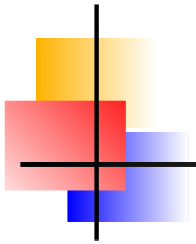


Gestion automatique de la mémoire dynamique pour des programmes Java temps-réel embarqués

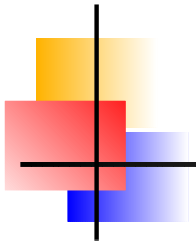
Stage de M2R Systèmes et logiciels

Guillaume Salagnac

salagnac@imag.fr

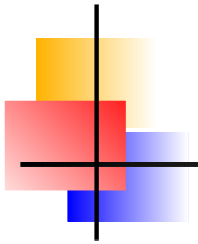
Laboratoire Verimag

Stage effectué sous la direction de Sergio Yovine



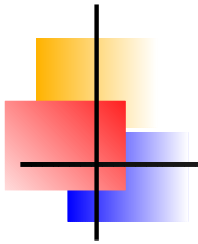
Plan

- Contexte: Java temps-réel et la mémoire
- État de l’art
- Notre approche
- Contribution
- Conclusions et perspectives



Introduction

- Java
  - Langage attractif
  - pas de gestion manuelle de la mémoire dynamique
- Programmation temps-réel et mémoire
  - Machine virtuelle
  - Performances du ramasse-miettes



Contexte

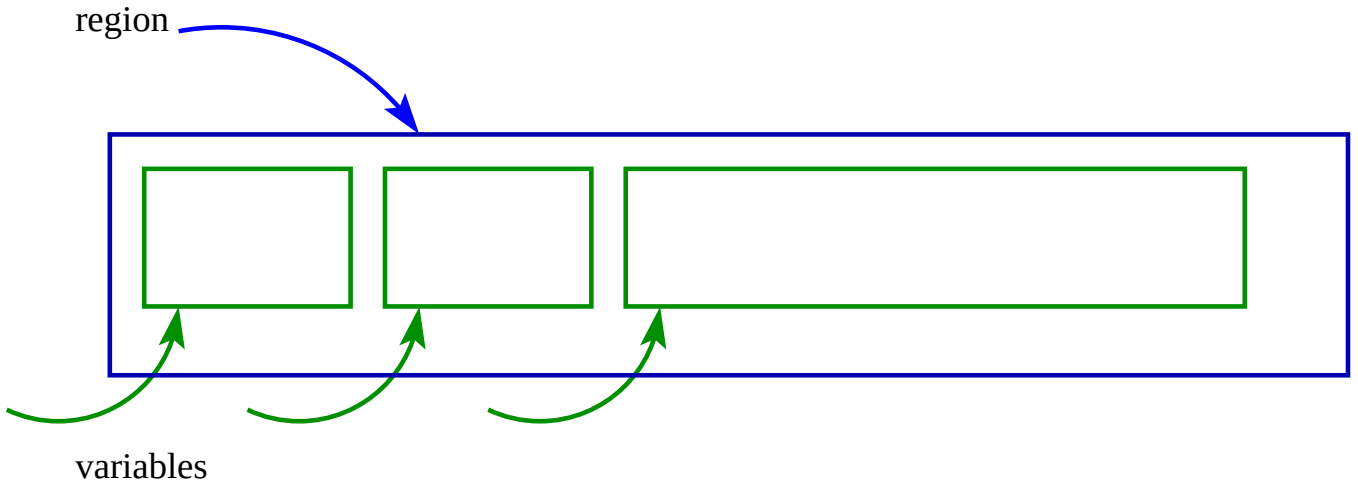
Gestion de la mémoire dynamique :

- Manuelle
  - difficile à programmer
  - source d'erreurs complexes à trouver
  - *Dangling references* $\Longrightarrow$ non-sûreté
- Automatique (GC)
  - imprédictibilité du temps d'exécution

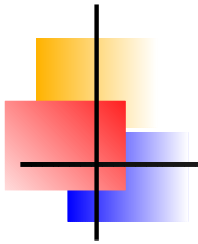
Problèmes récurrents :

- Fragmentation du tas
- Performances

Gestion de la mémoire en régions

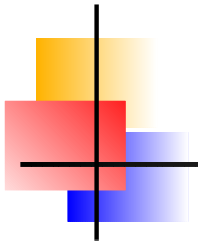


- Pas de fragmentation
- Allocation/désallocation en temps prédictible
- Toujours difficile à programmer manuellement
  - Détermination manuelle des durées de vie
  - *Dangling references*



Gestionnaires de mémoire

- Langage RC de [GA01]:
  - Extension du langage C
  - Primitives de gestion explicite
  - Tests dynamiques $\Longrightarrow$ `abort()`
- RTSJ (RealTime Specification for Java)
  - Modification de la Machine Virtuelle
  - Une région par thread
  - Gestion manuelle de la mémoire dynamique



Automatisation ?

Problème: déterminer les durées de vie des objets

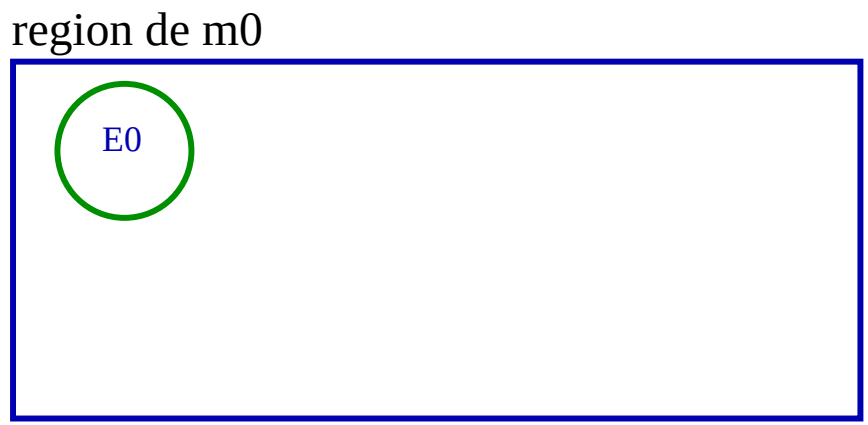
But: Transformation automatique du programme

- pas de ramasse-miettes
- gestion explicite, mais pas par le programmeur

$\Longrightarrow$hypothèse : une région par méthode

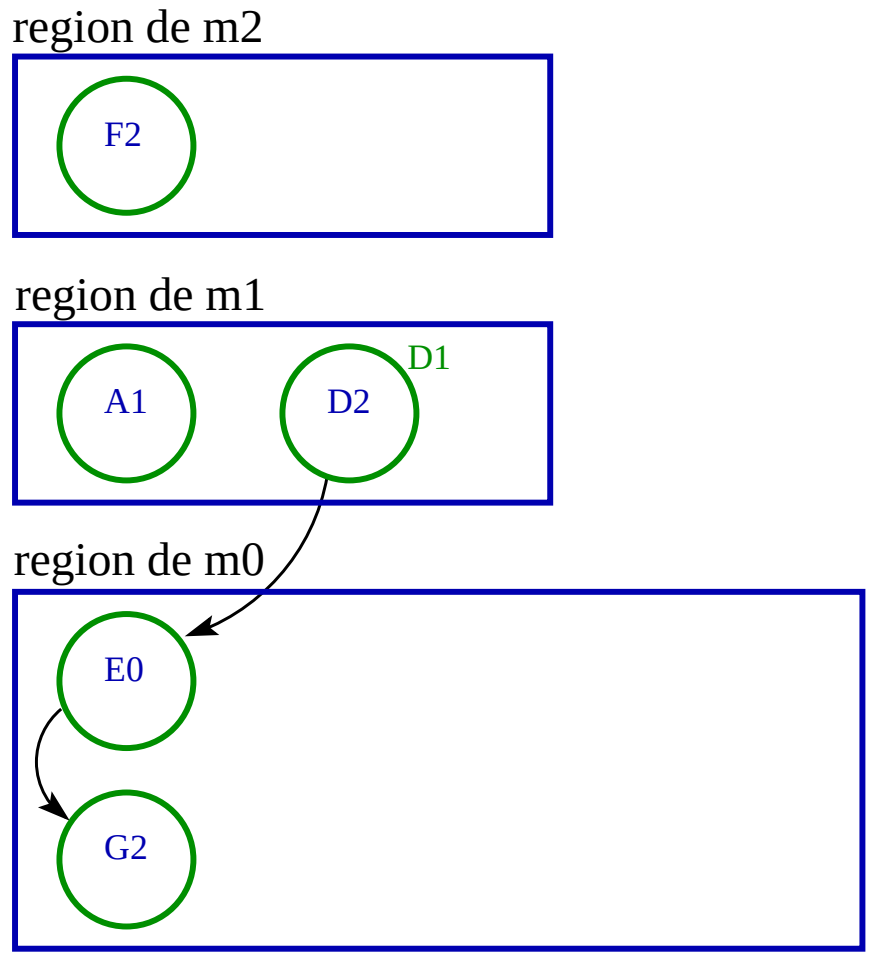
Un exemple

```
class refObject
{
    Object Ref;
}
void m0()
{
    RefObject E0 = new RefObject();
    m1(E0);
    Object D0 = m2(E0);
}
void m1(RefObject E1)
{
    RefObject A1 = new RefObject();
    Object D1 = m2(E1);
}
Object m2(RefObject E2)
{
    RefObject D2=new RefObject();
    Object F2 = new Object();
    D2.ref = E2;
    Object G2 = new Object();
    E2.ref = G2 ;
    return D2 ;
}
```



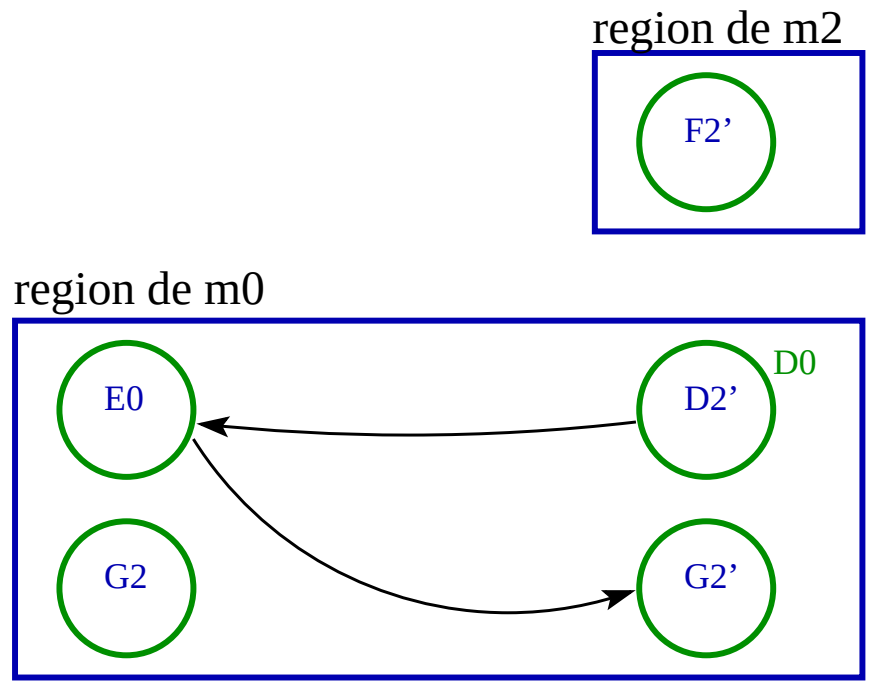
Un exemple

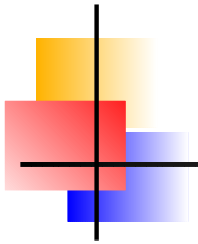
```
class refObject
{
    Object Ref;
}
void m0()
{
    RefObject E0 = new RefObject();
    m1(E0);
    Object D0 = m2(E0);
}
void m1(RefObject E1)
{
    RefObject A1 = new RefObject();
    Object D1 = m2(E1);
}
Object m2(RefObject E2)
{
    RefObject D2=new RefObject();
    Object F2 = new Object();
    D2.ref = E2;
    Object G2 = new Object();
    E2.ref = G2 ;
    return D2 ;
}
```



Un exemple

```
class refObject
{
    Object Ref;
}
void m0()
{
    RefObject E0 = new RefObject();
    m1(E0);
    Object D0 = m2(E0);
}
void m1(RefObject E1)
{
    RefObject A1 = new RefObject();
    Object D1 = m2(E1);
}
Object m2(RefObject E2)
{
    RefObject D2=new RefObject();
    Object F2 = new Object();
    D2.ref = E2;
    Object G2 = new Object();
    E2.ref = G2 ;
    return D2 ;
}
```





Approche de Deters et Cytron

- Déterminer les durées de vie des objets

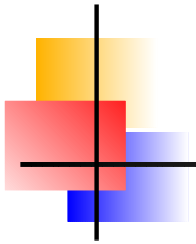
  $\Longrightarrow$ analyse dynamique (profilage)

  $\Longrightarrow$ non-sûreté

- Échappements supposés constants

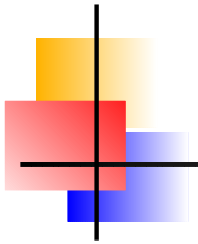
  $\Longrightarrow$ inefficacité

- Surcoûts de la RTSJ
  - création des threads



Plan: Notre approche

- Contexte
- État de l’art
- *Notre approche*
- Contribution
- Conclusions et perspectives



Notre approche

- Analyse statique du code source

  $\Longrightarrow$ déterminer les durées de vie des objets

- Sûreté assurée par construction
- Echappements dynamiques (Enregistrement)
- pas de surcoûts des threads
- Quelles performances ?

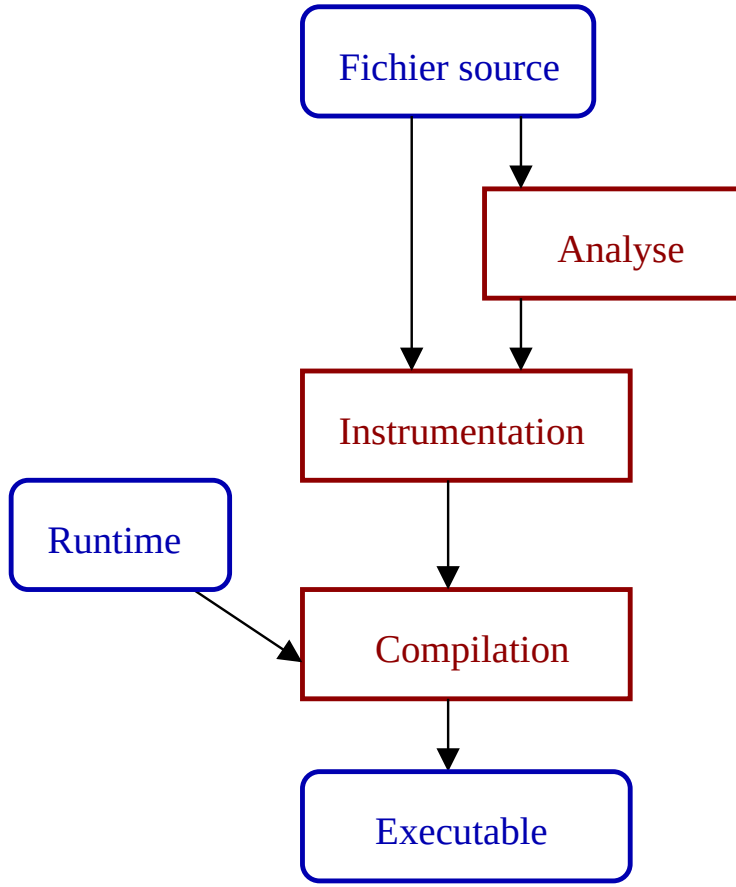
Domaine inexploré $\Longrightarrow$

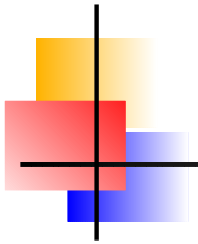
- Pas d'outils existants
- Pas de validation des hypothèses

Objectifs

Validation expérimentale de l'approche

- Forces et faiblesses
- Expérimentations et étude de cas
- $\Longrightarrow$ Implémentation d'un prototype

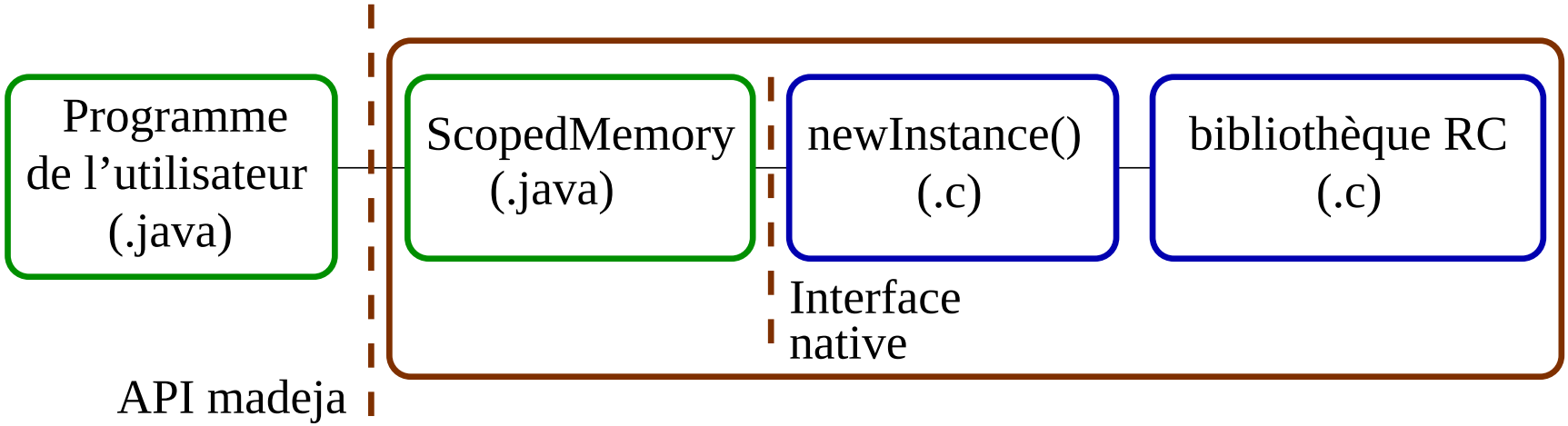




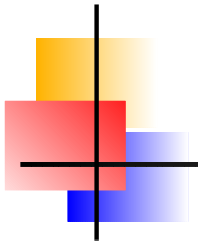
Plan: Contribution

- Contexte
- État de l’art
- Notre approche
- *Contribution*
- Conclusions et perspectives

Implémentation



- API Madeja en Java
- Prototype intégré dans l'exécutif



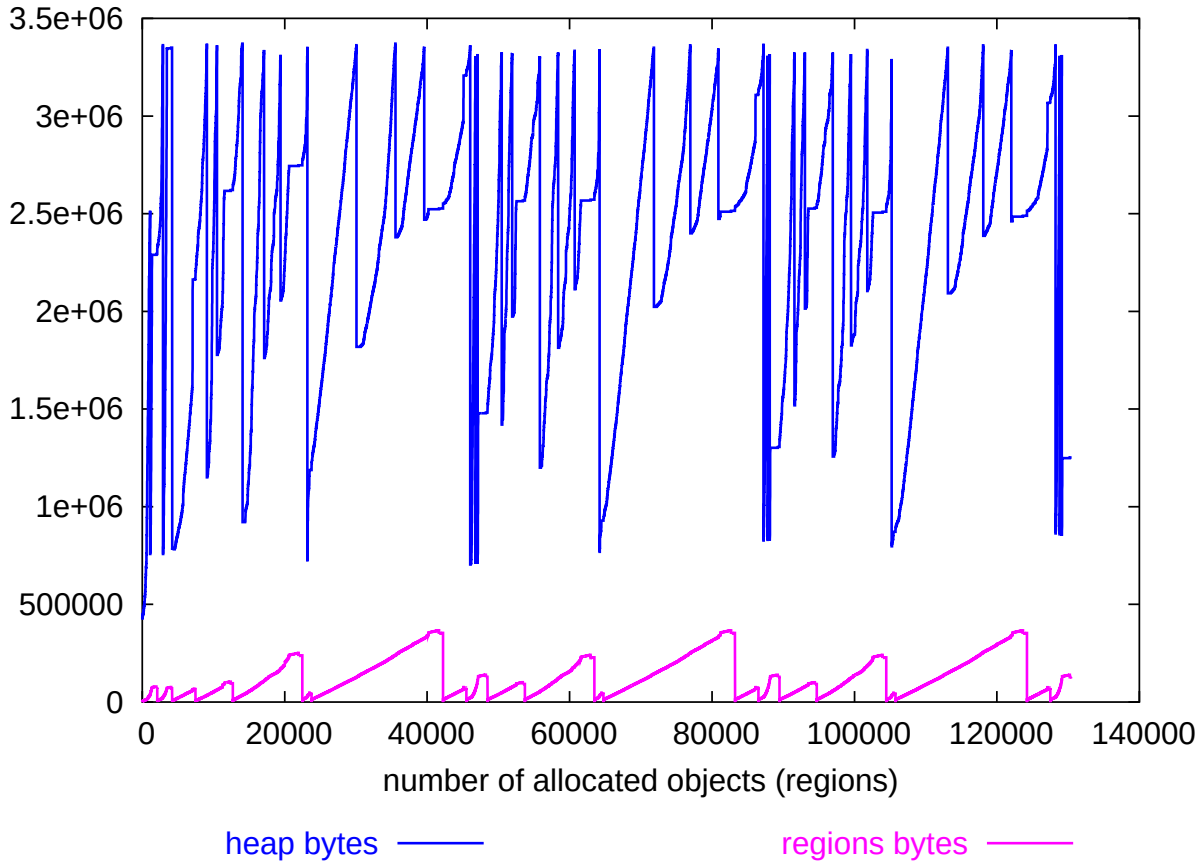
Expérimentations

Trois questions:

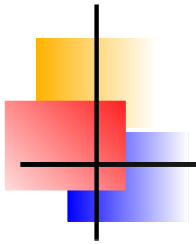
- Quel est l’impact des régions ?
- Quel bénéfice par rapport au GC ?
- Les hypothèses de l’approche sont-elles bonnes ?

1: Impact des régions

Les régions sont-elles efficaces ?



Majorité des allocations : non détectables syntaxiquement dans le programme.

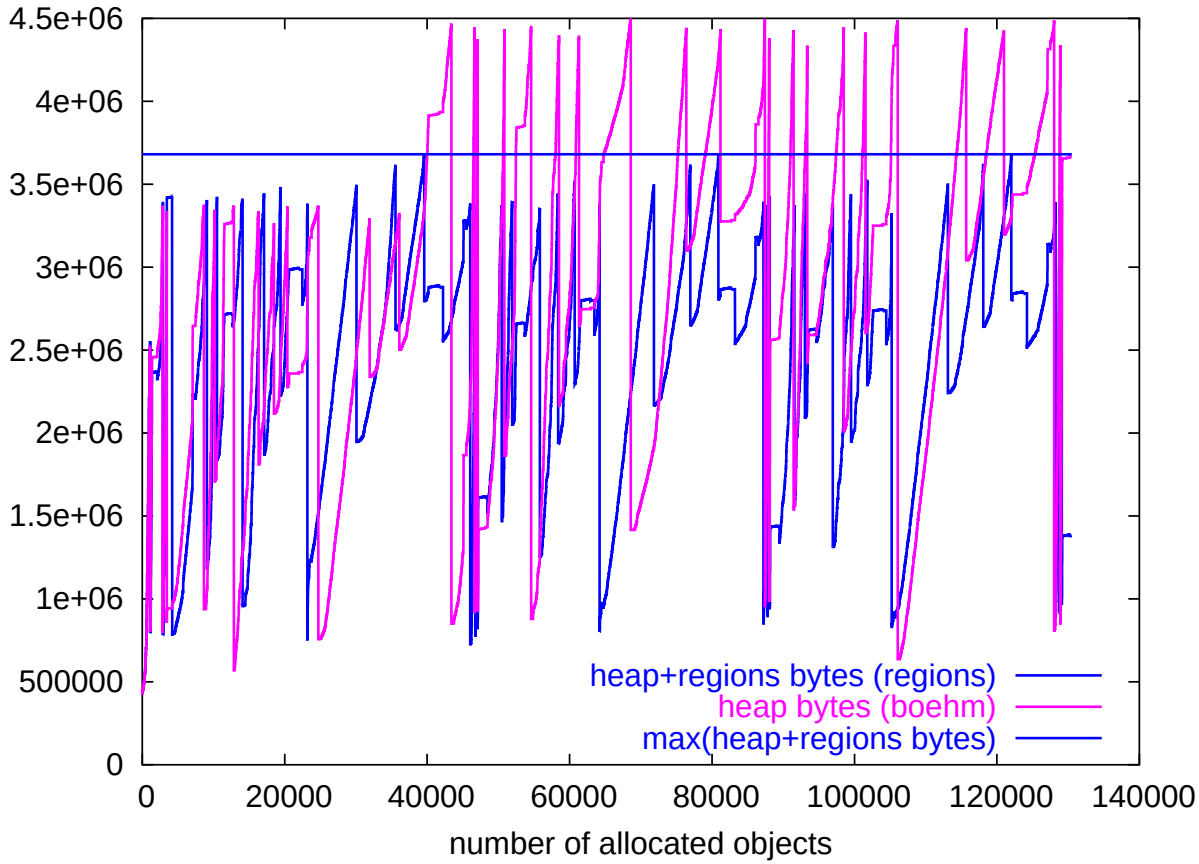


Résultats

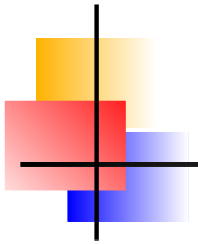
- Utilisation *régulière* de la mémoire
- Beaucoup d’allocations *automatiques*
- Analyse impraticable au niveau source

2: Progrès vis-à-vis du GC

Gagne-t-on face à un programme *sans régions* ?



Moins grande occupation en pic qu'un programme sans régions.

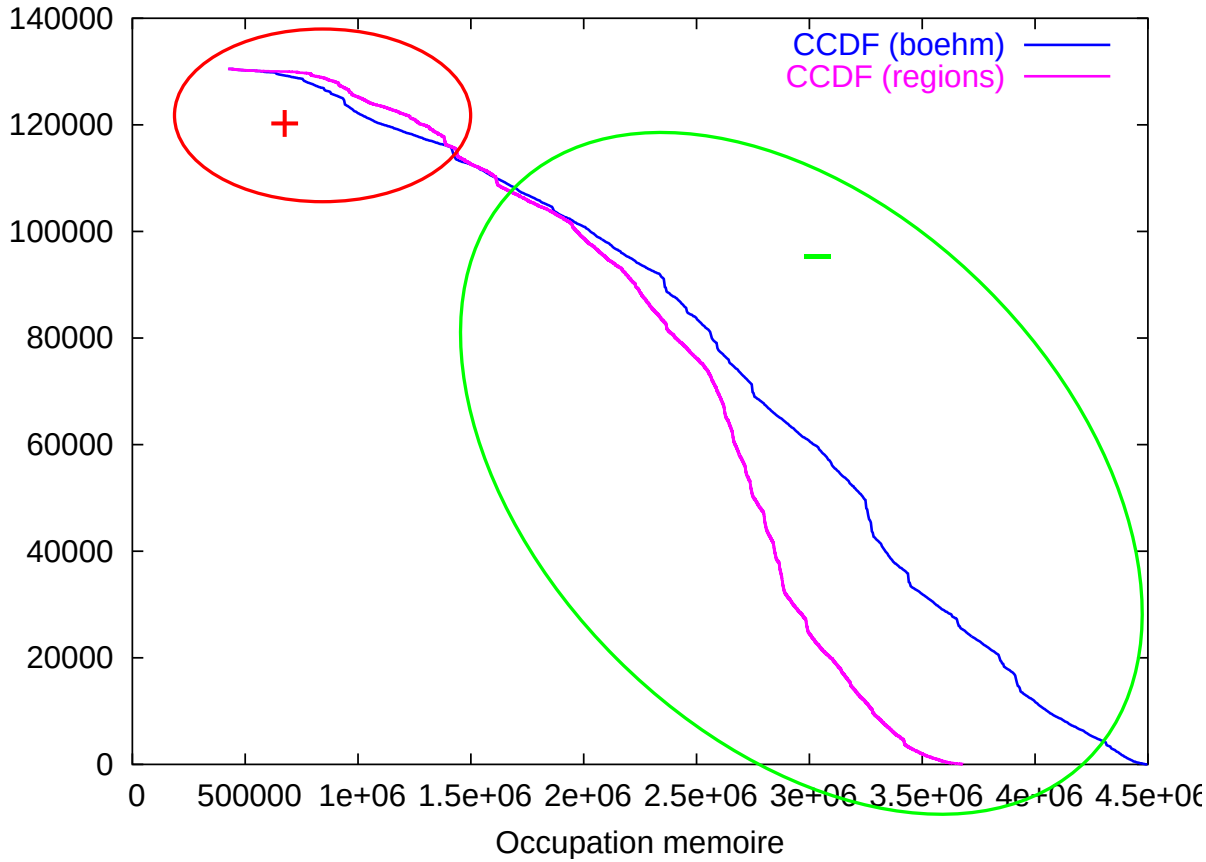


Résultats

- Meilleure consommation mémoire en pic
- Collecte anticipée

  $\Longrightarrow$ Gain *plus important* que la taille des régions
- Espace exploitable dans un contexte multithreadé

Progrès vis-à-vis du GC

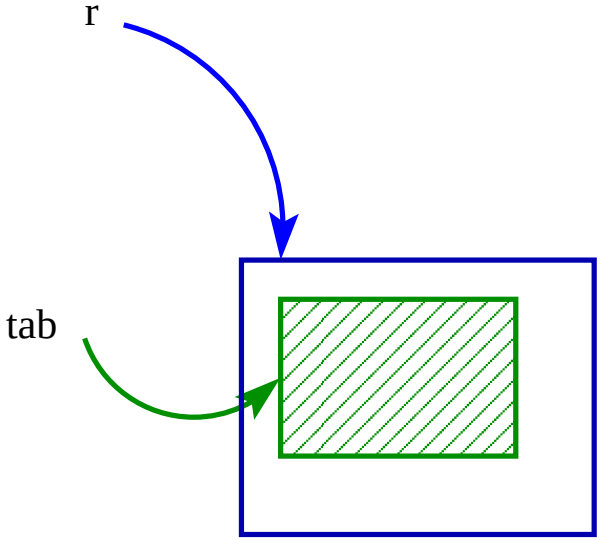


Même avec une faible part des sites d'allocation, on gagne quand même en moyenne

3: Une région pour une méthode ?

L'approche basée sur les méthodes est-elle meilleure que l'approche non structurée ?

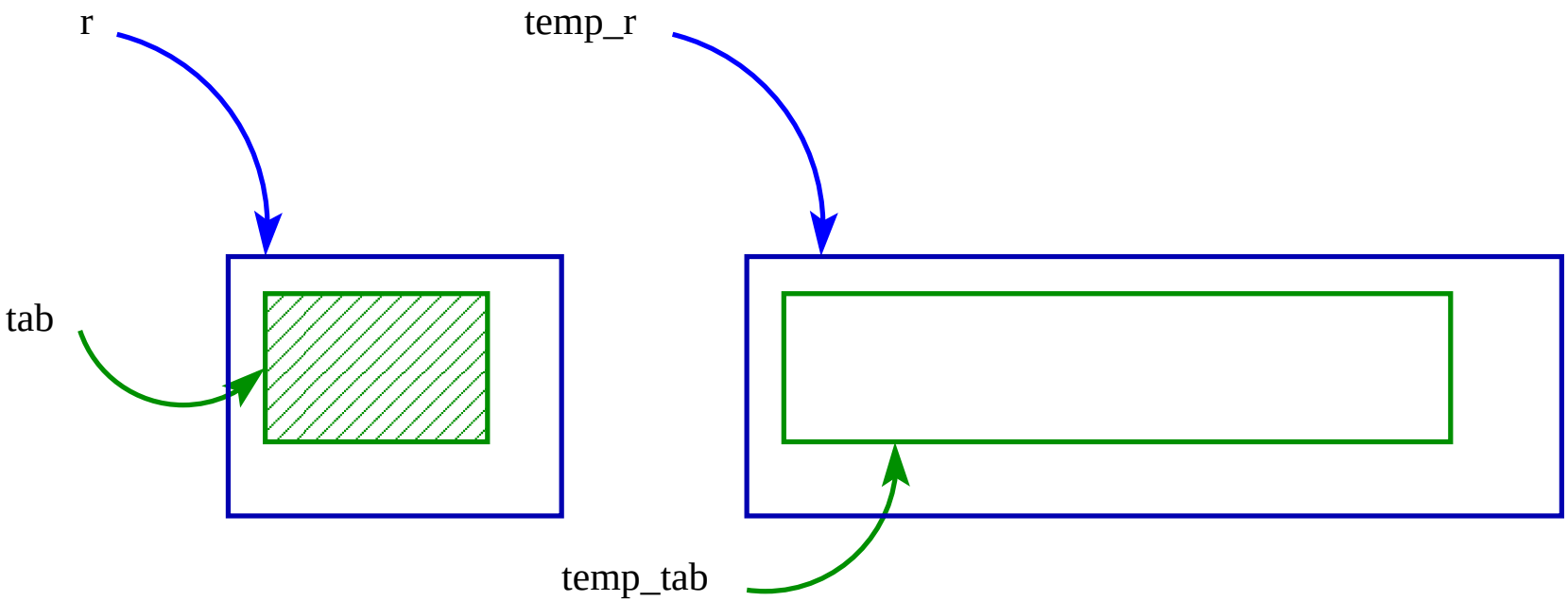
Un cas courant: la réallocation de tableau :



Le tableau `tab` est rempli.

3: Une région pour une méthode ?

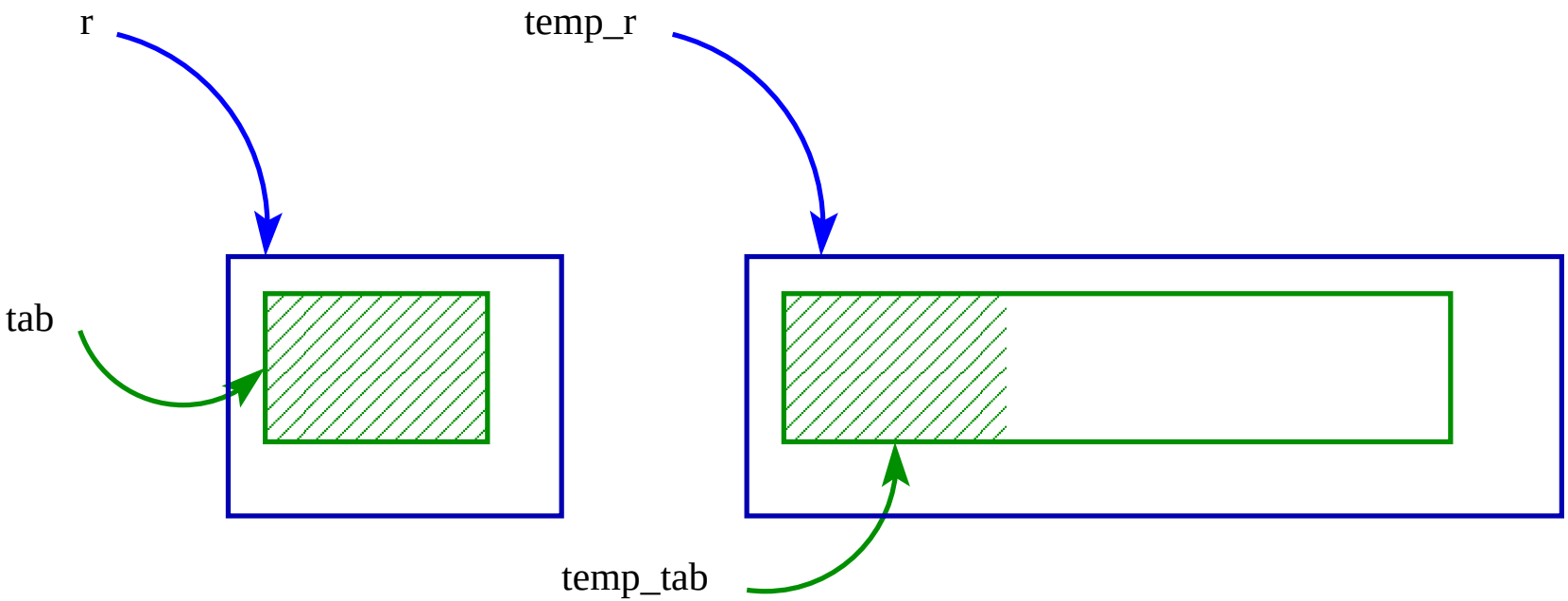
Un cas courant: la réallocation de tableau :



en RC: on alloue un nouveau tableau, dans une nouvelle région.

3: Une région pour une méthode ?

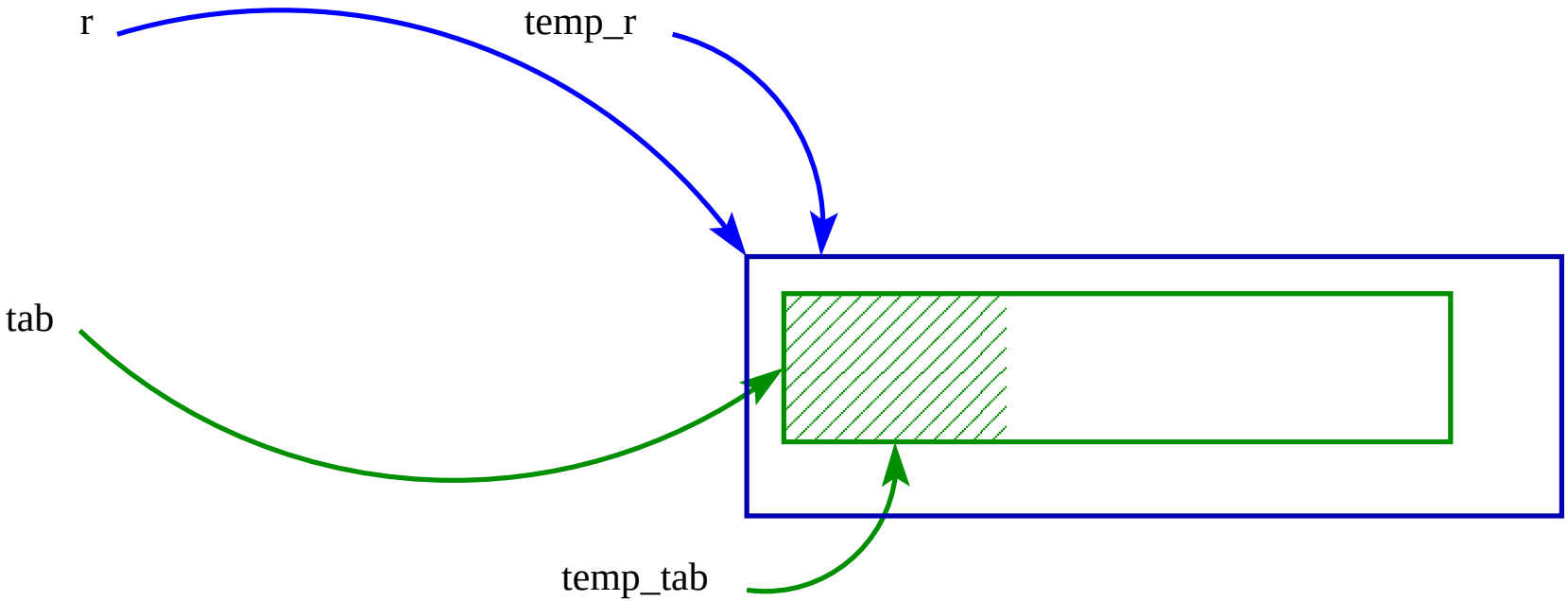
Un cas courant: la réallocation de tableau :



On recopie le contenu de l’ancien tableau.

3: Une région pour une méthode ?

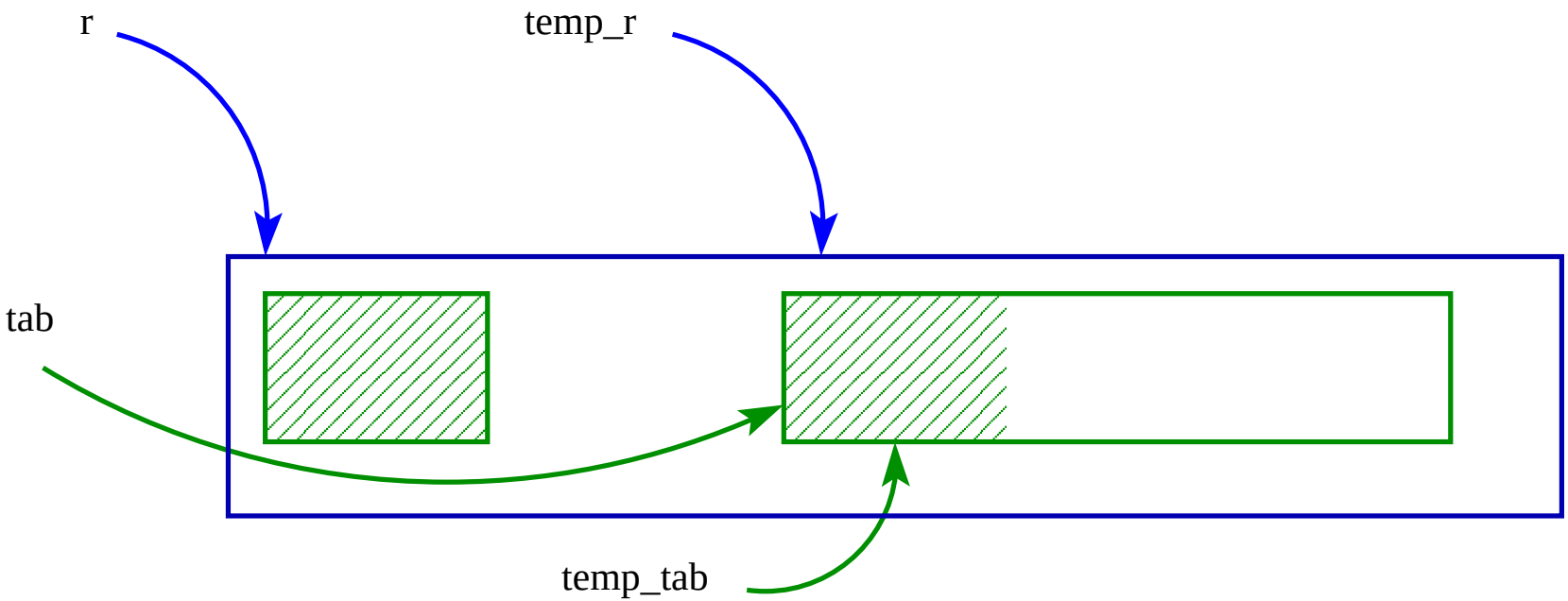
Un cas courant: la réallocation de tableau :



On supprime les anciens `r` et `tab`.

3: Une région pour une méthode ?

Un cas courant: la réallocation de tableau :

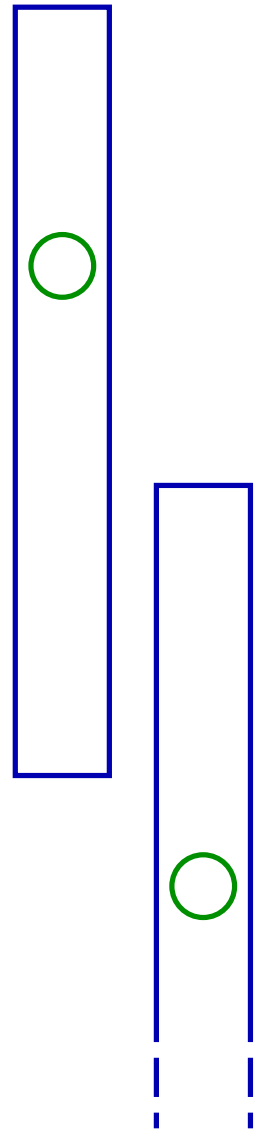


Madeja: les deux tableaux sont alloués dans la région associée à la méthode.

Solution = chevauchement des durées de vie des régions ?

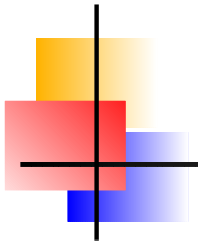
Chevauchement

```
m0:
    enter(r1)
       .
       x = new(r1)
       .
       .
       enter(r2)
       .
    exit(r1)
    .
    y = new(r2)
    .
```



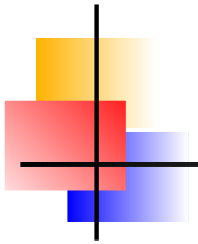
Plan: Conclusions et perspectives

- Contexte
- État de l’art
- Notre approche
- Contribution
- *Conclusions et perspectives*



Conclusions

- Un prototype fonctionnel pour expérimenter sur la théorie
  - Régions: économisent de la mémoire
  - Beaucoup d'allocations non syntaxiques
- Des conclusions quant à l'analyse d'échappement
  - Analyse peu efficace au niveau source
  - une région par méthode : inadéquat



Perspectives

- Gestion des régions
  - Instrumenter le code natif
  - Répartition de charge entre les régions

    $\Longrightarrow$ conflit avec l'entrelacement des régions ?
- Analyse d'échappement
  - Passer à plus bas niveau (Bytecode)
  - (Pré)analyser les bibliothèques
  - Varier les imbrications à divers points du GFC
  - Faire le lien avec l'ordonnancement
  - Identifier les paradigmes de programmation ?

Merci pour votre attention.

Des questions ?