INF 560
Calcul Parallèle et Distribué
Cours 6

Eric Goubault et Sylvie Putot

Ecole Polytechnique

26 janvier 2015

Plan du cours: algorithmique sur anneau de processeurs

- Communications collectives; implémentation sur un anneau
- Message Passing Interface (MPI)
- Produit matrice-vecteur
- Factorisation LU

Des modèles de programmation parallèle

- A mémoire partagée
  - communication non structurée
  - naturel à programmer, mais sujet à erreurs et peut ne pas bien passer à l'échelle
- Parallélisme de données (SIMD)
  - un espace partagé pour charger/stocker résultats, mais communications/synchronisations très limitées
- A mémoire distribuée
  - communications structurées sous forme de messages
  - un standard = bibliothèque MPI (Message Passing Interface)
- Correspondant à l'architecture, mais de façon souple: commun par exemple d'implémenter des abstractions par passage de message sur des machines à mémoire partagée

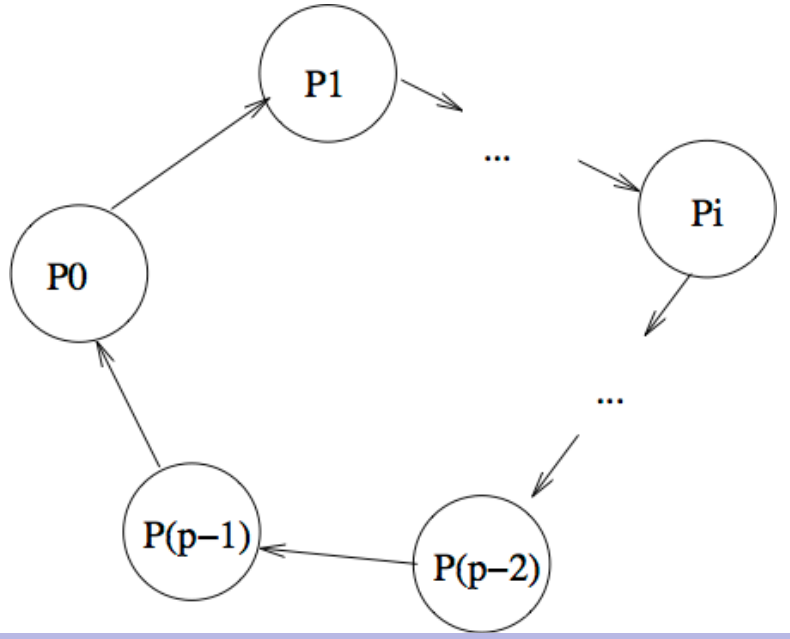
Les modèles modernes sont essentiellement mixtes

- Machines modernes: différents types d'abstractions à différentes échelles
- Par exemple CUDA: modèle par parallélisme de données pour pouvoir passer à l'échelle en nombre de coeurs mais avec:
  - de la mémoire partagée pour de la communication par blocs
  - des primitives de synchronisation
- Modèle très courant: mémoire partagée sur un noeud multi-coeur d'un cluster, et passage de messages entre coeurs

Modèle par passage de message

- Pas d'espace mémoire / variables partagées
- Chaque processeur a sa propre mémoire
- Les processeurs communiquent et se synchronisent par envoi de message sur un réseau

Architecture



Architecture en réseau: fonctionnement

- p processeurs en anneau, chacun a accès à:
  - son numéro d'ordre (entre 0 et $p-1$), par `my_num()`
  - nombre total de processeurs: `tot_proc_num` ($=p$)
- Mode SPMD:
  - tous les processeurs exécutent le même programme,
  - ils opèrent sur des données dans leur mémoire locale,
  - ils peuvent envoyer un message au processeur de numéro `(my_num()+1) modulo p` par `send(adr,L)` avec,
    - `adr` est l'adresse dans la mémoire de l'expéditeur de la première valeur à envoyer (on suppose les mots à envoyer rangés de façon contigue)
    - `L` la longueur du message
  - ils peuvent recevoir un message de `(my_num()-1) modulo p` par `receive(adr,L)`
    - `adr` est l'adresse dans la mémoire du récepteur
- Communication point à point: à tout `send` d'un expéditeur doit correspondre un `receive` d'un destinataire.
  - les expéditeurs/destinataires sont ici implicites (pas forcément le cas sur un réseau plus complexe)

Sémantique des primitives de communication

Plusieurs hypothèses possibles:

- Le plus classique: asynchrone avec `send` non bloquant et `receive` bloquant (mode par défaut en PVM, MPI)
  - L'emetteur envoie son message et n'attend pas: il ne sait pas à quel moment l'expéditeur l'a pris en compte
- Communication synchrone ("rendez-vous"): `send` et `receive` bloquants (OCCAM etc.)
  - l'appel du `send` rend la main quand le message est arrivé au récepteur et l'expéditeur a reçu une notification de réception
  - l'appel du `receive` rend la main quand le message est arrivé et l'expéditeur notifié
  - En général, codage de communication synchrone à l'aide de primitives asynchrones (receive bloquant):

```
rdv_send (destinataire, mess){
  send (destinataire, mess);
  receive (destinataire, ACK);
}
```

```
rdv_receive (expediteur, mess){
  receive (expediteur, mess);
  send (expediteur, ACK);
}
```

- Plus moderne: aucun bloquant (trois threads en fait: 1 pour calcul, 1 pour `send`, 1 pour `receive`)

Modélisation du coût d'une communication

Difficile à modéliser en général: ici envoyer/recevoir un message de longueur L (au voisin immédiat) coûtera:

$$\beta + L\tau$$

où:

- β est le coût d'initialisation (latence)
- τ (inverse du débit) mesure la vitesse de transmission en régime permanent

D'où envoyer/recevoir un message de longueur L de `my_num()+/-q` coûte $q(\beta + L\tau)$.

Quelques communications classiquement utilisées

- Synchronisation: pas d'échange d'information, les processus sont assurés que tous ont atteint le point de synchronisation
- Diffusion (broadcast): un processus distingué envoie un même message à tous les autres
- Diffusion personnalisée (scatter): un processus distingué envoie un message personnalisé à chacun des autres
- Echange total (all-to-all): chaque processus a une information qu'il partage avec tous les autres
- Echange total personnalisé (gossiping)
- Rassemblement (reduce): un processus reçoit un message de tous les autres; éventuellement réduction sur les données

Implémentées:

- En MPI: primitives pour chacune de ces communications
- Mais aussi ... les opérations shuffle sur des warps Cuda

Problème élémentaire: la diffusion

- C'est l'envoi par un P_k d'un message de longueur L (stocké à l'adresse *adr*) à tous les autres processeurs
- Par exemple processeur maître qui diffuse des informations d'initialisation aux esclaves
- Implémenté de façon efficace dans la plupart des librairies de communication (PVM, MPI etc.).

Implémentation (**receive** *bloquant*)

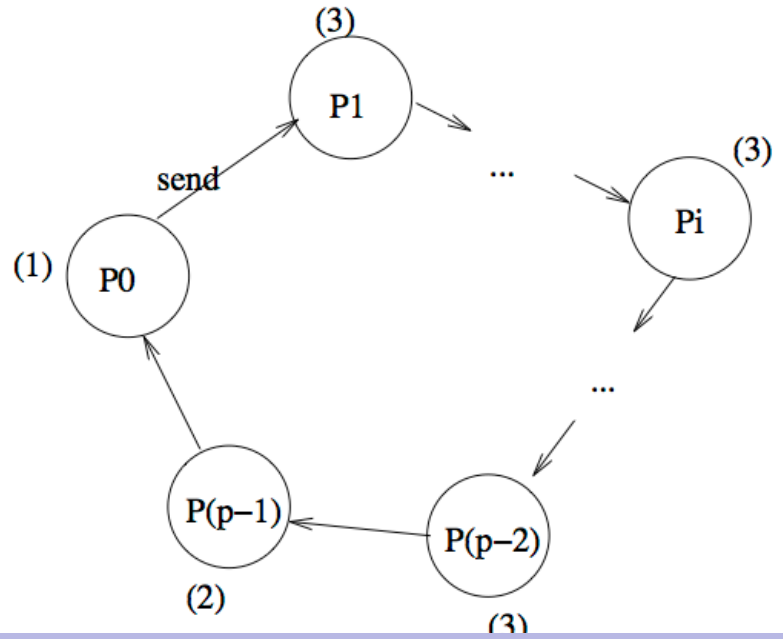
Message initialement stocké à l'adresse `adr` du processeur k, à la fin il sera à l'adresse `adr` de chaque processeur.

broadcast(k,adr,L) // emetteur initial=k

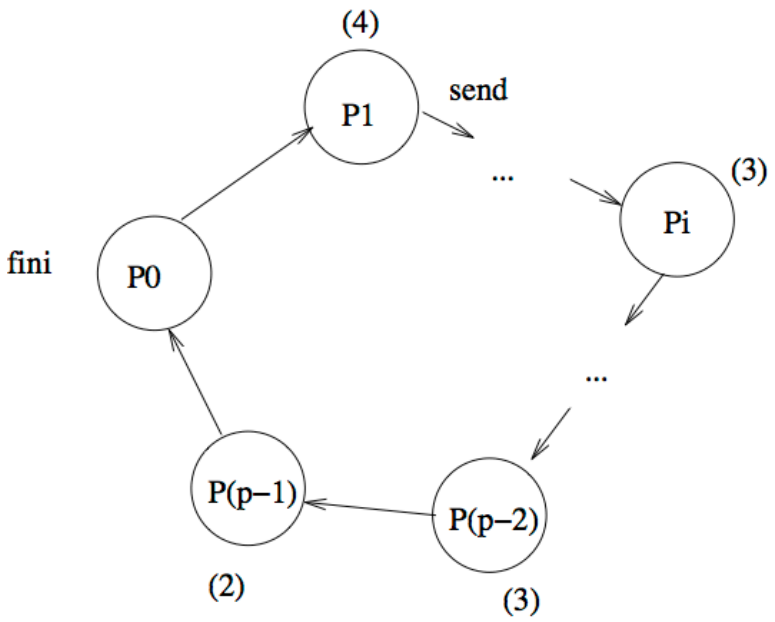
```
q = my_num();
p = tot_proc_num();
if (q == k)
  (1) send(adr,L);
else
  if (q == k-1 mod p)
    (2) receive(adr,L);
  else {
    (3) receive(adr,L);
    (4) send(adr,L);
  }
```

- Pas de parallélisme: chacun attend le message puis le renvoie
- Le receive doit être bloquant
- Le prédécesseur de P_k ne doit pas envoyer de message: P_k a déjà l'info, et surtout pas de receive correspondant!

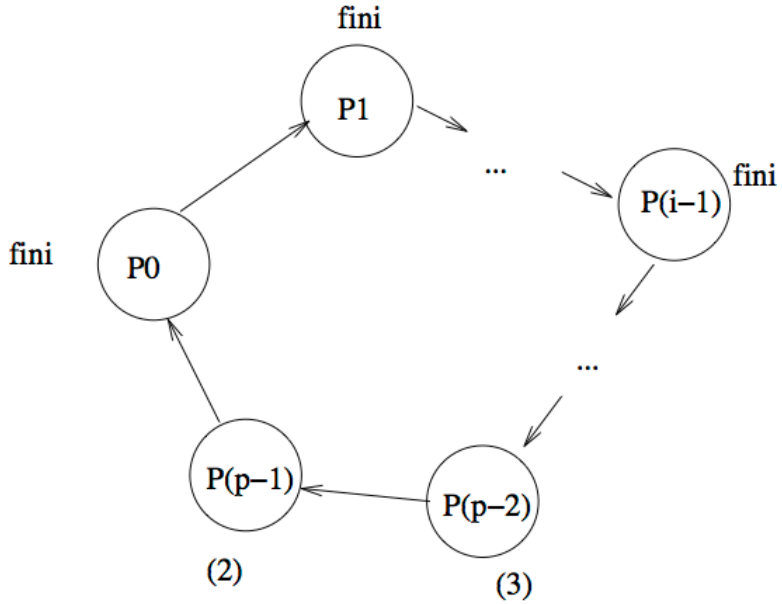
Exécution - temps 0 et κ=0



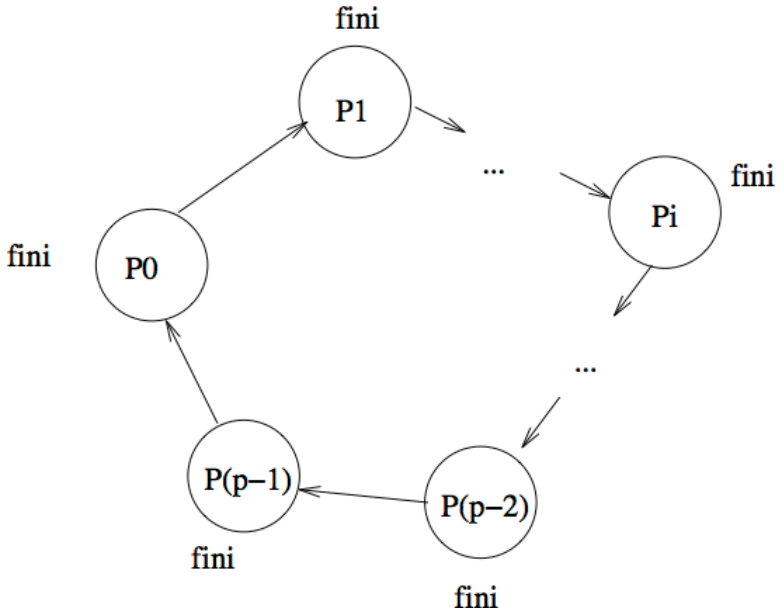
Exécution - temps $\beta + L\tau$



Exécution - temps $i(\beta + L\tau)$ $(i < p - 1)$



Exécution - temps $(p - 1)(\beta + L\tau)$



Diffusion personnalisée

- Envoi par P_k d'un message différent à tous les processeurs (en `adr[q]` dans P_k pour P_q)
- A la fin chaque processeur a son message à la location `adr`
- `send` non-bloquant, `receive` bloquant
- Opère en pipeline: recouvrement entre les différentes communications:
  - en commençant par envoyer le message destiné au processeur le plus éloigné (P_{k-1})
  - ainsi le temps total est le même que pour la diffusion: $(p-1)(\beta + L\tau)$

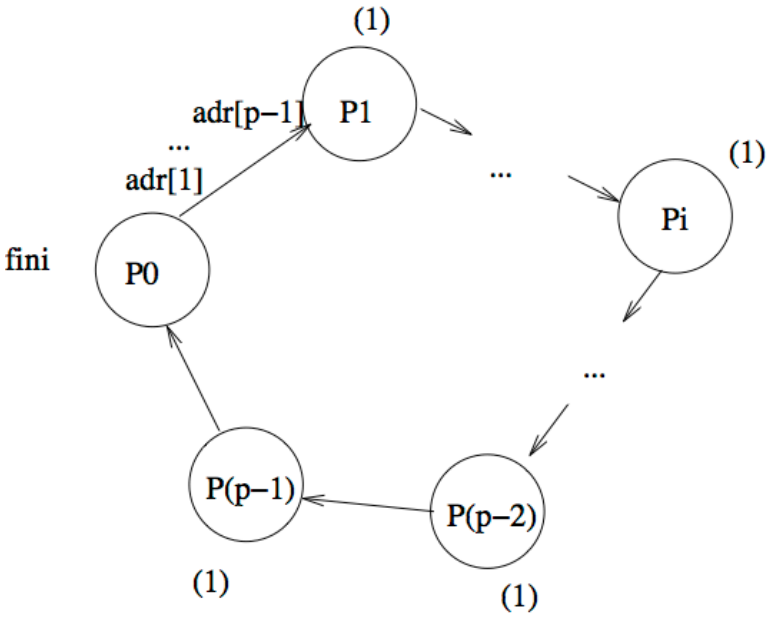
Programme

SCATTER(K,ADR,L)

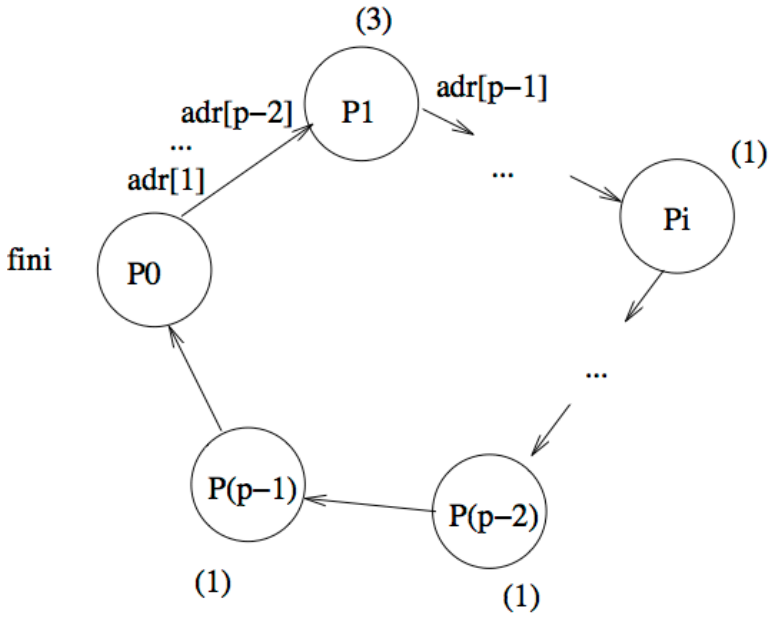
```
q = my_num();
p = tot_proc_num();
if (q == k) {
  for (i=1;i<p;i=i+1)
    send(adr[k-i mod p],L);
  adr <- adr[k];
}
else {
  (1) receive(adr,L);
  for (i=1;i<k-q mod p;i = i+1) {
    (2) send(adr,L);
    (3) receive(adr,L);
  }
}
```

Avec deux buffers, on peut aussi paralléliser l'émission d'un message et la réception du suivant

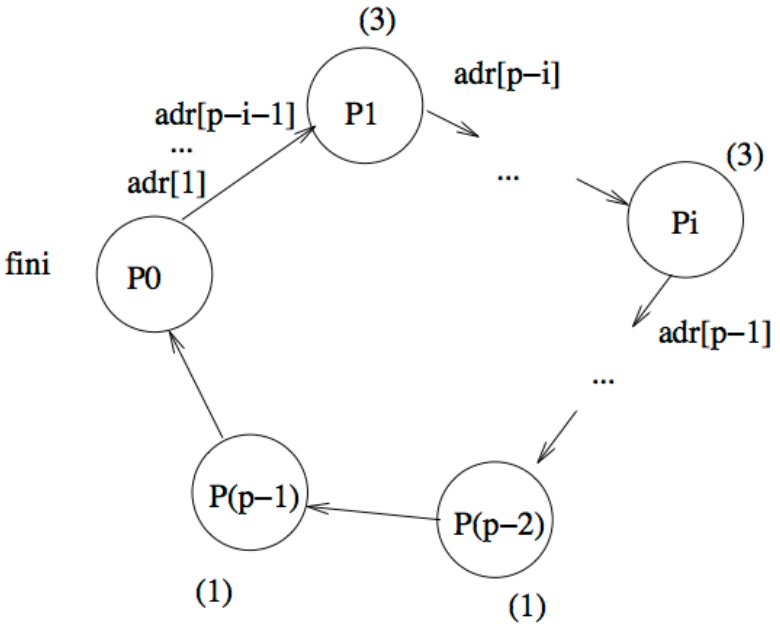
Exécution - temps 0 et κ=0



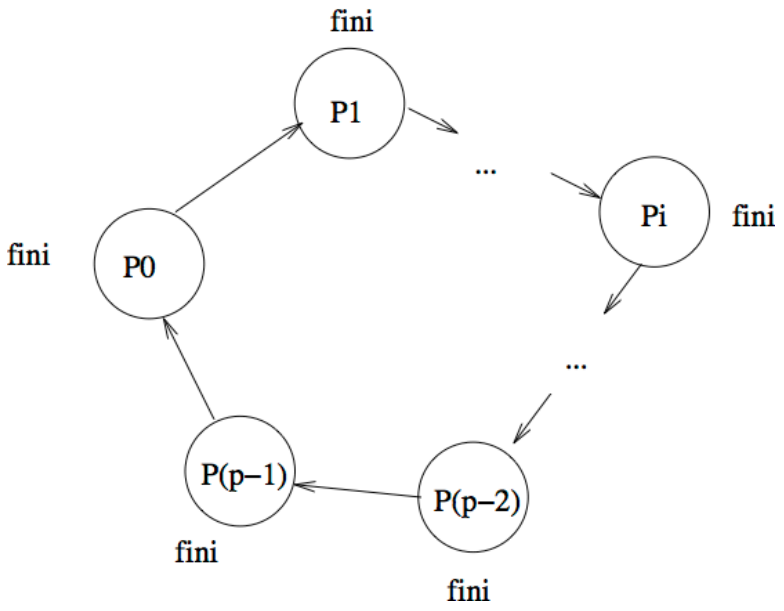
Exécution - temps $\beta + L\tau$



Exécution - temps $i(\beta + L\tau)$



Exécution - temps $(p-1)(\beta + L\tau)$



Echange total

- Chaque processeur veut envoyer un message à tous les autres
- Au départ chaque processeur dispose de son message à envoyer à tous les autres à la location `my_adr`
- A la fin, tous ont un tableau (le même) `adr[]` tel que `adr[q]` contient le message envoyé par le processeur `q`
- p diffusions simultanées: se fait aussi en $(p-1)(\beta + L\tau)$, en utilisant cette fois les liens de communications à plein régime.

ALL-TO-ALL(MY_ADR,ADR,L)

```
q = my_num();
p = tot_proc_num();
adr[q] = my_adr;
for (i=1;i<p;i++) {
  send(adr[q−i+1 mod p],L);
  receive(adr[q−i mod p],L);
}
```

Comme précédemment, on peut paralléliser émission et réception du suivant

Diffusion pipelinée

Peut-on diminuer le temps de la diffusion?

- Algorithme en $(p-1)(\beta + L\tau)$ simple mais peu efficace pour L grand
- Tronçonner le message à envoyer en r morceaux (r divise L)
- L'émetteur envoie successivement les r morceaux, qui commencent à circuler sur l'anneau: mode pipeliné
- Au début ces morceaux de messages sont dans `adr[0]`,...,`adr[r-1]` du processeur k
- A la fin: idem dans tous les processeurs

Programme

BROADCAST_PILEINE(K,ADR,L)

```
q = my_num();
p = tot_proc_num();
if (q == k)
  for (i=0;i<r;i++) send(adr[i],L/r);
else {
  if (q == k-1 mod p)
    for (i=0;i<r;i++) receive(adr[i],L/r);
  else {
    for (i=0;i<r;i++) {
      receive(adr[i],L/r);
      send(adr[i],L/r);
    }
  }
}
```


Temps d'exécution

- le premier morceau de longueur `L/r` du message sera arrivé au dernier processeur `k-1 mod p` en temps $(p-1)(\beta + \frac{L}{r}\tau)$ (diffusion simple)
- les $r-1$ autres morceaux arrivent les uns derrière les autres, d'où un temps supplémentaire de $(r-1)(\beta + \frac{L}{r}\tau)$
- En tout $(p-2+r)(\beta + \frac{L}{r}\tau)$

Optimisation du paramètre r

On cherche r qui minimise $(p-2+r)(\beta + \frac{L}{r}\tau)$:

- $r_{opt} = \sqrt{\frac{L(p-2)\tau}{\beta}}$
- Le temps optimal d'exécution est donc

$$\left(\sqrt{(p-2)\beta} + \sqrt{L\tau}\right)^2$$

- Quand L tend vers l'infini, ceci est asymptotiquement équivalent à $L\tau$, le facteur p devient négligeable!

MPI en très court

- Message Passing Interface - MPI-1 en 1994
- Bindings pour C, C++, Fortran, ... et des implémentations compatibles Cuda
- Mode SPMD, chaque processus a sa mémoire locale et un rang global
- Pas d'hypothèse sur le type d'interconnections entre processeurs
- On doit spécifier des communicateurs= collections de processus qui peuvent communiquer entre eux
- Primitives de communications point à point ou collectives comme définies précédemment

Produit matrice-vecteur sur anneau

Problème: calculer $y = Ax$ avec,

- A matrice de dimension $n \times n$
- x vecteur à n composantes (de 0 à $n-1$)
- sur un anneau de p processeurs, avec $r = n/p$ entier

Programme séquentiel

Le calcul produit matrice-vecteur revient au calcul de n produits scalaires indépendants, de la ligne i de A par le vecteur x:

```
for (i=1;i<=n;i++)
  for (j=1;j<=n;j++)
    y[i] = y[i]+a[i,j]*x[j];
```

On va distribuer aux p processeurs ($r = n/p$ entier)

- le calcul de ces produits scalaires
- mais aussi la mémoire: permet de résoudre des problèmes plus gros qu'en séquentiel

Principe de la distribution

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 `A` et les composantes de même rang des vecteurs `x` ety dans des tableaux locaux:

```
float a[r][n];
float x[r],y[r];
```

Remarque

Pour un calcul indépendant, il faudrait tout le vecteur x dupliqué sur chaque processeur: mais plus modulaire de supposer x distribué comme A et y: permet par exemple d'utiliser le même programme pour enchainer $z = By$ après $y = Ax$.

Principe du calcul -distribution initiale des données

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

Etape 0 (p étapes)

- Le processeur P_k calcule le produit partiel $y_k = A_{kk}x_k$, avec une notation par blocs (x_k est le bloc k de x sur r)
- Pendant ce temps on décale les blocs de vecteur x sur l'anneau

$$P_0 \begin{pmatrix} A_{00} & A_{01} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{10} & A_{11} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}$$

$$P_1 \begin{pmatrix} \bullet & \bullet & A_{22} & A_{23} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{32} & A_{33} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$P_2 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{44} & A_{45} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{54} & A_{55} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

$$P_3 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{66} & A_{67} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{76} & A_{77} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

Etape 1

- P_k vient de recevoir $x_{(k-1)}$ modulo p et calcule $y_k = y_k + A_{k,k-1}x_{(k-1)}$ modulo p
- Pendant ce temps on décale les blocs de x sur l'anneau

$$P_0 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{06} & A_{07} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{16} & A_{17} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

$$P_1 \begin{pmatrix} A_{20} & A_{21} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{30} & A_{31} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}$$

$$P_2 \begin{pmatrix} \bullet & \bullet & A_{42} & A_{43} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{52} & A_{53} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$P_3 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{64} & A_{65} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{74} & A_{75} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

Etape 2

$$P_0 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{04} & A_{05} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{14} & A_{15} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

$$P_1 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{26} & A_{27} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{36} & A_{37} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

$$P_2 \begin{pmatrix} A_{40} & A_{41} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{50} & A_{51} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}$$

$$P_3 \begin{pmatrix} \bullet & \bullet & A_{62} & A_{63} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{72} & A_{73} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

Etape 3 = $p-1$ (dernière étape)

- P_k contient le bloc y_k
- Dernière communication sur x inutile mais remet x en place

$$P_0 \begin{pmatrix} \bullet & \bullet & A_{02} & A_{03} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{12} & A_{13} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$P_1 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{24} & A_{25} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{34} & A_{35} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

$$P_2 \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{46} & A_{47} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{56} & A_{57} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

$$P_3 \begin{pmatrix} A_{60} & A_{61} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{70} & A_{71} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad temp \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}$$

Programme

matrice-vecteur(A,x,y)

```
q = my_num();
p = tot_proc_num();
for (step=0;step<p;step++) {
  send(x,r);
  for (i=0;i<r;i++)
    for (j=0;j<r;j++)
      y[i] = y[i]+a[i,(q-step mod p)r+j]*x[j];
  receive(temp,r);
  x <- temp
}
```

Communications en parallèle avec les calculs en utilisant un buffer temporaire pour x

Performances

- En notant τ_a le temps de calcul élémentaire (un opération), τ_c le temps de communication élémentaire
- Il y a p étapes, chacune de temps égal le plus long entre le calcul local et le temps de communication: $max(r^2\tau_a, \beta + r\tau_c)$
- D'où temps total de $p * max(r^2\tau_a, \beta + r\tau_c)$
- Quand n est grand, pour p fixé, $r^2\tau_a$ devient prépondérant, d'où asympotiquement un temps de $\frac{n^2}{p}\tau_a$: efficacité 1!

Remarque: on aurait aussi pu procéder à un échange total de x au début, au prix d'un peu plus de mémoire...

Factorisation LU

Problème: résolution d'un système linéaire dense $Ax = b$ avec A inversible, par décomposition LU.

LU-sequentiel(A,n)

```
for (k=0;k<n-1;k++) {
  prep(k): for (i=k+1;i<n;i++)
             a[i,k]=a[i,k]/a[k,k];
  for (j=k+1;j<n;j++) {
    update(k,j): for (i=k+1;i<n;i++)
                   a[i,j]=a[i,j]-a[i,k]*a[k,j];
  }
}
```

Pas de pivotage ici: peut se généraliser sans communication supplémentaire

Distribution

- Naturel de distribuer les colonnes aux différents processeurs
- On suppose que cette distribution nous est donnée par une fonction `alloc` telle que `alloc(k)=q` veut dire que la $k$ième colonne est affectée à la mémoire locale de P_q
- On utilise la fonction `broadcast`, pour faire en sorte qu'à l'étape k, le processeur qui possède la colonne k la diffuse à tous les autres

Programme - ici alloc(k)=k

LU-broadcast(A,n)

```
q = my_num();
p = tot_proc_num();
for (k=0;k<n-1;k++) {
  if (k == q) {
    prep(k): for (i=k+1;i<n;i++)
      buffer[i-k-1] = -a[i,k]/a[k,k];
    broadcast(k,buffer,n-k);
  }
  else {
      receive(buffer,n-k);
      update(k,q): for (i=k+1;k<n;k++)
        a[i,q] = a[i,q]+buffer[i-k-1]*a[k,q];
  }
}
```

Voir poly pour version la plus générale.

Allocation de colonnes à p processeurs

- Le nombre de données à traiter varie au cours des étapes (on met à jour de moins en moins de colonnes)
- le volume de calcul n'est pas proportionnel au volume des données: quand un processeur a par exemple r colonnes consécutives, le dernier processeur a plus de calcul (la dernière colonne est mise à jour $n-1$ fois) que le premier
- Il faut donc une allocation qui réussisse à équilibrer le volume des données et du travail!
- Equilibrage de charge à chaque étape de l'algorithme, et pas seulement global
- Allocation cyclique: la colonne j est allouée à P_j modulo p

Cas de l'allocation cyclique par colonnes

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
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}
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}
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}
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}

Allocation cyclique - k=0

P_0 $p_0; b$	P_1	P_2	P_3	P_0	P_1	P_2	P_3
U_{00}	A_{01}	A_{02}	A_{03}	A_{04}	A_{05}	A_{06}	A_{07}
L_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}
L_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}
L_{30}	A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}	A_{37}
L_{40}	A_{41}	A_{42}	A_{43}	A_{44}	A_{45}	A_{46}	A_{47}
L_{50}	A_{51}	A_{52}	A_{53}	A_{54}	A_{55}	A_{56}	A_{57}
L_{60}	A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}	A_{67}
L_{70}	A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}	A_{77}

Allocation cyclique - κ=0

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
	r; $u_{0,1}$	r; $u_{0,2}$	r; $u_{0,3}$	$u_{0,4}$			
U_{00}	U_{01}	U_{02}	U_{03}	U_{04}	A_{05}	A_{06}	A_{07}
L_{10}	A'_{11}	A'_{12}	A'_{13}	A'_{14}	A_{15}	A_{16}	A_{17}
L_{20}	A'_{21}	A'_{22}	A'_{23}	A'_{24}	A_{25}	A_{26}	A_{27}
L_{30}	A'_{31}	A'_{32}	A'_{33}	A'_{34}	A_{35}	A_{36}	A_{37}
L_{40}	A'_{41}	A'_{42}	A'_{43}	A'_{44}	A_{45}	A_{46}	A_{47}
L_{50}	A'_{51}	A'_{52}	A'_{53}	A'_{54}	A_{55}	A_{56}	A_{57}
L_{60}	A'_{61}	A'_{62}	A'_{63}	A'_{64}	A_{65}	A_{66}	A_{67}
L_{70}	A'_{71}	A'_{72}	A'_{73}	A'_{74}	A_{75}	A_{76}	A_{77}

Puis...

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
					$u_{0,5}$	$u_{0,6}$	$u_{0,7}$
U_{00}	U_{01}	U_{02}	U_{03}	U_{04}	U_{05}	U_{06}	U_{07}
L_{10}	A'_{11}	A'_{12}	A'_{13}	A'_{14}	A'_{15}	A'_{16}	A'_{17}
L_{20}	A'_{21}	A'_{22}	A'_{23}	A'_{24}	A'_{25}	A'_{26}	A'_{27}
L_{30}	A'_{31}	A'_{32}	A'_{33}	A'_{34}	A'_{35}	A'_{36}	A'_{37}
L_{40}	A'_{41}	A'_{42}	A'_{43}	A'_{44}	A'_{45}	A'_{46}	A'_{47}
L_{50}	A'_{51}	A'_{52}	A'_{53}	A'_{54}	A'_{55}	A'_{56}	A'_{57}
L_{60}	A'_{61}	A'_{62}	A'_{63}	A'_{64}	A'_{65}	A'_{66}	A'_{67}
L_{70}	A'_{71}	A'_{72}	A'_{73}	A'_{74}	A'_{75}	A'_{76}	A'_{77}

Allocation cyclique - κ=1

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
	p_1; b						
U_{00}	U_{01}	U_{02}	U_{03}	U_{04}	U_{05}	U_{06}	U_{07}
L_{10}	U_{11}	A'_{12}	A'_{13}	A'_{14}	A'_{15}	A'_{16}	A'_{17}
L_{20}	L_{21}	A'_{22}	A'_{23}	A'_{24}	A'_{25}	A'_{26}	A'_{27}
L_{30}	L_{31}	A'_{32}	A'_{33}	A'_{34}	A'_{35}	A'_{36}	A'_{37}
L_{40}	L_{41}	A'_{42}	A'_{43}	A'_{44}	A'_{45}	A'_{46}	A'_{47}
L_{50}	L_{51}	A'_{52}	A'_{53}	A'_{54}	A'_{55}	A'_{56}	A'_{57}
L_{60}	L_{61}	A'_{62}	A'_{63}	A'_{64}	A'_{65}	A'_{66}	A'_{67}
L_{70}	L_{71}	A'_{72}	A'_{73}	A'_{74}	A'_{75}	A'_{76}	A'_{77}

Allocation cyclique - κ=1

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
		r; $u_{1,2}$	r; $u_{1,3}$	r; $u_{1,4}$	$u_{1,5}$		
U_{00}	U_{01}	U_{02}	U_{03}	U_{04}	U_{05}	U_{06}	U_{07}
L_{10}	U_{11}	U_{12}	U_{13}	U_{14}	A'_{15}	A'_{16}	A'_{17}
L_{20}	L_{21}	A''_{22}	A''_{23}	A''_{24}	A''_{25}	A'_{26}	A'_{27}
L_{30}	L_{31}	A''_{32}	A''_{33}	A''_{34}	A''_{35}	A'_{36}	A'_{37}
L_{40}	L_{41}	A''_{42}	A''_{43}	A''_{44}	A''_{45}	A'_{46}	A'_{47}
L_{50}	L_{51}	A''_{52}	A''_{53}	A''_{54}	A''_{55}	A'_{56}	A'_{57}
L_{60}	L_{61}	A''_{62}	A''_{63}	A''_{64}	A''_{65}	A'_{66}	A'_{67}
L_{70}	L_{71}	A''_{72}	A''_{73}	A''_{74}	A''_{75}	A'_{76}	A'_{77}

Cout de la mise à jour (update) des colonnes

- Coût des phases de mise à jour (update) de la colonne j:
  - à toutes les étapes $k = 0$ à $k = j - 1$
  - un coût de $n - k - 1$ pour l'étape k (éléments en position $k + 1$ à $n - 1$)
  - d'où un coût total pour la colonne j de

$$t = \sum_{k=0}^{j-1} (n - k - 1)\tau_a$$

- Pour la colonne n:

$$\sum_{k=0}^{j-1} (n - k - 1)\tau_a = \frac{n(n-1)}{2}\tau_a$$

Temps de calcul total

- Le chemin critique d'exécution est:
  $prep_0(0) \rightarrow update_1(0,1), prep_1(1) \rightarrow update_2(1,2), prep_2(2) \rightarrow \ldots$
  ($\rightarrow$ = communication vers proc. voisin)
- $n\beta + \frac{n^2}{2}\tau_c$ pour les $n-1$ communications ($n^2/2$ données)
- $\frac{n^2}{2}\tau_a$ pour les phases de préparation prep
- Pour l'update des $r = n/p$ colonnes qui se fait séquentiellement sur chaque processeur (en parallèle sur tous les processeurs), environ $r\frac{n(n-1)}{2}\tau_a$
- D'où un coût de $\frac{n^3}{2p}$ pour les update des p processeurs: terme dominant si $p << n$ et efficacité excellente asymptotiquement

Mais:

- Recouvrement des communications, mais pas entre communication et calcul
- Procédure de diffusion abstraite, sans connaissance de la topologie du réseau

Anneau: recouvrement communication/calcul

LU-pipeline(A,n)

```
q = my_num();
p = tot_proc_num();
l = 0;
for (k=0;k<n-1;k++) {
  if (k == q mod p) {
    prep(k): for (i=k+1;i<n;i++)
      buffer[i-k-1] = a[i,l]/a[k,l];
    l++; send(buffer,n-k); }
  else { receive(buffer,n-k);
    if (q != k-1 mod p) send(buffer,n-k); }
  for (j=l;j<r;j++)
    update(k,j): for (i=k+1;k<n;k++)
      a[i,j] = a[i,j]-buffer[i-k-1]*a[k,j];
}
```

- Ici r colonnes par processeur: l est le numéro de colonne en cours de traitement parmi les r
- On utilise le réseau en anneau: send = envoi au voisin
- Dès qu'un processeur reçoit une colonne, il la fait circuler

P_0	P_1	P_2	P_3
$prep(0)$			
$send(0)$	$receive(0)$		
$update(0,4)$	$send(0)$	$receive(0)$	
$update(0,8)$	$update(0,1)$	$send(0)$	$receive(0)$
$update(0,12)$	$update(0,5)$	$update(0,2)$	$update(0,3)$
	$update(0,9)$	$update(0,6)$	$update(0,7)$
	$update(0,13)$	$update(0,10)$	$update(0,11)$
	$prep(1)$	$update(0,14)$	$update(0,15)$
	$send(1)$	$receive(1)$	
	$update(1,5)$	$send(1)$	$receive(1)$
$receive(1)$	$update(1,9)$	$update(1,2)$	$send(1)$
$update(1,4)$	$update(1,13)$	$update(1,6)$	$update(1,3)$
$update(1,8)$		$update(1,10)$	$update(1,7)$
$update(1,12)$		$update(1,14)$	$update(1,11)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$

Encore améliorable...

- Certaines communications ont lieu en parallèle avec des calculs: les premiers processeurs commencent leur calcul alors que les dernières reçoivent encore le pivot
- On peut alors pipeliner les étapes de façon emboitée
- Mais des bulles d'inactivité se forment quand c'est le tour du processeur le plus en retard de devenir pivot

Le pivot de la prochaine étape aurait pu, au lieu de commencer par exécuter tous ses update, faire seulement le premier update, puis insérer la phase suivante de pivot + envoi au voisin, avant de finir ses update. Pour P_1 cela donne:

- `update(0,1)`
- `prep(1)`
- Envoi vers P_2
- `update(0,j)` pour $j = 1 \ mod \ p$ et $j > 1$
- etc.

En autorisant de plus communication et calcul simultanés sur le même processeur (par exemple avec des threads):

P_0	P_1	P_2	P_3
$prep(0)$			
$sd(0)\|\|up(0,4)$	$rcv(0)$		
$up(0,8)$	$sd(0)\|\|up(0,1)$	$rcv(0)$	
$up(0,12)$	$prep(1)$	$sd(0)\|\|up(0,2)$	$rcv(0)$
	$sd(1)\|\|up(0,5)$	$rcv(1)\|\|up(0,6)$	$up(0,3)$
	$up(0,9)$	$sd(1)\|\|up(0,10)$	$rcv(1)\|\|up(0,7)$
$rcv(1)$	$up(0,13)$	$up(0,14)$	$sd(1)\|\|up(0,11)$
$up(1,4)$	$up(1,5)$	$up(1,2)$	$up(0,15)$
$up(1,8)$	$up(1,9)$	$prep(2)$	$up(1,3)$
$up(1,12)$	$up(1,13)$	$sd(2)\|\|up(1,6)$	$rcv(2)\|\|up(1,7)$
$rcv(2)$		$up(1,10)$	$sd(2)\|\|up(1,11)$
$sd(2)\|\|up(2,4)$	$rcv(2)$	$up(1,14)$	$up(1,15)$
...	...	...	...

En conclusion

Implémentation:

- Implémenter une topologie logique parfois laissé à l'utilisateur: fonction permettant à chaque processeur d'identifier ses voisins
- Raisonnable en général qu'une topologie logique ressemblant à la topologie physique produira des bonnes performances

Quelques principes (éventuellement antagonistes):

- Envoyer des messages pas trop courts (latence d'envoi des messages) - mais on a vu des contre-exemples...
- Envoyer les données dès que possible
- Essayer de recouvrir communication et calcul
- Distribution de blocs de données contigues sur les processeurs pour réduire les coûts de communication
- Distribution cyclique de données pour un meilleur équilibrage de charge

La suite

- En TD: on commence les projets
  - M'envoyer par mail (Sylvie.Putot@polytechnique.edu) d'ici la fin de semaine le titre de votre projet en donnant les grandes lignes en qq phrases, et nom du binôme éventuel
- La semaine prochaine: algorithmique distribuée, suite et fin