Parallélisme

ERIC GOUBAULT
COMMISSARIAT À L'ENERGIE ATOMIQUE
SACLAY

PROBLÈMES D'ORDONNANCEMENT

- Revenons à la mémoire partagée... et aux PRAM!
- Problème important: paralléliser les nids de boucle
- ou au moins comprendre ce qui est intrinsèquement séquentiel, de ce qui peut être calculé en parallèle

LOI D'AMDAHL

- s pourcent séquentiel,
- sur p processeurs,
- accélération du calcul complet par rapport à un calcul séquentiel sera au maximum de,

$$\frac{1}{s+\frac{1-s}{p}}$$

(maximum de $\frac{1}{s}$)

Conséquence: même si on parallélise 80 pourcent d'un code (le reste étant séquentiel), on ne pourra jamais dépasser, quel que soit la machine cible, une accélération d'un facteur 5!

EXEMPLE DE NID DE BOUCLE

```
for (i=1;i<=N;i++) {
  for (j=i;j<=N+1;j++)
    for (k=j-i;k<=N;k++) {
      S1;
      S2;
    }
  for (r=1;r<=N;r++)
    S3;
}
```

Explication

- Ceci n'est pas un nid de boucle parfait:
- `S1` et `S2` sont englobées par les boucles `i`, `j` et `k`
- alors que `S3` est englobée par les boucles `i` et `r`

Vecteurs d'itération

- Les itérations de n boucles parfaitement imbriquées sont représentées par un vecteur de dimension n
- Par exemple, pour `S3` on a un vecteur d'itération en (i, r) dont le domaine est $1 \leq i, r \leq N$
- Pour `S1` on a un vecteur d'itération en (i, j, k) dont le domaine est $1 \leq i \leq N$, $i \leq j \leq N+1$ et $j - i \leq k \leq N$.
- On note une instance de S à l'itération I, $S(I)$
- On suppose que les domaines d'itération forment des polyèdres.

Ordre séquentiel d'exécution

Définit l'ordre d'exécution "par défaut":

- Pour les nids de boucles non-parfaits, les domaines d'itérations sont incomparables a priori
- Il suffit de "compléter" les vecteurs d'itération de façon cohérente: $I \rightarrow \tilde{I}$
- $$\begin{aligned} S(I) <_{seq} T(J) \Leftrightarrow & \\ & (\tilde{I} <_{seq} \tilde{J}) \text{ ou} \\ & (\tilde{I} = \tilde{J} \text{ et } S <_{text} T) \end{aligned}$$

Dépendances de données

- On veut mettre en parallèle certaines instructions: une partie de l'ordre séquentiel est à respecter absolument, une autre pas (permutation possible d'actions)
- On va définir un ordre partiel (Bernstein), ordre minimal à respecter pour produire un code sémantiquement équivalent à l'ordre initial
- En fait, l'ordre séquentiel va être une *extension* de l'ordre partiel de Bernstein
- Cet ordre partiel est défini à partir de 3 types de dépendances de données: flot, anti et sortie.

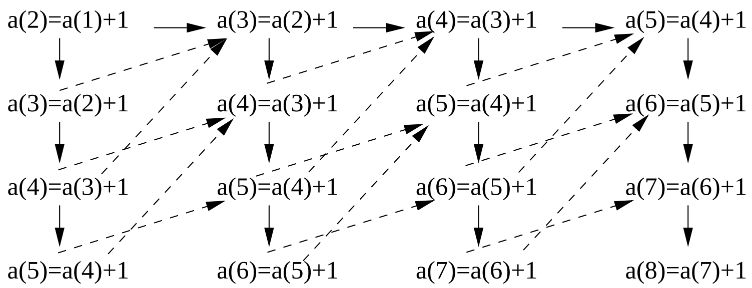
Flot, anti et sortie...

- Toujours dirigées par l'ordre séquentiel
- Dépendance de flot de $S(I)$ vers $T(J)$: si un emplacement mémoire commun est en écriture pour $S(I)$ et en lecture pour $T(J)$
- Dépendance anti de $S(I)$ vers $T(J)$: si un emplacement mémoire commun est en lecture pour $S(I)$ et en écriture pour $T(J)$
- Dépendance de sortie de $S(I)$ vers $T(J)$: si un emplacement mémoire commun est en écriture pour $S(I)$ et en écriture pour $T(J)$

Exemple

```
for (i=1;i<=N;i++)
  for (j=1;j<=N;j++)
    a(i+j) = a(i+j-1)+1;
```


Dépendances



Calcul des dépendances

- Supposons que $S(I)$ et $T(J)$ accèdent au même tableau a (S en écriture, T en lecture)
- $S(I) : a(f(I)) = ...$ et $T(J) : ... = a(g(J))$;
- L'accès au tableau est commun si $f(I) = g(J)$
- Si f et g sont des fonctions affines à coefficients entier: peut se tester en temps polynomial

Calcul des dépendances

- Chercher une dépendance de flot par exemple revient à prouver en plus $S(I) <_{seq} T(J)$
- Encore une fois: en général résolution de systèmes d'égalités et d'inégalités affines
- Chercher les dépendances *directes*: plus compliqué (par programmation linéaire, problèmes d'optimisation dans les entiers)

Exemple

- $f(I) = i + j$ et $g(J) = i + j - 1$
- On cherche dépendance entre écriture $S(i', j')$ et lecture $S(i, j)$
- $f(I) = g(J) \Leftrightarrow i' + j' = i + j - 1$
- $S(i', j') <_{seq} S(i, j) \Leftrightarrow \big((i' \leq i - 1)$ ou $(i = i'$ et $j' \leq j - 1)\big)$

Exemple

- Dépendance de flot directe implique résoudre:

  $max_{<_{seq}}\{(i', j') \mid (i', j') <_{seq} (i, j), i'+j' = i+j-1, 1 \leq i, i', j, j' \leq$

- Solution:

  $$\begin{array}{l} (i, j-1) \text{ si } j \geq 2 \\ (i-1, j) \text{ si } j = 1 \end{array}$$

- Antidépendance directe de $S(i, j)$ vers $S(i', j')$ implique résoudre:

  $min_{<_{seq}}\{(i', j') \mid (i, j) <_{seq} (i', j'), i'+j' = i+j-1, 1 \leq i, i', j, j' \leq$

- Solution (pour $j \geq 3$, $i \leq N - 1$):

  $$(i + 1, j - 2)$$

Approximation des dépendances

Graphe de Dépendance Etendu (GDE):

- Sommets: instances $S_i(I)$, $1 \leq i \leq s$ et $I \in D_{S_i}$
- Arcs: $S(I) \rightarrow T(J)$ pour chaque dépendance

Quelques définitions

- Ensemble des paires de dépendances entre S et T:
$$\{(I,J) \mid S(I) \to T(J)\} \subseteq \mathbb{Z}^{n_S} \times \mathbb{Z}^{n_T}$$
- Ensemble de distance de ces dépendances:
$$\{(\tilde{J} - \tilde{I}) \mid S(I) \to T(J)\} \subseteq \mathbb{Z}^{n_{S,T}}$$
- Problème: on ne peut calculer le GDE à la compilation! (de toutes façons, est trop gros!)

Graphe de Dépendance Réduit

...ou GDR:

- Sommets: instructions S_i $(1 \leq i \leq s)$
- Arcs: $e : S \to T$ si il existe au moins un arc $S(I) \to T(J)$ dans le GDE
- Etiquette: $w(e)$ décrivant un sous-ensemble D_e de $\mathbb{Z}^{n_{S,T}}$

Le tri topologique du GDR permet d'avoir une idée des portions séquentielles, et des portions parallélisables.

Quelques abstractions

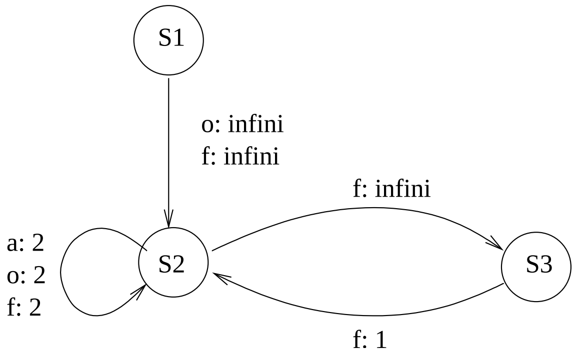
...des étiquettes du GDR. Par exemple, niveaux de dépendance (GDRN):

- Une dépendance entre $S(I)$ et $T(J)$ est boucle indépendante si elle a lieu au cours d'une même itération des boucles englobant S et T
- Sinon elle est portée par la boucle...
- Etiquetage en conséquence, de $e : S \to T$ du GDR:
  - $l(e) = \infty$ si $S(I) \to T(J)$ avec $\tilde{J} - \tilde{I} = 0$
  - $l(e) \in [1, n_{S,T}]$ si $S(I) \to T(J)$, et la première composante non nulle de $\tilde{J} - \tilde{I}$ est la $l(e)^{ieme}$ composante

Exemple

```
for (i=2;i<=N;i++) {
  S1: s(i) = 0;
  for (j=1;j<i-1;j++)
    S2: s(i) = s(i)+a(j,i)*b(j);
  b(i) = b(i)-s(i);
}
```

GDRN



Vecteurs de direction

Dans notre exemple:

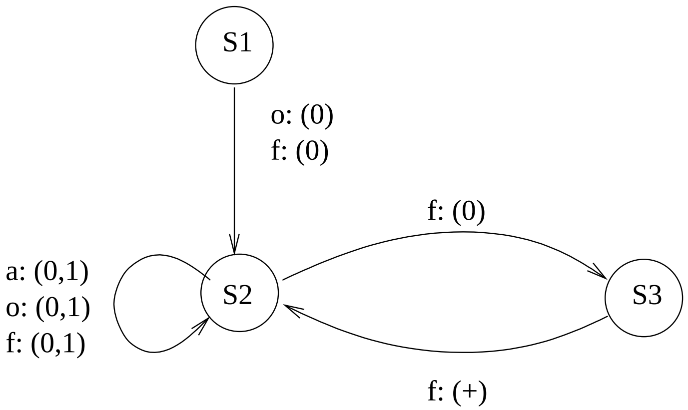
- lecture $a(i+j-2)$-écriture $a(i+j)$: ensemble de distance $\{(1,-2)\}$
- écriture $a(i+j)$-lecture $a(i+j-1)$: ensemble de distance $\{(1,0),(0,1)\}$
- écriture $a(i+j)$-écriture $a(i+j)$: ensemble de distance $\{(1,-1)\}$

Vecteurs de direction

Sont des abstractions de ces ensembles de distance:

- on note $z+$ pour une composante si toutes les distances sur cette composante ont au moins la valeur z
- on note $z-$ pour une composante si toutes les distances sur cette composante ont au plus la valeur z
- on note $+$ à la place de $1+$, $-$ à la place de $-1-$
- on note $*$ si la composante peut prendre n'importe quelle valeur
- on note z si la composante a toujours la valeur z

GDRV



Transformations de boucles

- Distribution de boucle
- Torsion de boucle
- Inversion de boucle
- Permutation de boucle
- etc.

Algorithmes de parallélisation de boucles

- Allen et Kennedy (basé sur le GDRN) : distribution de boucles
- Lamport - transformations unimodulaires (basé sur le GDRV) : torsion, inversion, permutation de boucles

Distribution de boucles

- Toute instance d'une instruction S peut être exécutée avant toute instance de T si on n'a jamais $T(J) \to S(I)$ dans le GDE
- Plusieurs cas à considérer, pour le code:

```
for (i=...) {
  S1;
  S2;
}
```


Cas 1

$S1 \to S2$ mais on n'a pas de dépendance de $S2$ vers $S1$, alors on peut transformer le code en:

```
for (i=...)
  S1;
for (i=...)
  S2;
```

Cas 2

$S2 \to S1$ mais on n'a pas de dépendance de $S1$ vers $S2$, alors on peut transformer le code en:

```
for (i=...)
  S2;
for (i=...)
  S1;
```


Fusion de boucles

```
forall (i=1;i<=N;i++)
 D[i]=E[i]+F[i];
forall (j=1;j<=N;j++)
 E[j]=D[j]*F[j];
```

devient:

```
forall (i=1;i<=N;i++)
 {
  D[i]=E[i]+F[i];
  E[j]=D[j]*F[j];
 }
```

Cela permet une vectorisation et donc une réduction du coût des boucles parallèles.

Composition de boucles

```
forall (j=1;j<=N;j++)
 forall (k=1;k<=N;k++)
  ...
```

devient,

```
forall (i=1;i<=N*N;i++)
 ...
```

Cela permet de changer l'espace d'itérations (afin d'effectuer éventuellement d'autres transformations).

Echange de boucles

```
for (i=2;i<=N;i++)
 for (j=2;j<=M;j++)
  A[i,j]=A[i,j-1]+1;
```

devient,

```
for (j=1;j<=M;j++)
 A[1:N,J]=A[1:N,j-1]+1;
```

Division de noeud

```
for (i=1;i<=N;i++)
 { A[i]=B[i]+C[i];
   D[i]=A[i-1]*A[i+1]; }
```

devient,

```
for (i=1;i<=N;i++)
 { A[i]=B[i]+C[i];
   temp[i]=A[i+1];
   D[i]=A[i-1]*temp[i]; }
```

Le cycle interne de dépendance est ainsi remplacé par une anti-dépendance.

Intérêt

Donc après distribution de boucle, on aura:

```
forall (i=1;i<=N;i++)
 A[i]=B[i]+C[i];
forall (i=1;i<=N;i++)
 temp[i]=A[i+1];
forall (i=1;i<=N;i++)
 D[i]=A[i-1]*temp[i];
```


Réduction de boucle

Dans le cas où le cycle interne de dépendance est de type dépendance de flot on ne peut pas utiliser la méthode précédente mais, par exemple:

```
for (i=3;i<=N;i++)
 { A[i]=B[i-2]-1;
   B[i]=A[i-3]*k; }
```

devient,

```
for (j=3;j<=N;j=j+2)
 forall (i=j; i<=j+1; i++)
  { A[i]=B[i-2]-1;
    B[i]=A[i-3]*k; }
```


Déroulement de boucle

```
for (i=1;i<=100;i++)
 A[i]=B[i+2]*C[i-1];
```

devient,

```
for (i=1;i<=99;i=i+2)
 {
  A[i]=B[i+2]*C[i-1];
  A[i+1]=B[i+3]*C[i];
 }
```

Rotation de boucle

```
for (i=2;i<=N-1;i++)
 for (j=2;j<=N-1;j++)
  A[i,j]=(A[i+1,j]+A[i-1,j]+A[i,j+1]+A[i,j-1])/4;
```

Il y a un front d'onde, cela devient,

```
forall (i=2;i<=N-1;i++)
 for (j=i+2;j<=i+N-1;j++)
  A[i,j-1]=(A[i+1,j-i]+A[i-1,j-i]+A[i,j+1-i]+A[i,j-1-i])/4;
```


Exemple de parallélisation de code

Phase de *remontée* après une décomposition LU:
Soit donc à résoudre le système triangulaire supérieur

$$Ux = b$$

avec,

$$U = \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} & \cdots & U_{1,n} \\ 0 & U_{2,2} & U_{2,3} & \cdots & U_{2,n} \\ & & & \cdots & \\ 0 & 0 & \cdots & 0 & U_{n,n} \end{pmatrix}$$

et $U_{i,i} \neq 0$ pour tout $1 \leq i \leq n$.

Remontée...

On procède par "remontée" c'est à dire que l'on calcule successivement,

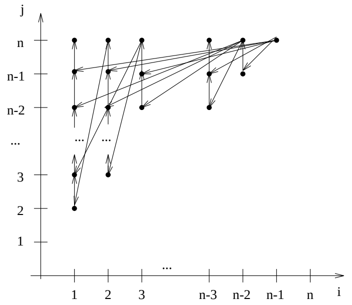
$$\begin{aligned} x_n &= \frac{b_n}{U_{n,n}} \\ x_i &= \frac{b_i - \Sigma_{j=i+1}^{n} U_{i,j} x_j}{U_{i,i}} \end{aligned}$$

pour $i = n-1, n-2, \cdots, 1$.

Code séquentiel

```
x[n]=b[n]/U[n,n];
for (i=n-1;i>=1;i--)
{
x[i]=0;
for (j=i+1;j<=n;j++)
     L: x[i]=x[i]+U[i,j]*x[j];
     x[i]=(b[i]-x[i])/U[i,i];
     }
```

Graphe de dépendances



Il y a aussi des anti-dépendances des itérations (i, j) vers $(i-1, j)$...

Parallélisation

En faisant une rotation et une distribution de boucles:

```
H': forall (i=1;i<=n-1;i++)
       x[i]=b[i];
T': x[n]=b[n]/U[n][n];
H:  for (t=1;t<=n-1;t++)
     forall (i=1;i<=n-t;i++)
      L: x[i]=x[i]-x[n-t+1]*U[i][n-t+1];
     T: x[n-t]=x[n-t]/U[n-t][n-t];
```

Le ratio d'accélération est d'ordre $\frac{n}{4}$ asymptotiquement.

Allen et Kennedy: principe

- Remplacer certaines boucles `for` par des boucles `forall`
- Utiliser la distribution de boucles pour diminuer le nombre d'instructi dans les boucles, et donc réduire les dépendances

Algorithme

- Commencer avec $k = 1$
- Supprimer dans le GDRN G toutes les arêtes de niveau inférieur à k
- Calculer les composantes fortement connexes (CFC) de G
- Pour tout CFC C dans l'ordre topologique:
  - Si C est réduit à une seule instruction S sans arête, alors générer des boucles parallèles dans toutes les dimensions restantes (i.e. niveaux k à n_S) et générer le code pour S
  - Sinon, $l = l_{min}(C)$, et générer des boucles parallèles du niveau k au niveau $l-1$, et une boucle séquentielle pour le niveau l. Puis reboucler l'algorithme avec C et $k = l+1$

Exemple

```
for (i=1;i<=N;i++)
  for (j=1;j<=N;i++) {
    S1: a(i+1,j+1) = a(i+1,j)+b(i,j+2);
    S2: b(i+1,j) = a(i+1,j-1)+b(i,j-1);
    S3: a(i,j+2) = b(i+1,j+1)-1;
  }
```

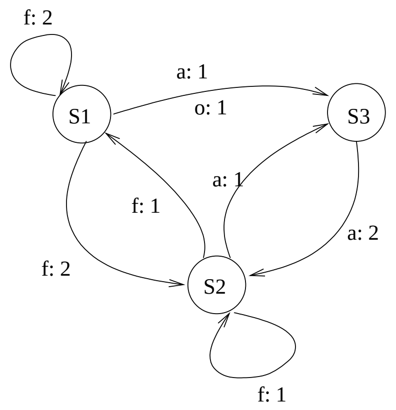

Exemple

- Flot $S1 \to S1$, variable a, distance $(0,1)$,
- Flot $S1 \to S2$, variable a, distance $(0,2)$,
- Flot $S2 \to S1$, variable b, distance $(1,-2)$,
- Flot $S2 \to S2$, variable b, distance $(1,1)$,

Exemple

- Anti $S1 \to S3$, variable a, distance $(1,-2)$,
- Anti $S2 \to S3$, variable a, distance $(1,-3)$,
- Anti $S3 \to S2$, variable b, distance $(0,1)$,
- Sortie $S1 \to S3$, variable a, distance $(1,-1)$.

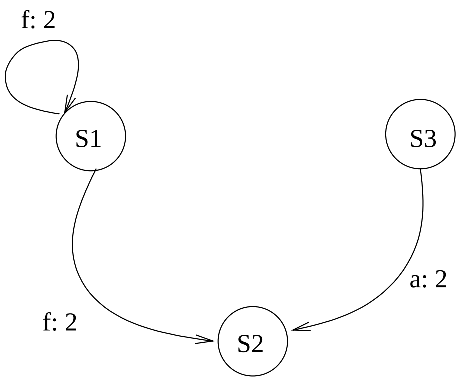
Exemple



Algorithme

- Le GDRN est fortement connexe et a des dépendances de niveau 1
- La boucle sur i sera donc séquentielle
- On enlève maintenant les dépendances de niveau 1...

GDRN modifié



Parallélisation finale

```
for (i=1;i<=N;i++) {
  for (j=1;j<=N;j++)
    S1: a(i+1,j+1) = a(i+1,j)+b(i,j+2);
  forall (j=1;j<=N;j++)
    S3: a(i,j+2) = b(i+1,j+1)-1;
  forall (j=1;j<=N;j++)
    S2: b(i+1,j) = a(i+1,j-1)+b(i,j-1);
}
```


Quelques perspectives

- Modèles sémantiques du parallélisme
- Autres langages ou paradigmes:
  - Linda, Facile, CHOCS, Erlang, CML, OCCAM, PVM/MPI etc.
  - algèbres de processus: CCS, CSP, π-calcul, Join-calcul etc.
- Analyse statique par interprétation abstraite
- Algorithmes distribués... (voir N. Lynch "Distributed Algorithms")
- Systèmes distribués tolérants aux pannes - Systèmes auto-stabilisants
- Projets GRID

Modèles sémantiques

[DEA Programmation]

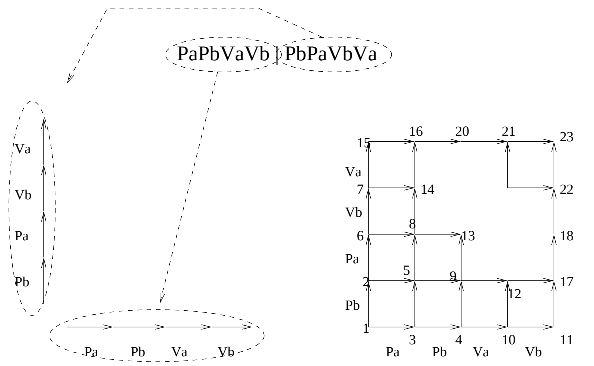
- Systèmes de transitions
- Structures d'événements
- Réseaux de Petri
- etc.

Motivations

- Associer à un programme (ici parallèle) un modèle mathématique
- Manipulable, et dont on puisse tirer (par algorithmes) des propriétés d'intérêt
- Exemple: points morts, états atteignables, chemins d'exécutions, valeurs que peuvent prendre les variables etc.

Exemple: systèmes de transitions

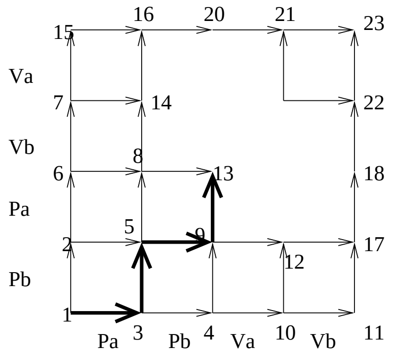
Processus: systèmes dynamiques (discrets en général) = graphes dirigés d'actions:



Exemple d'exécution possible

Programme $(T_1 = PaPbVaVb) \mid (T_2 = PbPaVbVa)$:

T_1	T_2
Pa	$-$
$-$	Pb
$Pb(?)$	$-$
$-$	$Pa(?)$

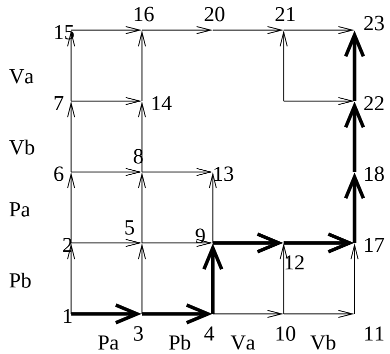


$\boxed{\text{Point mort!}}$

Exemple

1 autre exécution possible, complète (au total 16!):

T_1	T_2
Pa	$-$
Pb	$-$
$-$	$Pb(?)$
Va	$-$
Vb	$-$
$-$	Pa
$-$	Va
$-$	Vb



Chemins

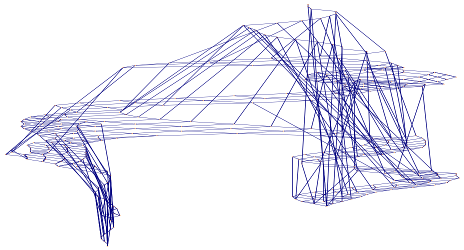
$\boxed{\text{... il y a très peu de chemins “intéressants”}}$

En fait $T_1 = PaPb(b = b + 1)VaVb$ et $T_2 = PbPa(b = b * 2)VbVa$ et au début, $b = 2$:

- Les 5 chemins les plus en haut à gauche sont “équivalents” à T_2 puis T_1 : ils calculent $\underline{b = 5}$ (2*2+1)
- Les 5 chemins les plus en bas à droite sont “équivalents” à T_1 puis T_2: ils calculent $\underline{b = 6}$ ((2+1)*2)
- Les 6 chemins près de la diagonale sont équivalents: ils ne terminent pas

En général, pénible...

Grand nombre d'états: ici algorithme de Dekker, quelques lignes de C pour 2 processus, centaines de chemins/^{t}2 “vrais” chemins!



D'où nécessité d”'abstraire”, ou d'utiliser d'autres modèles...

Autres langages

[DEA Programmation]

- PVM/MPI: voir par exemple: http://www-unix.mcs.anl.gov/mpi/
- Algèbres de processus: exemple CCS
- Join calcul: voir par exemple: http://join.inria.fr/
- Paradigme Linda: “tuple space” et récupération/accès par pattern matching
- cf. également www.cs.yale.edu/Linda/linda.html

CCS

Abstraction de la partie “coordination” dans un système à passage de messages (CML, Erlang, Facile etc.) :

- plus de calcul, fonctions etc. mais flot de contrôle (récursion, suite d'instructions)
- **choose** $\longrightarrow +$ (choix non-deterministe),
- **send**$(a,x) \longrightarrow \overline{a}(x)$ (envoi),
- **val** x=**accept**$(a) \longrightarrow a(x)$

CCS PUR

Maintenant, on oublie les valeurs passées...

$$
\begin{array}{rll}
t = & nil & \text{fin de processus} \\
| & \overline{a} \in \Sigma & \text{envoi sur } a \\
| & a \in \Sigma & \text{réception sur } a \\
| & t + t & \text{choix non-déterministe} \\
| & t.t & \text{composition séquentielle} \\
| & t \mid t & \text{composition parallèle} \\
| & t \backslash c & \text{cache le canal } c \\
| & t[f] & \text{renommage par fonction f} \\
| & \text{rec } x.t[x] & \text{agent récursif}
\end{array}
$$

EXEMPLES

- Composition parallèle et *hiding* :

  let *c*=**channel***()* *in* **spawn***(fn () ⇒* **accept***(c));* **send***(c,0)* **end**

  devient $(c.nil \mid \overline{c})\backslash c$
- Choix :

  select *[***transmit***(a,1);* **wrap***(***accept** *b, fn x ⇒* **send***(b,x))]*

  devient $\overline{a} + (b.\overline{b})$
- Récursion :

  let fun *entiers n =* **send***(c,n); entiers(n+1) in entiers(0)* **end**

  devient rec $X.\overline{c}.X$

SÉMANTIQUE OPÉRATIONNELLE

Idée : Graphe des états de la machine et de leurs dépendances temporelles

Exemple :

$$
\begin{tikzcd}
 & (x=1) \arrow[r, "x := x+1"] & (x=2) \\
(x=0) \arrow[ur, "x := 1"] \arrow[r, "x := x+3"'] & (x=3) &
\end{tikzcd}
$$

Sémantique de CCS

<u>Principe :</u> Les états sont les programmes qui restent à évaluer.
<u>Exemple :</u>

$$\begin{tikzcd}
 & b.nil \arrow[r, "b"] & nil \\
a.b.nil + c.nil \arrow[ur, "a"] \arrow[r, "c"'] & nil &
\end{tikzcd}$$

Notion d'observation

<u>Exemple:</u>

$$\begin{tikzcd}
 & nil \\
a.nil \arrow[ur, "a"] \arrow[dr, "a"'] & \\
 & nil
\end{tikzcd}
\quad \text{et} \quad
\begin{tikzcd}
a.nil \arrow[r, "a"] & nil
\end{tikzcd}$$

sont essentiellement les même programmes.

Mais...

$$\begin{tikzcd}
 & b.nil \arrow[r, "b"] & nil \\
a.b.nil + a.c.nil \arrow[ur, "a"] \arrow[dr, "a"'] & & \\
 & c.nil \arrow[r, "c"'] & nil
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & nil \\
a.(b.nil + c.nil) \arrow[r, "a"] & b.nil + c.nil \arrow[ur, "b"] \arrow[dr, "c"'] & \\
 & & nil
\end{tikzcd}$$

sont des programmes différents parce que les choix entre b et c sont faits à des temps différents (supposer un deadlock après un des a).

Exemple

On considère les agents A et B définis par:

$$A = a.A'$$
$$A' = \overline{c}.A$$

et

$$B = c.B'$$
$$B' = \overline{b}.B$$

Peut-on trouver un équivalent pour $(A \mid B)\backslash c$?

Exemple

<u>La réponse est oui</u>

Soient,

$$C_0 = \overline{b}.C_1 + a.C_2$$
$$C_1 = a.C_3$$
$$C_2 = \overline{b}.C_3$$
$$C_3 = \tau.C_0$$

Alors, $(A \mid B)\backslash c \sim C_1$.

Equations algébriques

- $P + Q \sim Q + P$
- $P + (Q + R) \sim (P + Q) + R$
- $P + P \sim P$
- $P + nil \sim P$
- $P \mid Q \sim Q \mid P$
- $P \mid (Q \mid R) \sim (P \mid Q) \mid R$
- $P \mid nil \sim P$
- $P\backslash L \sim P$ si $\mathcal{L}(P) \cap (L \cup \overline{L}) =$
- $P\backslash K\backslash L \sim P\backslash(K \cup L)$
- $P[f]\backslash L \sim P\backslash f^{-1}(L)[f]$
- $(P \mid Q)\backslash L \sim (P\backslash L) \mid (Q\backslash L)$
- $P[Id] \sim P$
- $P[f] \sim P[f']$ si $f_{\mathcal{L}(P)} = f'_{\mathcal{L}(P)}$
- $P[f][f'] \sim P[f' \circ f]$
- $(P \mid Q)[f] \sim P[f] \mid Q[f]$ si $f_{L \cup \overline{L}}$ est bijective où $L = \mathcal{L}(P) \cup \mathcal{L}(Q)$

Remarquer que $(a.nil) \mid (b.nil) \sim a.b.nil + b.a.nil$ (loi d'expansion en général).

Suite

(Propriétés de congruence) Si $P_1 \sim P_2$,

- $\alpha.P_1 \sim \alpha.P_2$
- $P_1 + Q \sim P_2 + Q$
- $P_1 \mid Q \sim P_2 \mid Q$
- $P_1\backslash L \sim P_2\backslash L$
- $P_1[f] \sim P_2[f]$

Spécification : bit alterné

<u>Protocole</u> : P envoie alternativement un bit à Q, qui le retourne. P renvoie le message s'il a été mal reçu.

$$P = \overline{bit_0}.P'$$
$$P' = ack'_1.\overline{bit_0}.P' + ack'_0.\overline{bit_1}.P'$$

$$Q = bit'_0.\overline{ack_0}.Q + bit'_1.\overline{ack_1}.Q$$

Spécification, suite

Ligne de communication parfaite $L = bit_0.\overline{bit'_0}.L + bit_1.\overline{bit'_1}.L$

Ligne de communication imparfaite $M = (bit_0 + bit_1).(\overline{bit'_0} + \overline{bit'_1} + \tau).M$

Ligne de retour $R = ack_0.\overline{ack'_0}.R + ack_1.\overline{ack'_1}.R$

Vérification

$P \mid M \mid R \mid Q$ peut se bloquer en $P' \mid M \mid R \mid Q$ après une action τ de M.

Par contre, $(P \mid L \mid R \mid Q) \sim$ rec $X.\tau.X$

En effet,

Vérification

$$\begin{aligned} P \mid L \mid R \mid Q & \stackrel{\tau^2}{\rightarrow} P' \mid L \mid R \mid \overline{ack_0}.Q \\ & \stackrel{\tau^2}{\rightarrow} \overline{bit_1}.P' \mid L \mid R \mid Q \\ & \stackrel{\tau^2}{\rightarrow} P' \mid L \mid R \mid \overline{ack_1}.Q \\ & \stackrel{\tau^2}{\rightarrow} P \mid L \mid R \mid Q \end{aligned}$$

Analyse statique

[DEA Programmation]

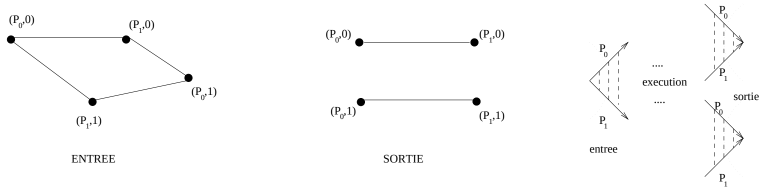
- Vous avez vu un premier exemple avec la logique de Hoare en IF
- Voir maintenant http://www.di.ens.fr/˜cousot
- Par exemple, à partir d'une sémantique par systèmes de transitions, "abstraire" les valeurs que peuvent prendre les variables par des intervalles d'entiers (abstraction ici de la logique de Hoare)
- Sujet très actif aussi bien dans la recherche qu'industriellement (cf. Polyspace Technologies par exemple)

Systèmes distribués tolérants aux pannes

Existe-t-il un protocole qui gère N processus asynchrones tels que,
- ils démarrent avec certaines valeurs d'entrée spécifiées, calculent et communiquent, et terminent avec certaines valeurs de sortie spécifiées,
- ils fonctionnent selon un modèle de calcul spécifié (mémoire partagée, passage de messages etc.)
et qui soit, par exemple sans-attente (résistant aux pannes), c'est à dire qui termine même si $N-1$ processeurs tombent en panne?
Exemple: Le consensus: les valeurs d'entrée sont une valeur quelconque dans un ensemble S, les valeurs de sortie doivent être toutes les mêmes et égales à une des valeurs d'entrée.

Résultat

Il n'existe pas de protocole pour le consensus sur une machine à mémoire partagée sans attente [Herlihy/Shavit+FLP etc.].



Systèmes auto-stabilisants

- Cas particulier de protocoles "tolérants aux pannes"
- Communauté importante, longtemps après le papier important (1974) sur le sujet, de E. W. Dijkstra
- Chaque processus doit arriver en un temps fini à "réparer" une mauvaise valeur fournie à un moment donné par un processus
- Voir par exemple: http://www.cs.uiowa.edu/ftp/selfstab/bibliograph

Projets GRID

- Un sujet à la mode!
- Systèmes distribués "coopératifs" à l'échelle de la planète... (cluster de stations sur internet par exemple)
- Au menu: algorithmique hétérogène, tolérance aux pannes etc.
- Voir par exemple http://www.gridcomputing.com/