 0 et 1

- Etape 0:

$$\begin{matrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{matrix} += \begin{matrix} A_{00} & A_{00} & A_{00} & A_{00} \\ A_{11} & A_{11} & A_{11} & A_{11} \\ A_{22} & A_{22} & A_{22} & A_{22} \\ A_{33} & A_{33} & A_{33} & A_{33} \end{matrix} \otimes \begin{matrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{matrix}$$

Pour tous $i, j \in \{0, \ldots, q-1\}$ en parallèle, calcul de $C_{ij} = A_{ii} \times B_{ij}$

- Etape 1:

$$\begin{matrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{matrix} += \begin{matrix} A_{01} & A_{01} & A_{01} & A_{01} \\ A_{12} & A_{12} & A_{12} & A_{12} \\ A_{23} & A_{23} & A_{23} & A_{23} \\ A_{30} & A_{30} & A_{30} & A_{30} \end{matrix} \otimes \begin{matrix} B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \\ B_{00} & B_{01} & B_{02} & B_{03} \end{matrix}$$

Pour tous $i, j \in \{0, \ldots, q-1\}$ en parallèle, calcul de $C_{ij} = C_{i,j} + A_{i(i+1)} \times B_{(i+1)j}$

Algorithme de Fox

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

/* calcul du produit de matrices */
broadcast(A(x,x));
forall (k=0; k<q-1) {
  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards);
  broadcast(A(k+x,k+x)); }

  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards);

/* pas de mouvements de donnees apres les calculs */
```


Principe de l'algorithme de Snyder

- Transposition préalable de B
- Sommes globales sur les lignes de processeurs (des produits calculés à chaque étape) accumulées dans le coefficient correspondant dans C
- Une diagonale du tore calculée dans C à chaque étape - représentée en gras dans les transparents ci-après
- Rotation verticale de B vers le haut à chaque étape

Algorithme de Snyder: étapes 0 et 1

- Etape 0:

$$\begin{matrix} \mathbf{C_{00}} & C_{01} & C_{02} & C_{03} \\ C_{10} & \mathbf{C_{11}} & C_{12} & C_{13} \\ C_{20} & C_{21} & \mathbf{C_{22}} & C_{23} \\ C_{30} & C_{31} & C_{32} & \mathbf{C_{33}} \end{matrix} += \begin{matrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{matrix} \times \begin{matrix} B_{00} & B_{10} & B_{20} & B_{30} \\ B_{01} & B_{11} & B_{21} & B_{31} \\ B_{02} & B_{12} & B_{22} & B_{32} \\ B_{03} & B_{13} & B_{23} & B_{33} \end{matrix}$$

Pour tous $i \in \{0, \ldots, q-1\}$ en parallèle, calcul de $C_{ii} = \sum_{j=0}^{q-1} A_{ij} \times B_{ji}$ (somme par ligne des produits par blocs)

- Etape 1:

$$\begin{matrix} C_{00} & \mathbf{C_{01}} & C_{02} & C_{03} \\ C_{10} & C_{11} & \mathbf{C_{12}} & C_{13} \\ C_{20} & C_{21} & C_{22} & \mathbf{C_{23}} \\ \mathbf{C_{30}} & C_{31} & C_{32} & C_{33} \end{matrix} += \begin{matrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{matrix} \times \begin{matrix} B_{01} & B_{11} & B_{21} & B_{31} \\ B_{02} & B_{12} & B_{22} & B_{32} \\ B_{03} & B_{13} & B_{23} & B_{33} \\ B_{00} & B_{10} & B_{20} & B_{30} \end{matrix}$$

Pour tous $i \in \{0, \ldots, q-1\}$ en parallèle, calcul de $C_{i(i+1)} = \sum_{j=0}^{q-1} A_{ij} \times B_{j(i+1)}$ (somme par ligne des produits blocs)

Principe de l'algorithme de Snyder

```
/* mouvements des donnees avant les calculs */
Transpose(B);
/* calcul du produit de matrices */
forall () {
  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards); }
forall (k=0;k<q-1) {
  GlobalSum(C(i,(i+k) mod q));
  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards); }
GlobalSum(C(i,(i+q-1) mod q));
/* mouvements des donnees apres les calculs */
Transpose(B);
```


Algorithmique hétérogène

Allocation statique de tâches:

- B tâches atomiques identiques et indépendantes à exécuter
- $t_1, t_2, \ldots, t_p$ temps de cycle des processeurs pour effectuer une tâche atomique
- Nombre de tâches c_i à allouer au processeur i?
- Principe: $c_i \times t_i = constante$, d'où idéalement:

$$B = \underbrace{(c_i \times t_i)}_{\text{temps de l'appli}} \times \underbrace{(\sum_{k=1}^{p} \frac{1}{t_k})}_{\text{nb de tâches exécutables par cycle}}$$

En général, l'équilibrage parfait ($c_i \times t_i = constante$) n'est pas possible, on minimise le temps total en partant de:

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

Allocation statique optimale

Distribute(B,t[1],t[2],...,t[n])

```
// initialisation tq c[i]*t[i] ~ cste et c[1]+C[2]+...+C[n] <= B
for (i=1;i<=p;i++)
  c[i] = ... // formule du transparent précédent
// incrementer iterativement les c[i] minimisant le temps
while (sum(c[])<B) {
  k = argmin(k in {1,...,p})(t[i]*(c[i]+1));
  c[k] = c[k]+1;
return(c[]);
```

L'algorithme donne l'allocation optimale. Au plus p étapes donc $O(p^2)$.

Algorithme incrémental

- Si maintenant B pas forcément fixé
- On veut allocation optimale pour tout nombre d'atomes entre 1 et B, étant donné $t_1, t_2, \ldots, t_n$
- Coût moyen d'exécution d'un atome pour l'allocation $C = (c_1, c_2, \ldots, c_n)$ est

$$t(C) = \frac{max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$$

- Programmation dynamique pour minimiser à chaque atome supplémentaire le coût: calculer

$\text{argmin}(t(c_1{+}1, c_2, \ldots, c_n), t(c_1, c_2{+}1, \ldots, c_n), \ldots, t(c_1, c_2, \ldots, c_n{+}1))$

Algorithme incrémental

Distribute_incr(B,t[1],t[2],...t[p])

```
/* Initialisation : aucune tache a distribuer m=0 */
for (i=1;i<=p;i++) c[i]=0;
/* construit iteративement l'allocation sigma */
for (m=1;m<=B;m++)
  k = argmin(k in {1,...p})(t[i]*(c[i]+1));
  c[k]=c[k]+1;
  sigma[m]=k;
return(sigma[],c[]);
```

Retourne la distribution optimale pour tout sous-ensemble d'atomes $[1, m], m \leq B$, complexité $O(pB)$

Exemple avec $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$

$\#atomes$	c_1	c_2	c_3	$cout$	$proc.sel.$	$alloc.\sigma$
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	$\sigma[5] = 1$
...						
9	5	3	1	1.67	3	$\sigma[9] = 2$
10	5	3	2	1.6		$\sigma[10] = 3$

Application à la factorisation LU

LU pour une matrice de taille $nb \times nb$, et des blocs de b colonnes alloués par processeurs:

- A chaque étape, le processeur qui possède le bloc pivot le normalise 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, et ainsi de suite
- L'étape coûteuse est la mise à jour: trouver une allocation statique pour équilibrer les charges?
- Difficulté: la taille des données à traiter passe d'une étape à l'autre de $(n-1) \times b$ à $(n-2) \times b$ etc.
  - 1ère idée: équilibrer les charges à chaque étape par l'algo précédent
  - il faut redistribuer les données à chaque étape
  - coût de communication important, et pas modulaire (dans une bibliothèque d'algèbre linéaire, on veut même allocation au début et à la fin)

Allocation équilibrant les charges 1D

Trouver une allocation statique des données fournissant un équilibrage des charges de mise à jour qui soit commun à toutes les étapes:

- Distribution de B tâches sur p processeurs de temps de cycle t_1, t_2 etc. t_p telle que
- Pour tout $i \in \{2, \ldots, B\}$, le nombre de blocs de $\{i, \ldots, B\}$ que possède chaque processeur P_j soit approximativement inversement proportionnel à t_j
- On va utiliser l'algorithme incrémental

Allocation 1D

- Matrice $(nb) \times (nb)$
- Allouer périodiquement, sous forme d'un motif de largeur B, les blocs de b colonnes aux processeurs
- B est un paramètre, par exemple si $B = n$ le motif a la largeur de la matrice et n'est pas répété
- Meilleur pour le recouvrement calcul communication si $B << n$, mais supposons $B = n$ pour simplifier
- Utiliser l'algorithme précédent en sens inverse: on commence par allouer le bloc de colonnes $B = n$ (sur $\sigma(1)$), puis $n - 1$ (sur $\sigma(2)$), etc bloc k sur $\sigma(B - k + 1)$.
- Cette distribution est quasi-optimale pour tout sous-ensemble $[i, n]$ de blocs de colonnes

Exemple

$n = B = 10$, $t_1 = 3$, $t_2 = 5$, $t_3 = 8$ le motif sera:

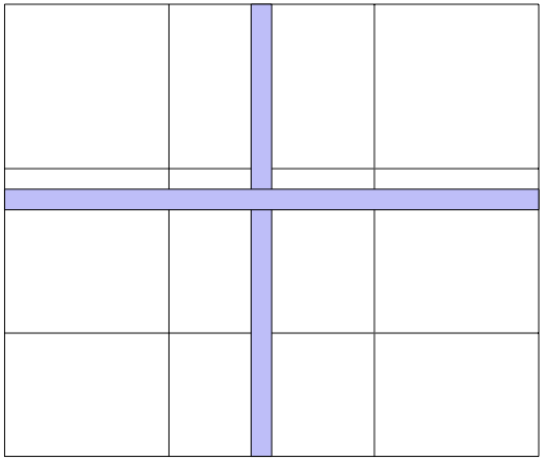
P_3	P_2	P_1	P_1	P_2	P_1	P_3	P_1	P_2	P_1

Equilibrage de charge 2D

Exemple: Multiplication de matrices sur grille homogène:

- Algorithme par blocs de ScaLAPACK (version plus simple que celles vues précédemment)
- Répartition par blocs de A_{ij}, B_{ij} et C_{ij} sur P_{ij}
- Double diffusion horizontale et verticale à chaque étape $k = 0, \ldots, p - 1$
  - A_{ik} est diffusé sur la ligne i et B_{ki} sur la colonne i pour tout i
  - puis produit sur chaque processeur: $C_{ij} = C_{ij} + A_{ik} B_{kj}$
- S'adapte facilement au cas de matrices et grilles rectangulaires
- Aucune redistribution initiale des données

Multiplication de matrices - cas homogène



Multiplication - cas inhomogène "régulier"

- Allouer des rectangles (blocs) de tailles différentes aux processeurs, en fonction de leur vitesse relative
- $p \times q$ processeurs $P_{i,j}$ de temps de cycle $t_{i,j}$
- On suppose le rectange de taille $r_i \times c_j$ alloué au processeur P_{ij}
- Temps d'exécution du processeur P_{ij} est donc $t_{ij} r_i c_j$
- Equilibrage de charge parfait:
  $t_{ij} r_i c_j = Cste = K$, $\forall (i,j) \in \{1,\ldots,p\} \times \{1,\ldots,q\}$

Equilibrage de charge parfait

- Possible ssi la matrice des temps de cycle $T = (t_{i,j})$ est de rang 1:
  - posons $r_1 = 1$ et $c_1 = 1/t_{11}$, puis $r_i = 1/(t_{i1} c_1)$ et $c_j = 1/t_{1j}$ pour $i, j \geq 2$.
  - alors on a $t_{ij} r_i c_j = 1$ pour $i, j \geq 2$ car le déterminant
    $$\begin{vmatrix} t_{11} & t_{1j} \\ t_{i1} & t_{ij} \end{vmatrix}$$
    est nul.
- Exemple, rang 2, $P_{2,2}$ est partiellement inactif:

	$c_1 = 1$	$c_2 = \frac{1}{2}$
$r_1 = 1$	$t_{11} = 1$	$t_{12} = 2$
$r_2 = \frac{1}{3}$	$t_{21} = 3$	$t_{22} = 5$

- Exemple, rang 1, équilibrage parfait:

	$c_1 = 1$	$c_2 = \frac{1}{2}$
$r_1 = 1$	$t_{11} = 1$	$t_{12} = 2$
$r_2 = \frac{1}{3}$	$t_{21} = 3$	$t_{22} = 6$

Résolution générale du problème

Deux façons d'écrire le problème d'optimisation donnant les r_i et c_j:

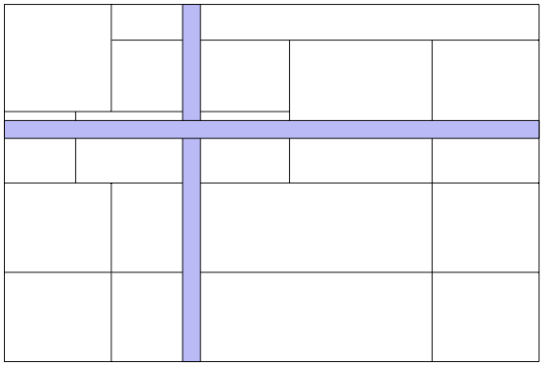
- Objectif $Obj1$ (minimiser le temps d'exécution, normalisé à un motif de taille 1×1, on multiplie ensuite le résultat par n et on prend des entiers proches):
  $$min_{\sum_i r_i = 1; \sum_j c_j = 1} max_{i,j} \{r_i \times t_{i,j} \times c_j\}$$
- Objectif $Obj2$ (dual: maximiser la taille du motif pouvant être traité en une unité de temps):
  $$max_{r_i \times t_{i,j} \times c_j \leq 1} \left\{ \left(\sum_i r_i \right) \times \left(\sum_j c_j \right) \right\}$$

Régularité?

- En fait, la position des processeurs dans la grille n'est pas une donnée du problème
- Toutes les permutations des pq processeurs en une grille $p \times q$ sont possibles, et il faut chercher la meilleure
- Problème NP-complet
- Conclusion: l'équilibrage 2D est très difficile!

Partitionnement libre

Ne plus se restreindre à une grille ... ou comment faire avec p (quelconque, ici 13) processeurs?



Partitionnement libre

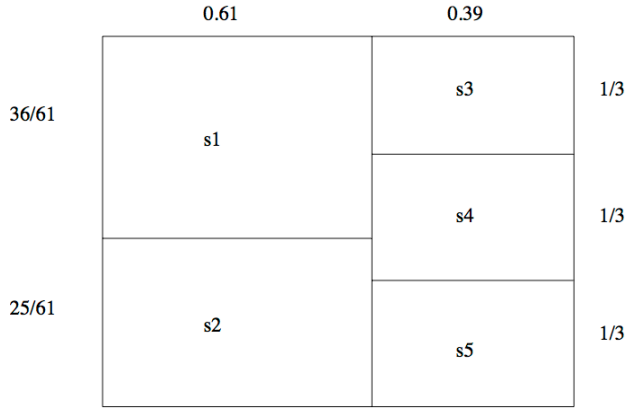
- p processeurs de vitesses $s_1, s_2, \ldots, s_p$ de somme 1 (normalisées)
- Partitionner le carré unité en p rectangles de surfaces $s_1, s_2, \ldots, s_p$
- Surface des rectangles $\Leftrightarrow$ vitesses relatives des processeurs
- Forme des rectangles $\Leftrightarrow$ minimiser les communications

Géométriquement

- Partitionner le carré unité en p rectangles d'aires fixées s_1, s_2, ..., s_p afin de minimiser
  - soit la somme des demi-périmètres des rectangles dans le cas des communications séquentielles
  - soit le plus grand des demi-périmètres des rectangles dans le cas de communications parallèles
- Problèmes NP-complets

Exemple

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



Exemple

- Demi-périmètre maximal pour R_1, approximativement 1.2002
  - 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 $= 4.39$
  - borne absolue inférieure $\sum_{i=1}^{p} 2\sqrt{s_i} = 4.36$ atteinte lorsque tous les rectangles sont des carrés