Parallélisme

ERIC GOUBAULT
COMMISSARIAT À L'ENERGIE ATOMIQUE
SACLAY

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*)

CARACTÉRISTIQUES

Topologie	# proc.	d^o	diam.	# liens
Complet	p	$p-1$	1	$\frac{p(p-1)}{2}$
Anneau	p	2	$\lfloor\frac{p}{2}\rfloor$	p
Grille 2D	$\sqrt{(p)}\sqrt{(p)}$	$2, 3, 4$	$2(\sqrt{(p)}-1)$	$2p-2\sqrt{(p)}$
Tore 2D	$\sqrt{(p)}\sqrt{(p)}$	4	$2\lfloor\frac{\sqrt{(p)}}{2}\rfloor$	$2p$
Hypercube	$p=2^d$	$d=log(p)$	d	$\frac{plog(p)}{2}$

Intérêt

- Réseau complet: idéal pour le temps de communications (toujours unitaire)
- Mais: passage à l'échelle difficile! (prix du cablage, rajouter des nouveaux processeurs?)
- Anneau: pas cher, mais communications lentes, et tolérance aux pannes des liens faible
- Tore 2D, Hypercube: bons compromis entre les deux

Routage

Modèle "Store and Forward" (SF):

- Chaque processeur intermédiaire reçoit et stocke le message M avant de le re-émettre en direction du processeur destinataire
- Modèle à commutation de message
- Première génération de machines
- Coût: $d(x,y)(\beta + L\tau)$
- Sauf programmation particulière (voir cours 5): $L\tau + O(\sqrt(L))$

Routage

Modèle "cut-through" (CT):

- Co-processeur de communication
- Coût $\beta + (d(x,y) - 1)\delta + L\tau$
- $\delta << \beta$: chemin calculé par le matériel, partie logicielle dans β
- Deux protocoles: "Circuit-switching" (CC) et "Wormhole" (WH)

Circuit-switching

- Chemin créé avant l'émission des premiers octets
- Puis message acheminé directement entre source et destinataire
- Immobilisation des liens
- iPSC d'INTEL

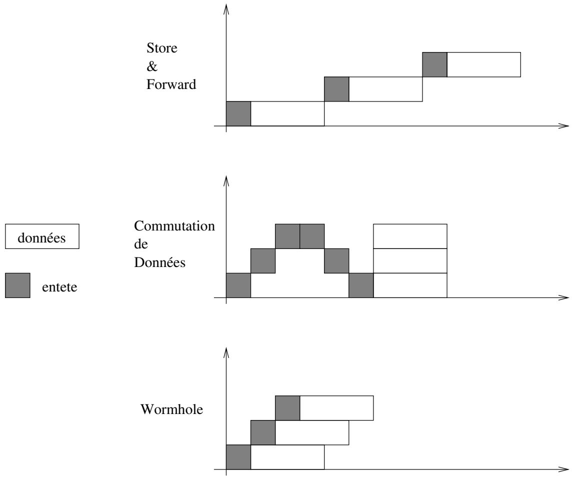
Wormhole

- Adresse du destinataire placée dans l'entête du message
- Routage sur chaque processeur
- Message découpé en petits paquets appelés *flits*
- En cas de blocage: *flits* stockés dans les registres internes des routeurs intermédiaires
- Paragon d'INTEL

Comparaison

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

Comparaison



Communications dans un hypercube

- Chemins et routage dans un hypercube
- Broadcast dans un hypercube

Numérotation des sommets

Un m-cube est la donnée de:

- sommets numérotés de 0 à $2^m - 1$
- il existe une arête d'un sommet à un autre si les deux diffèrent seulement d'un bit dans leur écriture binaire

$$\begin{tikzcd} 00 \arrow[r, "0"] \arrow[d, "1"'] & 01 \arrow[d, "1"] \\ 10 \arrow[r, "0"'] & 11 \end{tikzcd}$$

Chemins et routage

- A, B deux sommets, $H(A, B)$ leur distance de Hamming (le nombre de bits qui diffèrent dans l'écriture)
- 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 (passent par des sommets différents)

Chemins et routage

Un routage possible: on corrige les poids faibles d'abord,

- $A = 1011$, $B = 1101$
- A xor $B = 0110$ (ou exclusif bit à bit)
- A envoie donc son message sur le lien 1 (c'est à dire vers 1001) avec entête 0100
- Puis 1001, lisant l'entête, renvoie sur son lien 2, c'est à dire vers $1101 = B$.

Egalement algorithme dynamique qui corrige les bits selon les liens disponibles

Plongements d'anneaux et de grilles

Code de Gray G_m de dimension m, défini récursivement:

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

- 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
- $\mid$ est la concaténation (de "listes" de mots binaires)

Codes de Gray: exemple

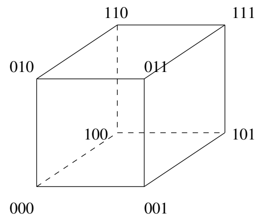
- $G_1 = \{0, 1\}$
- $G_2 = \{00, 01, 10, 11\}$
- $G_3 = \{000, 001, 010, 011, 111, 110, 101, 100\}$
- $G_4 = \{0000, 0001, 0010, 0011, 0111, 0110, 0101, 0100, 1100, 1101, 1110, 1111, 1011, 1010, 1001, 1000\}$

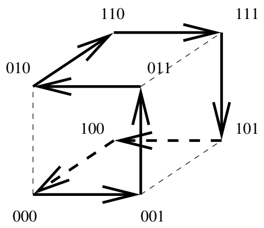
(imaginer la numérotation des processeurs sur l'anneau, dans cet ordre - un seul bit change à chaque fois)

Codes de Gray: intérêt

- Permet de définir un anneau de 2^m processeurs dans le m-cube grâce à G_m
- Permet de définir un réseau torique de taille $2^r \times 2^s$ dans un m-cube avec $r + s = m$ (utiliser le code $G_r \times G_s$)

Codes de Gray: exemple





Diffusion simple dans l'hypercube

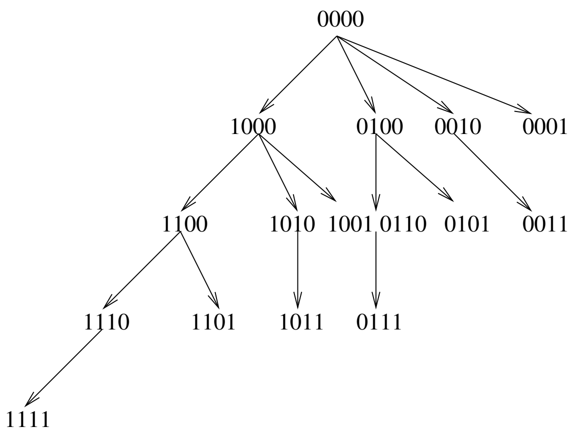
- on suppose que le processeur 0 veut envoyer un message à tous les autres
- algorithme naïf 1: le processeur 0 envoie à tous ses voisins, puis tous ses voisins à tous leurs voisins etc.: Très **redondant**!
- algorithme naïf 2: on utilise le code de Gray et on utilise la diffusion sur l'anneau

Arbres couvrants de l'hypercube

- Primitives: `send(cube-link,send-adr,L)`, `receive(cube-link,recv-adr,L)`
- 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
- Tout ceci en m phases, $i = m - 1 \to 0$

Arbres couvrants de l'hypercube

On construit à la fois un arbre couvrant de l'hypercube et l'algorithme de diffusion (ici $m = 4$):



Exemple

Pour $m = 4$,

- 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

Améliorations possibles, voir poly

Algorithmique sur grille 2D

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

Produit de matrices sur grille 2D

Distribution des données:

- Algorithmes par blocs: augmenter le grain de calcul et améliorer l'utilisation des mémoires hiérarchiques
- Distribution cyclique: équilibrer la charge

Distribution cyclique par blocs

Vecteur à M composantes sur p processeurs, $0 \leq m < M$ et $0 \leq q < p$:

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

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

Réseau linéaire par blocs (4 procs)

$M = 8$, $p = 4$, $r = 8$ (pour chaque colonne):

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3

Réseau linéaire cyclique

$M = 8$, $p = 4$, $r = 4$ (pour chaque colonne):

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3

Réseau en grille 2D par blocs (4×4 procs)

$M = 8$, $p = 4$, $r = 8$ en ligne et en colonne:

0.0	**0.0**	0.1	0.1	0.2	0.2	0.3	0.3
0.0	**0.0**	0.1	0.1	0.2	0.2	0.3	0.3
1.0	1.0	1.1	1.1	1.2	1.2	1.3	1.3
1.0	1.0	1.1	1.1	1.2	1.2	1.3	1.3
2.0	2.0	2.1	2.1	2.2	2.2	2.3	2.3
2.0	2.0	2.1	2.1	2.2	2.2	2.3	2.3
3.0	3.0	3.1	3.1	3.2	3.2	3.3	3.3
3.0	3.0	3.1	3.1	3.2	3.2	3.3	3.3

Réseau en grille 2D par blocs cycliques

$M = 8$, $p = 4$, $r = 4$ en ligne et en colonne:

0.0	0.1	0.2	0.3	**0.0**	0.1	0.2	0.3
1.0	1.1	1.2	1.3	1.0	1.1	1.2	1.3
2.0	2.1	2.2	2.3	2.0	2.1	2.2	2.3
3.0	3.1	3.2	3.3	3.0	3.1	3.2	3.3
0.0	0.1	0.2	0.3	**0.0**	0.1	0.2	0.3
1.0	1.1	1.2	1.3	1.0	1.1	1.2	1.3
2.0	2.1	2.2	2.3	2.0	2.1	2.2	2.3
3.0	3.1	3.2	3.3	3.0	3.1	3.2	3.3

En pratique?

- Version centralisée:
  - routines appelées avec données et résultats sur machine hôte
  - minimise le nombre de fonctions à écrire, permet de choisir la distribution des données la plus adaptée
  - MAIS coût prohibitif si on enchaîne les appels (penser à FFT 2D)
- Version distribuée:
  - données déjà distribuées, résultat distribué
  - passage à l'échelle ("scalable"), fonctions de redistribution des données
  - MAIS compromis: coût de redistribution/coût de l'algorithme avec rangement adapté/coût de l'algorithme avec rangement non-adapté

Produit de matrices sur grille 2D

- $C = C + AB$, avec A, B et C de taille $N \times N$
- $p = q^2$: on dispose d'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}

Au départ:

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

Principe de l'algorithme de Cannon

"Preskewing":

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

Principe de l'algorithme de Cannon (2)

Rotations sur A et B:

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

Principe de l'algorithme de Cannon

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

/* calcul du produit de matrice */
forall (k=1; k<=sqrt(P)) {
  LocalProdMat(A,B,C);
  VerticalRotation(B,downwards);
  HorizontalRotation(A,leftwards); }

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

Principe de l'algorithme de Fox

- Pas de mouvements de données initials
- Diffusions horizontales des diagonales de A (décalées vers la droite)
- Rotations verticales de B (de bas en haut)

Principe de l'algorithme de Fox

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

Principe de l'algorithme de Fox (2)

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

Principe de l'algorithme de Fox

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

/* calcul du produit de matrices */
broadcast(A(x,x));
forall (k=1; k<sqrt(P)) {
  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards);
  broadcast(A(k+x,k+x)); }
forall () {
  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)
- Accumulation sur les diagonales (décalées à chaque étape vers la droite) de C des résultats - représentées en gras dans les transparents ci-après
- Rotations verticales de B (de bas en haut)

Principe de l'algorithme de Snyder

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

Principe de l'algorithme de Snyder (2)

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

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=1;k<sqrt(P)) {
  GlobalSum(C(i,(i+k-1) mod sqrt(P)));
  LocalProdMat(A,B,C);
  VerticalRotation(B,upwards); }
GlobalSum(C(i,(i+sqrt(P)-1) mod sqrt(P)));
/* mouvements des donnees apres les calculs */
Transpose(B);
```

Analyse des algorithmes

Selon les modèles (CC, SF, WH): voir poly!

Algorithmique hétérogène

Allocation statique de tâches:

- $t_1, t_2, \ldots, t_p$ temps de cycle des processeurs
- B tâches identiques et indépendantes
- Principe: $c_i \times t_i = constante$, d'où:

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

Allocation statique optimale

```
Distribute(B,t1,t2,...,tn)
/* initialisation: calcule ci */
for (i=1;i<=p;i++)
  c[i]=...
/* incrementer iterativement les ci minimisant le temps */
while (sum(c[])<B) {
  find k in {1,...,p} st t[k]*(c[k]+1)=min{t[i]*(c[i]+1)};
  c[k] = c[k]+1;
return(c[]);
```


Algorithme incrémental

- Allocation optimale pour tout nombre d'atomes entre 1 et B
- Programmation dynamique avec $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$

Algorithme incrémental

```
Distribute(B,t1,t2,...tp)
/* Initialisation: aucune tache a distribuer m=0 */
for (i=1;i<=p;i++) c[i]=0;
/* construit iterativement l'allocation sigma */
for (m=1;m<=B;m++)
  find(k in {1,...p} st t[k]*(c[k]+1)=min{t[i]*(c[i]+1)});
  c[k]=c[k]+1;
  sigma[m]=k;
return(sigma[],c[]);
```


Exemple

$\#atomes$	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		$\sigma[10] = 3$

Ressources hétérogènes et équilibrage de charges

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
- ainsi de suite: la taille des données passe de $(n-1) \times b$ à $(n-2) \times b$ etc.

Allocation statique équilibrant les charges 1D

- Solution 1: on redistribue les colonnes restant à traiter à chaque étape entre les processeurs: pb, coût des communications
- Solution 2: trouver une allocation statique donnant un équilibrage de charges à chaque étapes

Allocation statique équilibrant les charges 1D

- 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

Allocation 1D

- Allouer périodiquement un motif de largeur B les blocs de colonnes
- B est un paramètre, par exemple si la matrice est $(nb) \times (nb)$, $B = n$ (mais meilleur pour le recouvrement calcul communication si $B << n$)
- Utiliser l'algorithme précédent en sens inverse: le bloc de colonne $1 \leq k \leq B$ alloué sur $\sigma(B - k + 1)$
- Cette distribution est quasi-optimale pour tout sous-ensemble $[i, B]$ de colonnes

Exemple

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

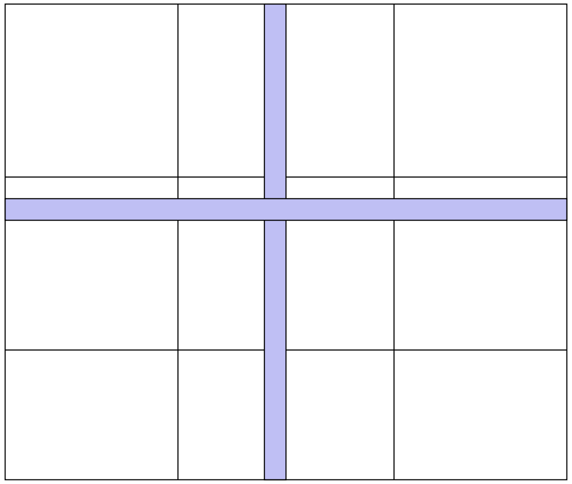
P_3	P_2	P_1	P_1	P_2	P_1	P_3	P_1	P_2	P_1

Allocation statique 2D

Exemple: 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 - cas homogène



Multiplication - cas inhomogène "régulier"

- 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

- Uniquement quand la matrice des temps de cycle $T = (t_{i,j})$ est de rang 1
- Exemple, rang 2, $P_{2,2}$ est partiellement inactif:

	1	$\frac{1}{2}$
1	$t_{11} = 1$	$t_{12} = 2$
$\frac{1}{3}$	$t_{21} = 3$	$t_{22} = 5$

- Exemple, rang 1, équilibrage parfait:

	1	$\frac{1}{2}$
1	$t_{11} = 1$	$t_{12} = 2$
$\frac{1}{3}$	$t_{21} = 3$	$t_{22} = 6$

Problème

Il faut optimiser:

- Objectif $Obj1$:

$$min_{\Sigma_i r_i = 1; \Sigma_j c_j = 1} max_{i,j} \{ r_i \times t_{i,j} \times c_j \}$$

- Objectif $Onj2$:

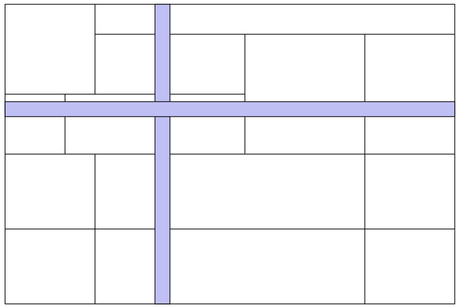
$$max_{r_i \times t_{i,j} \times c_j \leq 1} \left\{ \left(\sum_i r_i \right) \times \left(\sum_j c_j \right) \right\}$$

Régularité?

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

Partitionnement libre

où comment faire avec p (premier) processeurs??



Partitionnement libre

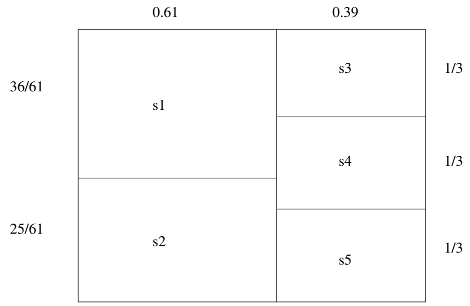
- p processeurs de vitesses $s_1, s_2, \ldots, s_n$ de somme 1 (normalisées)
- Partitionner le carré unité en p rectangles de surfaces $s_1, s_2, \ldots, s_n$
- Surface des rectangles $\Leftrightarrow$ vitesses relatives des processeurs
- Forme des rectangles $\Leftrightarrow$ minimiser les communications

Géométriquement

- Partitionner le carré unité en p rectangles d'aires fixées $s_1, s_2, \ldots, s_p$ afin de minimiser
  - soit la somme des demi-périmètres des rectangels 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 $\Sigma_{i=1}^{p} 2\sqrt{s_i} = 4.36$ atteinte lorsque tous les rectangles sont des carrés