Polyhedral Liveness Analysis

Alexandre Isoard

with Alain Darte
Compsys Team, LIP

21 septembre 2015

Durée de vie

Utilisation de l'analyse des durées de vie :

- Nécessaire à l'allocation mémoire :
  - Allocation de registres
  - Contraction de tableaux
- Fournit des informations cruciales :
  - Live-in/live-out (inlining, offloading)
  - Empreinte mémoire (prédiction des effets de cache)

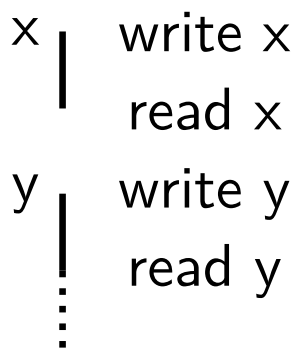
Deux types d'analyses :

- Basée sur les valeurs (value based)
- Basée sur la mémoire (memory based)

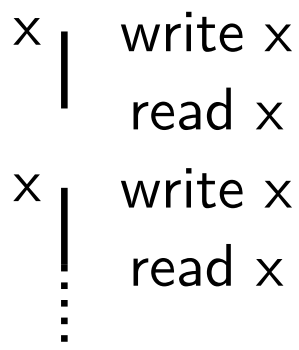
Allocation de registres (cas simple)

Durée de vie, conflit/interférence, réutilisation

```
x = ...;
y = x + ...;
... = y;
```



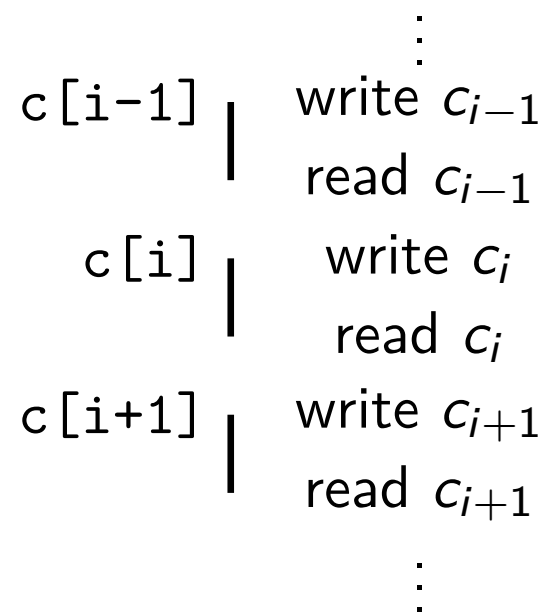

```
x = ...;
x = x + ...;
... = x;
```



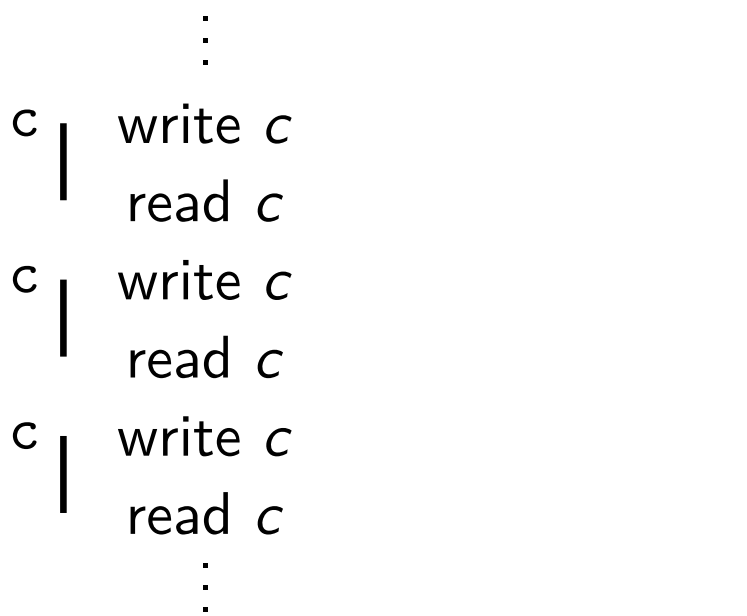
Repliement de tableau (cas simple)

Affine (polyédrique) : déroulage et analyse symbolique

```
c[0] = 0;
for(int i = 0; i < n; ++i)
   c[i+1] = c[i] + ...;
return c[n];
```

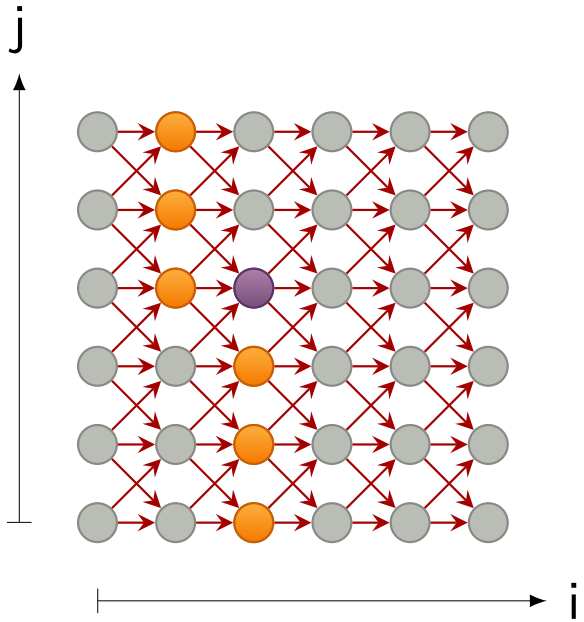



```
c = 0;
for(int i = 0; i < n; ++i)
   c = c + ...;
return c;
```



Exemple stencil

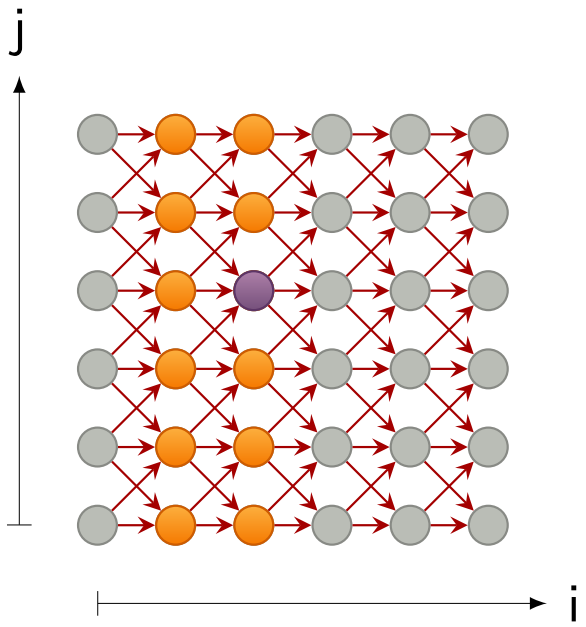
```
for(int i = 0; i < T-1; ++i)
    for(int j = 1; j < N; ++j)
        a[i+1][j] = a[i][j-1] + a[i][j] + a[i][j+1];
```



$$\texttt{a[i][j]} \mapsto \texttt{a[(j-i)\%N]}$$

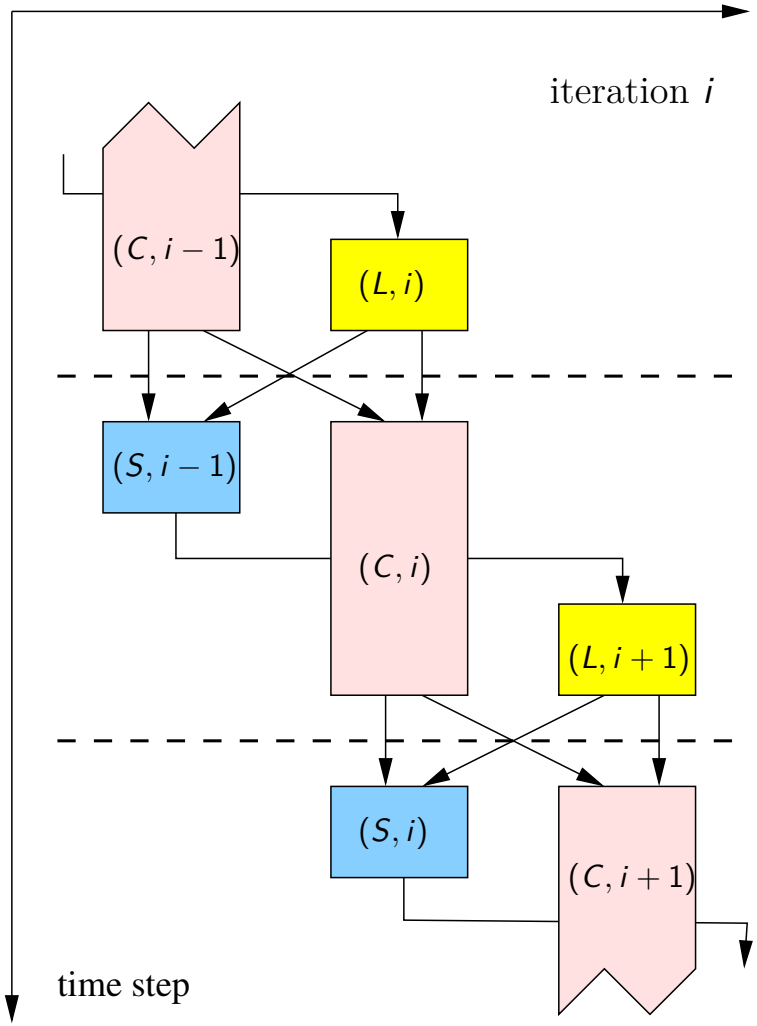
Exemple stencil (parallélisme interne)

```
for(int i = 0; i < T-1; ++i)
    #pragma omp parallel for
    for(int j = 1; j < N; ++j)
        a[i+1][j] = a[i][j-1] + a[i][j] + a[i][j+1];
```



$$\texttt{a[i][j]} \mapsto \texttt{a[i\%2][j]}$$

Double buffering (déjà un peu plus compliqué)



$$(S, i-1) \bowtie (C, i)$$

$$(C, i) \bowtie (L, i+1)$$

mais

$$(S, i-1) \not\bowtie (L, i+1)$$

Méthode standard : papillon avec ordonnancement θ

$\theta(S[\vec{i}]) =$ temps auquel on exécute l'instruction $S[\vec{i}]$

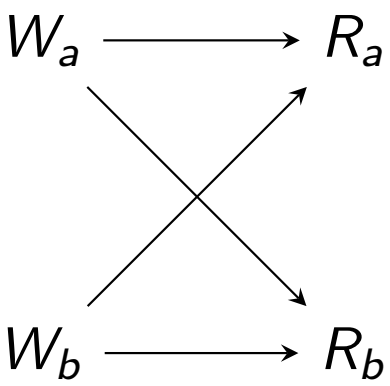
$W_a \longrightarrow R_a$

2 write, 2 read $\implies \mathcal{O}(\#\text{access}^4)$

$W_b \longrightarrow R_b$

Méthode standard : papillon avec ordonnancement θ

$\theta(S[\vec{i}]) =$ temps auquel on exécute l'instruction $S[\vec{i}]$

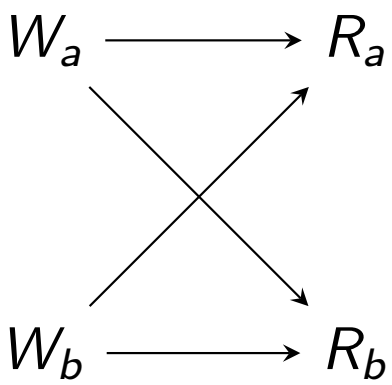


2 write, 2 read $\implies \mathcal{O}(\#\text{access}^4)$

Méthode standard : papillon avec ordonnancement θ

$\theta(S[\vec{i}])$ = temps auquel on exécute l'instruction $S[\vec{i}]$

$\theta(S[\vec{i}]) = \theta(S[\vec{j}]) \implies S[\vec{i}]$ en parallèle de $S[\vec{j}]$



2 write, 2 read $\implies \mathcal{O}(\#\text{access}^4)$

Gestion du parallélisme = cas d'égalité

- instance différente $\implies$ conflit
- même instance $\implies$ pas conflit

☛ s'étend au parallélisme série parallèle

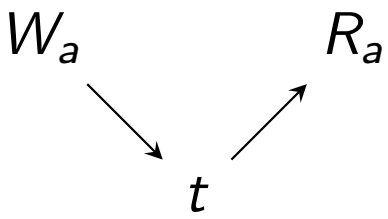
Objectifs

Analyse de durée de vie / conflits mémoire :

- Algorithmiquement plus performante (cf allocation de registres)
- Supportant une spécification parallèle générale (graphe de tâches)
- Profitant des propriétés du programme (évidence syntaxique)
- Supportant l'approximation du contrôle (`if` non déterminés)
- Des propriétés sur le graphe d'interférence ? Parfait ?

live x live (séquentiel)

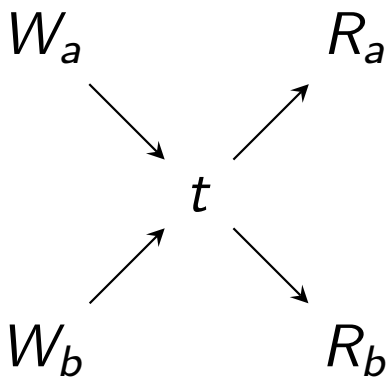
$live(t)$ = pour chaque temps, la liste des variables en vie



1 write, 1 read, 1 time $\Longrightarrow$ $\mathcal{O}(\#\text{prgm} \times \#\text{access}^2)$

live x live (séquentiel)

$live(t)$ = pour chaque temps, la liste des variables en vie



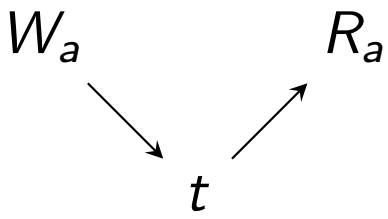
1 write, 1 read, 1 time $\Longrightarrow$ $\mathcal{O}(\#\text{prgm} \times \#\text{access}^2)$

2 mem, 1 time $\Longrightarrow$ $\mathcal{O}(\#\text{prgm} \times \text{mem}^2)$

Séquentiel $\Longrightarrow$ graphe d'interférence = graphe d'intervalles

live x write

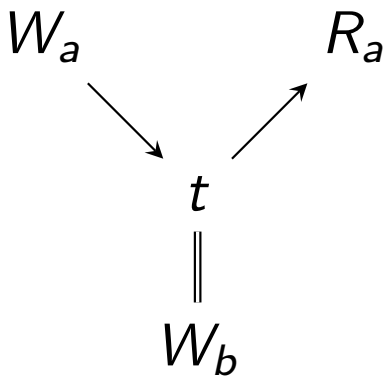
Inspirée de l'allocation de registres



2 write, 1 read $\implies \mathcal{O}(\#\text{access}^3)$

live x write

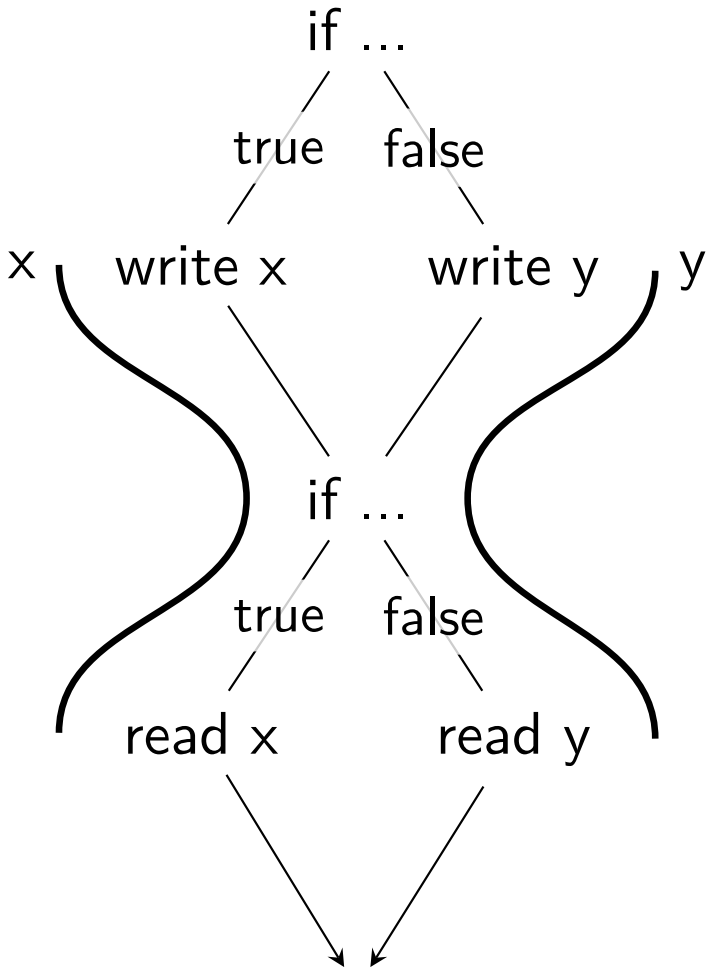
Inspirée de l'allocation de registres



2 write, 1 read $\implies \mathcal{O}(\#\text{access}^3)$

- En général plus précis
- Équivalent au précédent en SSA avec dominance

live x write : double diamant



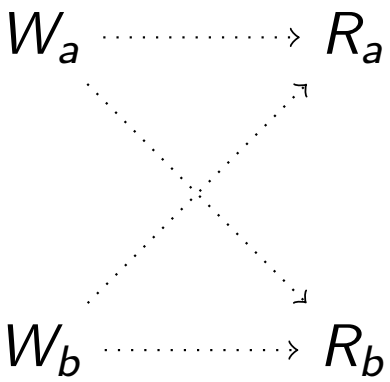
live x live = conflit

live x write = pas de conflit

☛ Correct : pas de chemin valide où les deux sont en vie en même temps

Papillon parallèle

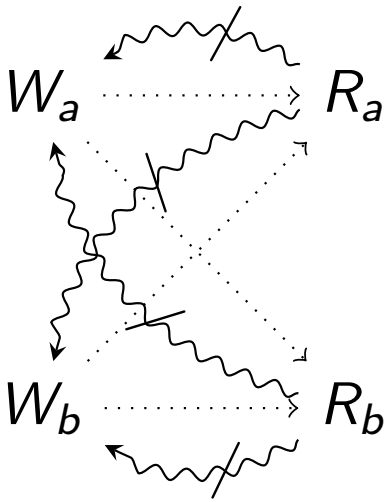
Relation happens-before : $S[i] \prec_{hb} S[j] = \forall$ trace, $S[i]$ avant $S[j]$.



2 write, 2 read $\implies \mathcal{O}(\#\text{access}^4)$

Papillon parallèle

Relation happens-before : $S[i] \prec_{hb} S[j] = \forall$ trace, $S[i]$ avant $S[j]$.



2 write, 2 read $\implies \mathcal{O}(\#\text{access}^4)$

Règle de base : si happens-before ne l'interdit pas, ça peut arriver.

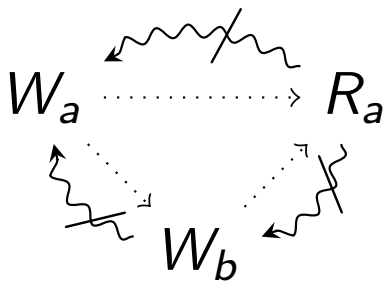
live x live (parallèle)

$[maylive(i)]^2$ ne fonctionne pas.
Par exemple sur le double buffering (conflit non transitif).

- Pas de notion de temps !
- Plus un graphe d'intervalles !
- Même plus un graphe chordal !

maylive x write

Relation happens-before : $S[i] \prec_{hb} S[j] = \forall$ trace, $S[i]$ avant $S[j]$.



2 write, 1 read $\implies \mathcal{O}(\#\text{access}^3)$

Propriétés

- $a \bowtie b$ ssi il existe une trace où a et b sont simultanément en vie.
- Une allocation respecte $\bowtie$ ssi elle est valide quelle que soit l'exécution.

Note : différent de trouver le nombre minimum de registres nécessaire quelle que soit l'exécution (NP-complet).

Propriétés du graphe d'interférence

Si basic block :
graphe d'intervalles
☛ Coloration complexité linéaire

Parallélisme ordre partiel :
complémentaire d'un graphe de comparabilité (graphe de réutilisation)
☛ Coloration polynomiale (graphe parfait)

Parallélisme avec atomic :
plus compliqué (quelconque ?)
☛ Probablement NP-complet à colorier

Task graph vs control flow graph

Definition (Task graph)

Tous les nœuds sont exécutés ; tout chemin est exécuté dans l'ordre ; les nœuds en parallèle créent en général des conflits

Definition (Control flow graph)

Un seul chemin voit ses nœuds exécutés ; le chemin est exécuté dans l'ordre ; les nœuds en parallèle ne créent en général pas de conflits

- Première étape pour réconcilier les deux.
- Lien avec la thèse d'Albert Cohen (expansion).

Conclusion

- Calcul des durées de vie pour nimporte quel ordre partiel (tant qu'il est exprimable)
  - code séquentiel
  - ordonnancement multi-dimensionnel (parallélisme interne)
  - boucles parallèles à la OpenMP (série parallèle)
  - parallélisme à la `async` voire `clock` de X10
  - dépendances de flot. . . mais
- Cloture transitive du happens-before nécessaire
  - dépendances uniformes des UOV : Strout & al. 1998
  - ordonnancement affine des AOV : Thies, Vivien, Amarasinghe 2007
  - dépendances uniformes pour code tuilé : Yuki, Rajopadhye 2013
- Interaction avec le contrôle
  - liens avec la thèse d'Albert Cohen

☛ vers une extension des analyses polyédriques aux langages parallèles.