Université Libre de Bruxelles
Département d'Informatique

MEMO-F524 Mémoire

Parallélisation d'un interpréteur abstrait au travers du modèle acteur

— Application à l'interpréteur abstrait Scala-AM —

Présentation finale

Olivier Pirson — opi@opimedia.be

5 septembre 2019

https://speakerdeck.com/opimedia/parallelisation-dun-interpreteur-abstrait-au-travers-du-modele-acteur

Vrije Universiteit Brussel

Promoteurs Coen De Roover
Wolfgang De Meuter
Conseiller Quentin Stiévenart

D'un problème non calculable à une technique suffisamment rapide

Nous voulons des **programmes corrects**.

↓

Analyse statique :
automatiser la preuve de propriétés sur les programmes
(ou la mise en évidence de bogues)
sans exécution des programmes.

Pour chaque propriété non triviale, c'est un problème **non calculable**.
(théorème de Rice)

↓

Interprétation abstraite :
technique d'analyse statique qui
évalue une abstraction (sur-approximation) des programmes.

Calculable mais peut être trop **lent**
(en particulier pour langages dynamiques avec fonctions d'ordre supérieur).

↓

Possibilité pour la rendre plus rapide : **paralléliser**...

Parallélisation de Scala-AM

Scala-AM : framework écrit en Scala par Quentin Stiévenart.
Permet d'expérimenter avec les différents aspects d'un interpréteur abstrait.

↓

Restriction à la machine abstraite la plus simple (AAM), sans optimisation.

Restriction de l'analyse aux programmes Scheme.

↓

Ajout de **3×3 implémentations parallèles** (ParAAM∗)
utilisant les acteurs de la bibliothèque Akka.

↓

Scala-Par-AM
https://bitbucket.org/OPiMedia/scala-par-am

Interprétation abstraite

Abstraction de machine abstraite (Abstracting Abstract Machine, AAM) :
dérive systématiquement la sémantique d'un langage de programmation
pour en déduire une sémantique abstraite
qui mime (sur-approxime) la sémantique concrète.

Généralisation :
toutes les entrées sont considérées.

Simplifications :

- Les valeurs concrètes sont sur-approximées (par exemple par leur type).
- La sémantique concrète du langage est adaptée.

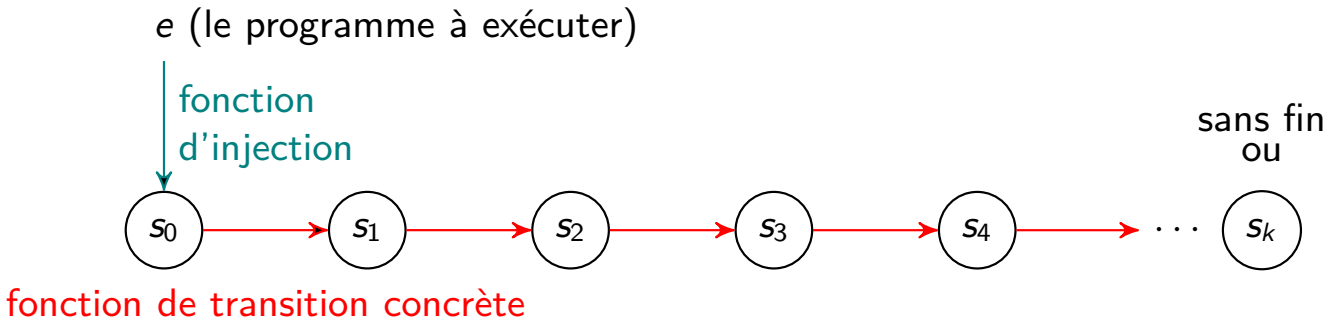
Basé sur la notion de treillis :
ordre partiel avec des propriétés supplémentaires
garantissant l'existence d'un point fixe,
et préservant au mieux les informations approchées.

Graphe d'états

Interpréteur concret selon la sémantique par petits pas.

Trace :

exécution concrète sur *une* seule entrée par un interpréteur concret.

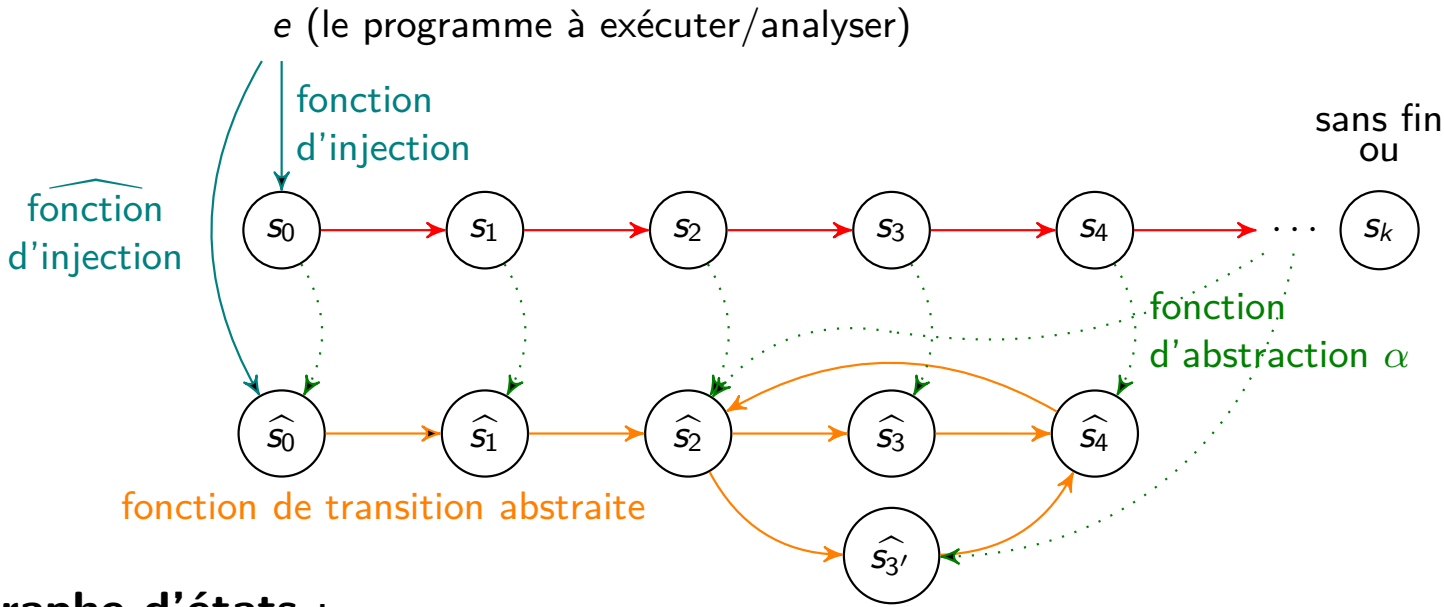


Graphe d'états

Interpréteur concret selon la sémantique par petits pas.

Trace :

exécution concrète sur *une* seule entrée par un interpréteur concret.

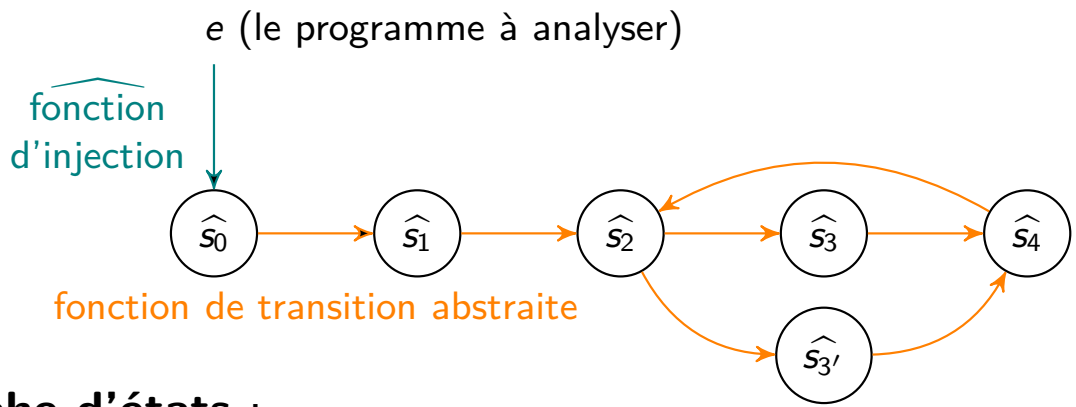


Graphe d'états :

interprétation abstraite sur *toutes* les entrées.

"L'abstraction simule le concret" [Might, 2011] sur *toutes* les entrées.

Graphe d'états



Graphe d'états :
interprétation abstraite sur *toutes* les entrées.

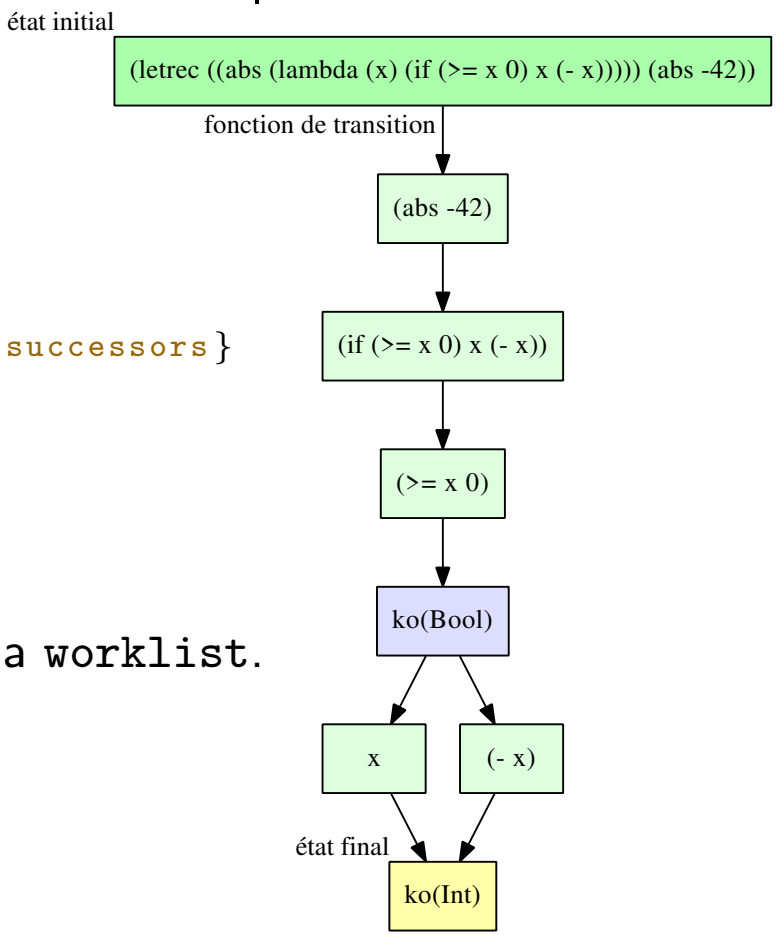
"L'abstraction simule le concret" [Might, 2011] sur *toutes* les entrées.

Algorithme séquentiel de calcul du point fixe


```
0  worklist = ensemble vide
1  visited = ensemble vide
2  halted = ensemble vide
3  graph = graphe vide
4
5  worklist = worklist ∪ {état initial}
6  repeat
7    state = prendre un état de worklist
8    if state ∉ visited then
9      visited = visited ∪ {state}
10     if state est un état final then
11       halted = halted ∪ {state}
12     else
13       successors = step(state)
14       worklist = worklist ∪ successors
15       graph = graph ∪ {arête state → s | s ∈ successors}
16 until worklist est vide
```

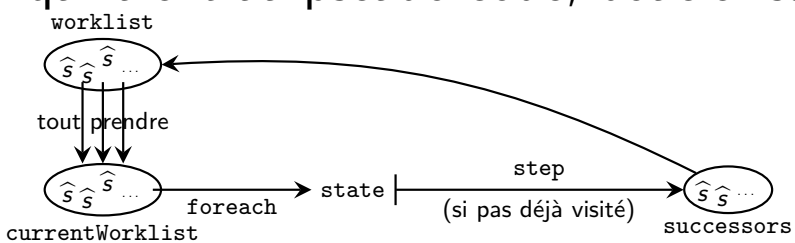
```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))

(abs -42)
```

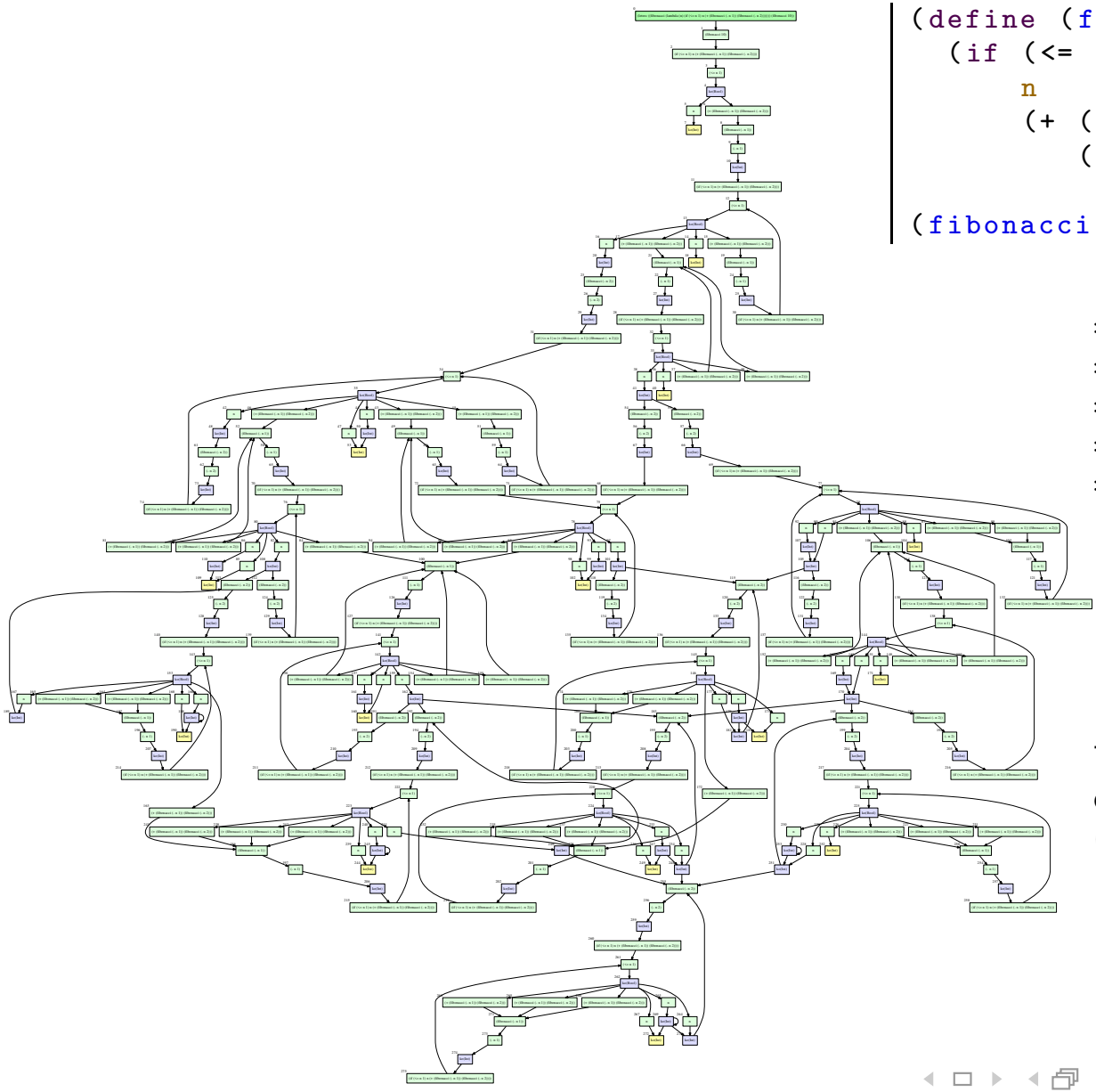


Le graphe d'états complet
ne dépend *pas* de l'ordre d'évaluation.
Crucial pour la parallélisation,
qui va évaluer en parallèle le contenu de la `worklist`.

Équivalent au pseudo-code, boucler sur



Autre simple exemple Scheme, avec un graphe d'états plus étoffé



```
(define (fibonacci n)
  (if (<= n 1)
      n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))

(fibonacci 10)
```

état : 276
arc : 346
valeur finale : 1 (`Int`)
déjà visité : 71
de successeurs par taille :
  # de taille 1 : 231
  # de taille 2 : 13
  # de taille 3 : 3
  # de taille 4 : 2
  # de taille 6 : 12

Taille maximale
de la `worklist` : 24
(pour le parcours séquentiel)

Métriques pour évaluer les performances

p : le nombre de processeurs/cœurs
$T_{séq}$: temps d'exécution de l'implémentation séquentielle de référence
T_{par} : temps d'exécution de l'implémentation parallèle considérée

accélération $= \frac{T_{séq}}{T_{par}}$ = nombre de fois plus rapide
(idéalement égale à p, mais généralement inférieure)

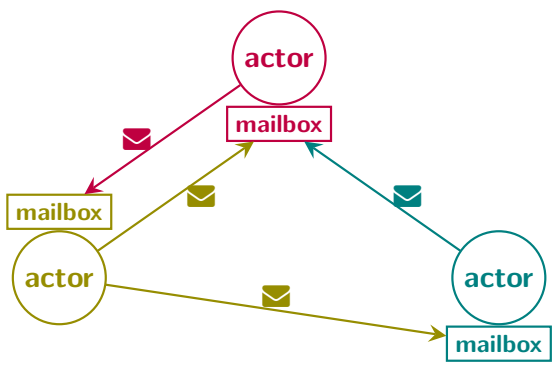
efficacité $= \frac{\text{accélération}}{p} = \frac{T_{séq}}{p\ T_{par}}$ = taux d'occupation des p proc./cœurs
(idéalement égale à 1, mais généralement inférieure)

Modèle d'acteurs

Modèle de parallélisme de haut niveau constituées d'acteurs.
(Modèle par défaut en Scala, utilisant la bibliothèque Akka.)

Acteur :

- Entité isolée, avec des données internes privées.
- Communique avec les autres acteurs par envoi de message asynchrone.
- Dispose d'une boîte aux lettres
  dans laquelle s'accumule les messages reçus (dans l'ordre d'arrivée).
- Chaque message réceptionné (un à la fois, dans l'ordre d'arrivée)
  déclenche le code associé qui s'exécute séquentiellement.
- ...



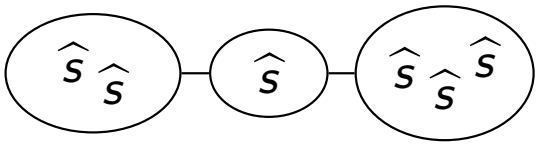
Tous les acteurs s'exécutent en parallèle
(si les ressources matérielles le permettent).

Évocation des 3×3 algorithmes parallèles implémentés

Par message envoie	**Loop – SenderAggregator**	**Concurrent – Sender**	**Concurrent**
un seul état	L-SA-state	C-S-state	C-state
ensemble d'états	L-SA-set	C-S-set	C-set
part "égale" d'états	**L-SA-part**	**C-S-part**	**C-part**

3 familles d'algorithmes, s'améliorant de gauche à droite.
Déclinée en 3 versions, s'améliorant du haut vers le bas.

Les versions $*$-set utilisent la découpe "naturelle" en ensembles induite par la structure hybride de `worklist` choisie :
une liste d'ensembles d'états.



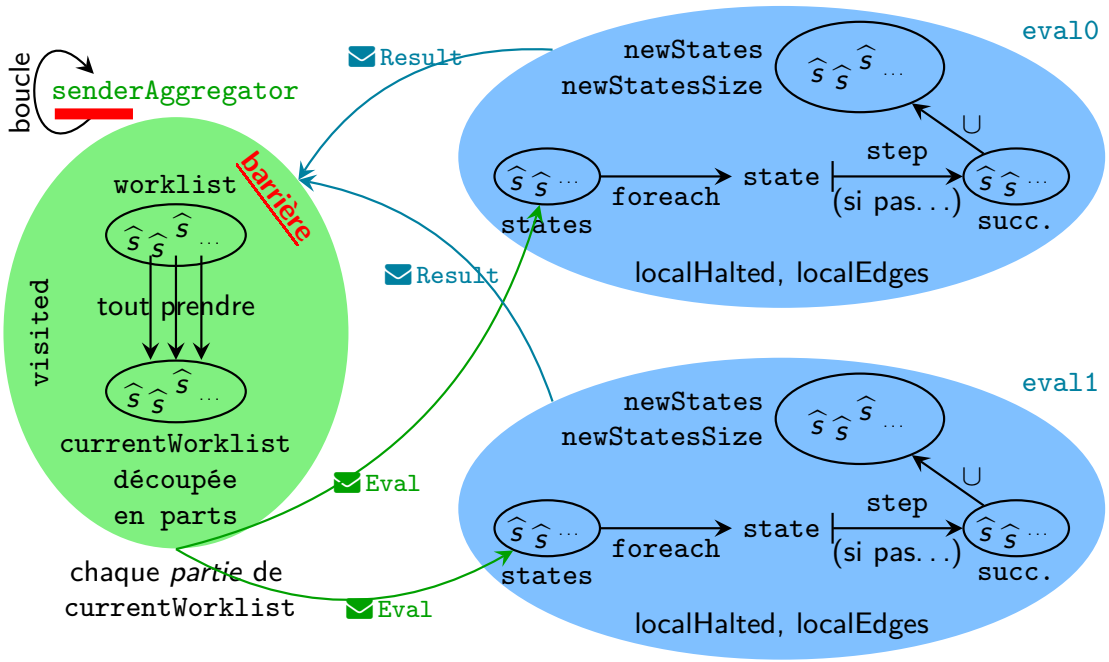
Chacun de ses ensembles correspond, suivant les versions,
à un ensemble de successeurs ou à l'accumulation locale des successeurs.

Pour chaque algorithme,
p acteurs sont créés lors de l'initialisation et utilisés jusqu'à la fin.

$p = 3$ dans les schémas qui suivent.

Algorithme parallèle L-SA-part

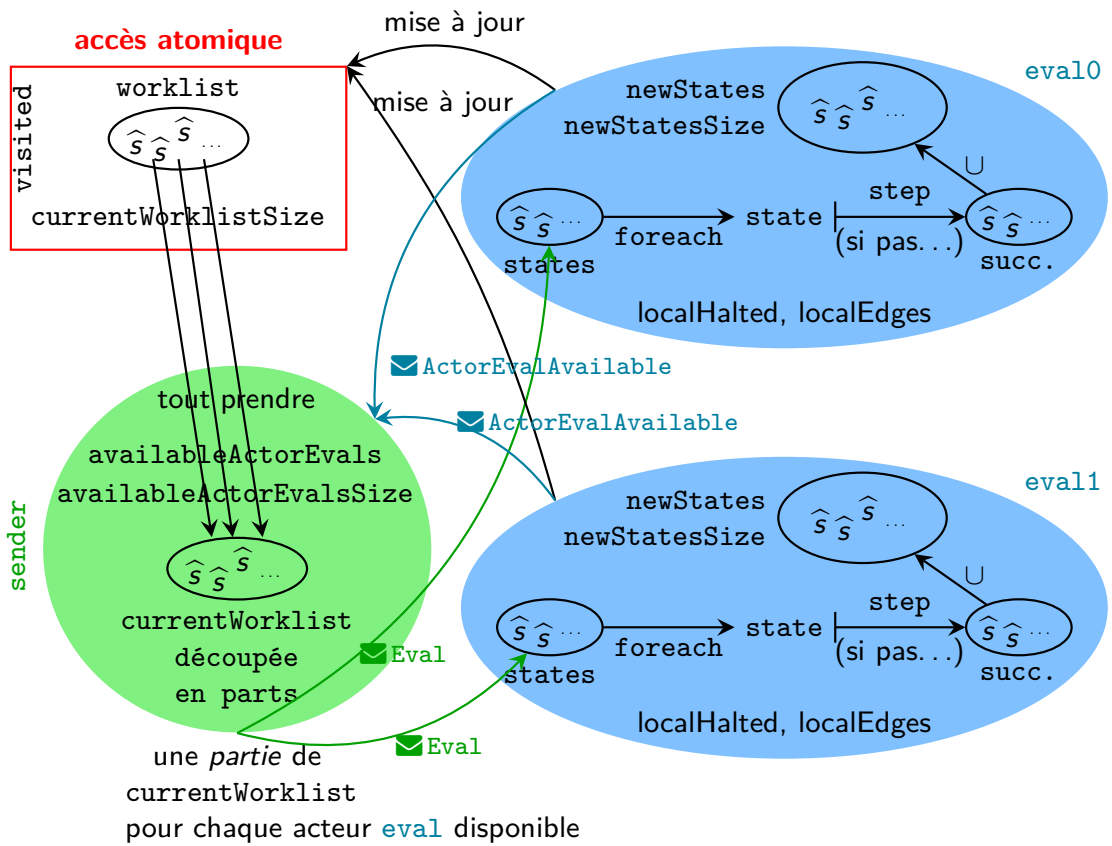
Dans une boucle, un acteur spécifique unique `senderAggregator` envoie les états aux $n = p - 1$ acteurs `eval` et réceptionne les états à ajouter à la `worklist`.



- Un acteur spécifique n'effectue aucune évaluation.
- Barrière, à chaque tour de boucle.

Algorithme parallèle C-S-part

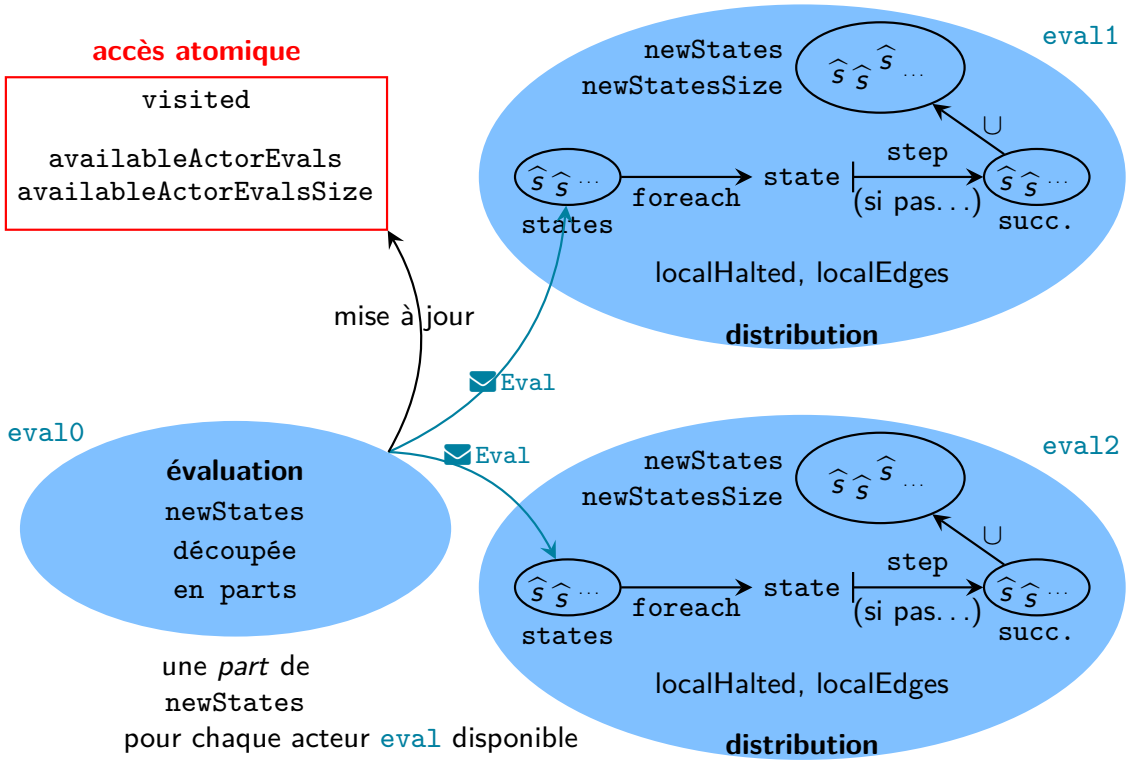
Un acteur spécifique unique `sender`
envoie les états aux $n = p - 1$ acteurs `eval`
qui eux-mêmes effectuent les mises à jour de façon concurrente.



Un acteur spécifique n'effectue aucune évaluation

Algorithme parallèle C-part

Chacun des $n = p$ acteurs `eval` envoie les états aux autres et effectue les mises à jour de façon concurrente.



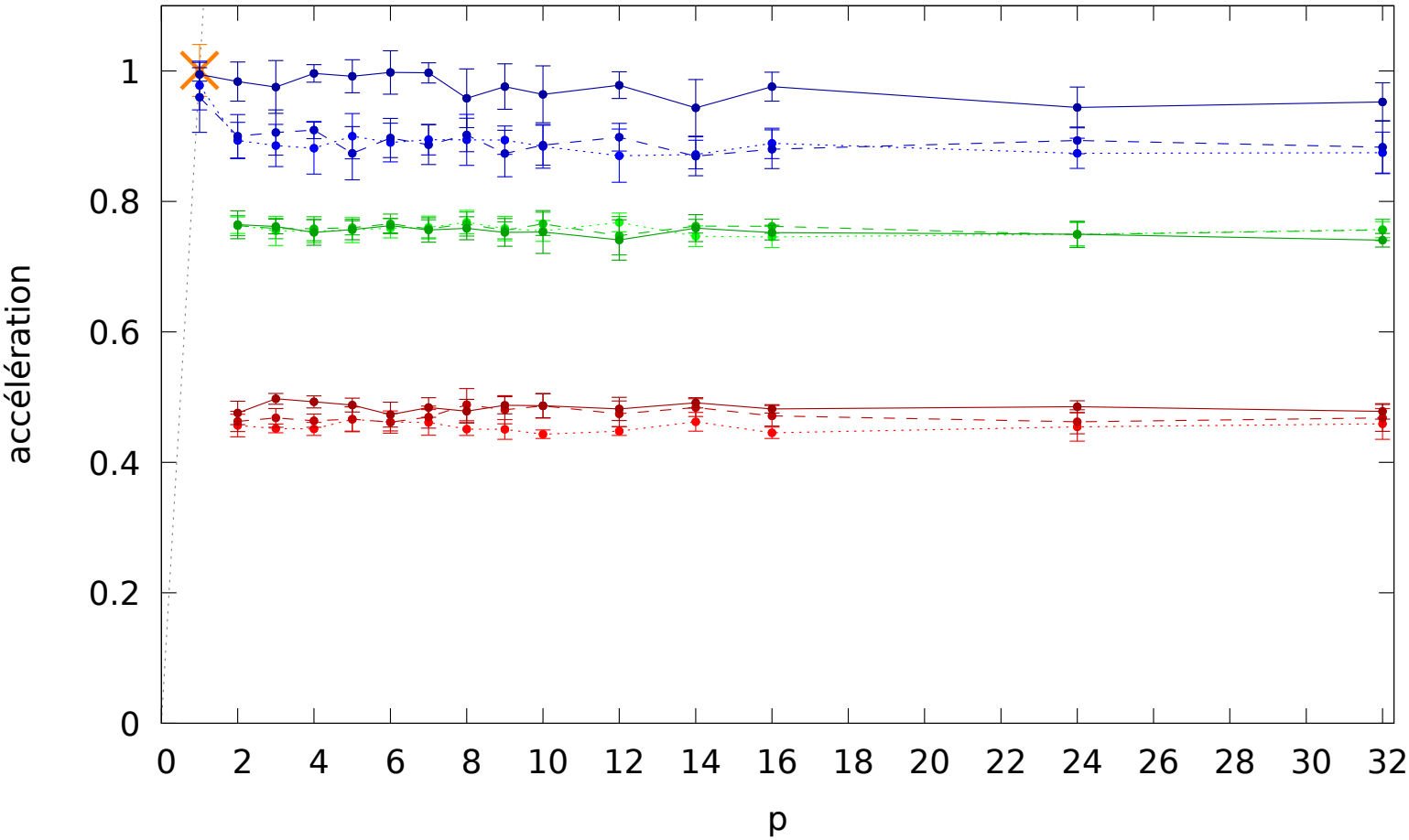
Dans cette version C-*part* il n'y a plus de `worklist` globale.

Ordinateur utilisé : *Bertha*

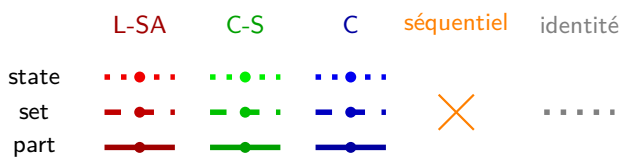
Sauf mention contraire,
les résultats qui suivent ont été calculés sur l'ordinateur *Bertha* :
2 processeurs de 4 cœurs physiques $\times 2$ avec l'Hyper-Threading.
Donc **8 cœurs physiques** et 16 cœurs logiques.

Moyennes sur 10 ou 30 répétitions.

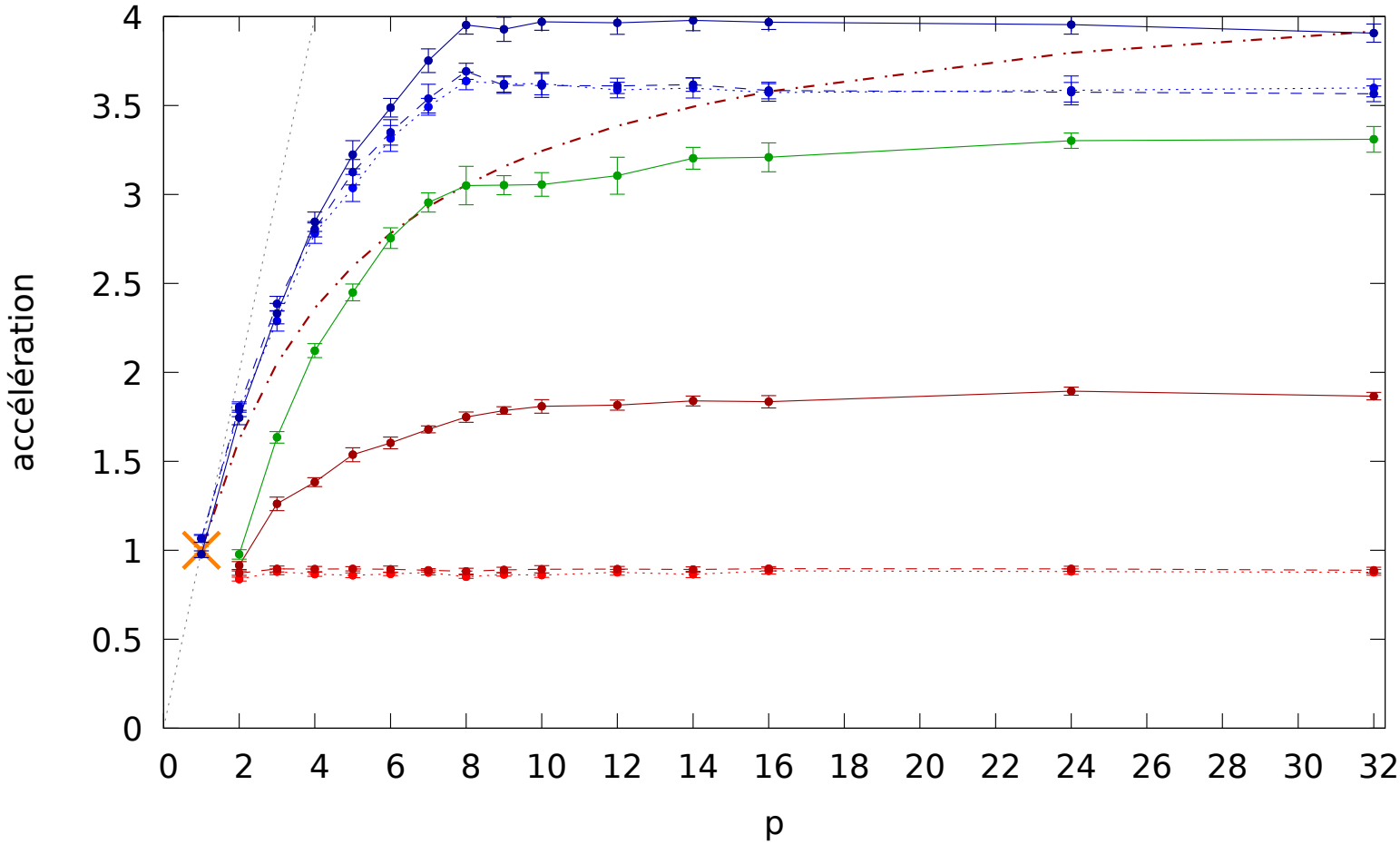
Accélération pour le programme Scheme `linear_10000`



Le graphe d'états pour `linear_10000` est complètement linéaire, donc aucune (selon nos algorithmes) parallélisation n'est possible.

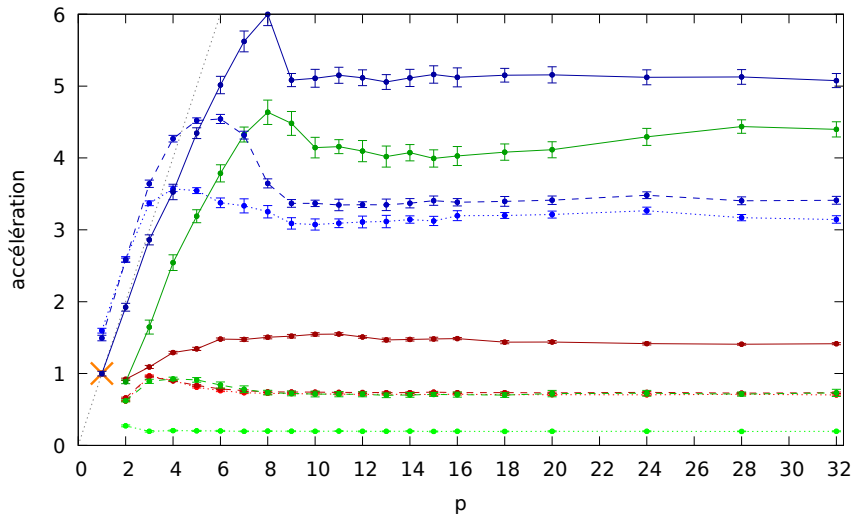


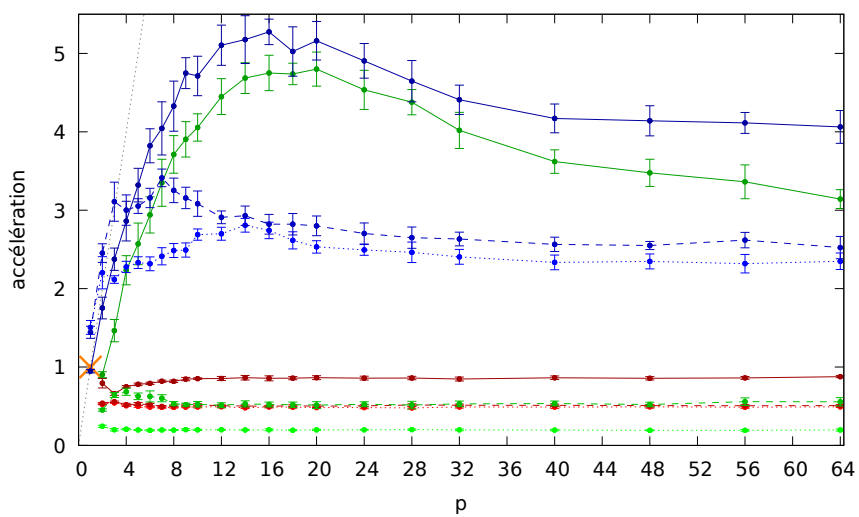
Accélération pour `linear_5000_gen4fibonacci_recur`



Le graphe d'états avec une chaîne initiale non négligeable, d'une proportion $\beta = \frac{L}{N} = \frac{15006}{64855} \simeq 0,231$.
La courbe en marron exprime la borne supérieure indicative pour cette longueur de chaîne initiale : accélération $= \frac{p}{(p-1)\beta+1}$.
$p = 8 =$ nombre de cœurs physiques

Accélération pour `primtest`

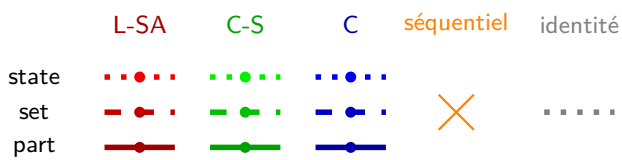




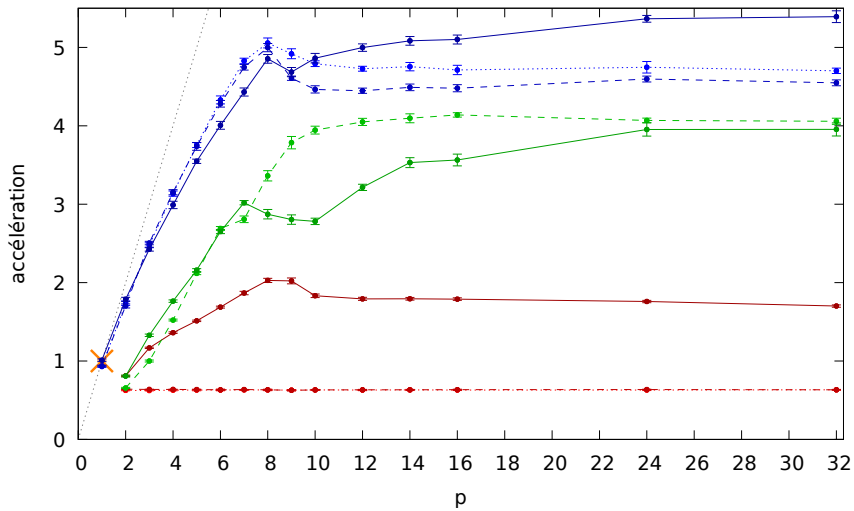
À gauche sur l'ordinateur *Bertha* avec 8 cœurs physiques.

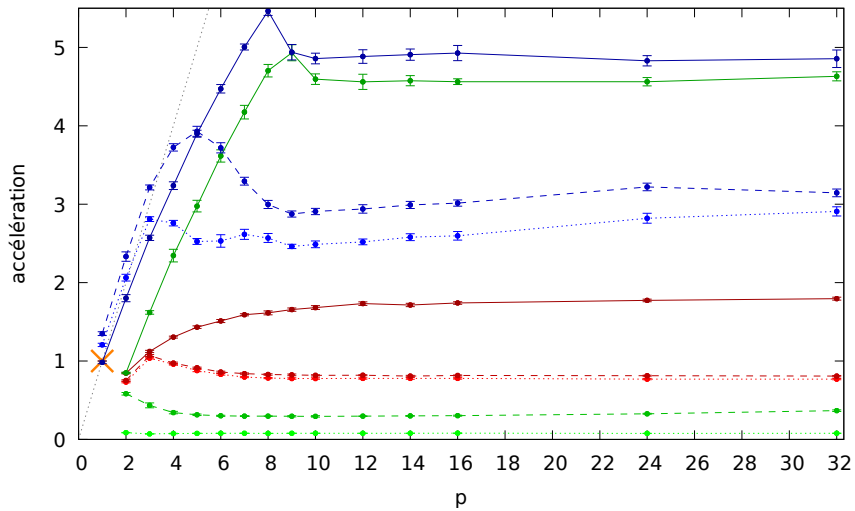
Les résultats de droite ont été calculés sur l'ordinateur *Serenity* :
4 processeurs de 16 cœurs physiques.
Donc 64 cœurs physiques.

Les meilleurs algorithmes profitent des cœurs physiques
mais jusqu'à une certaine limite.



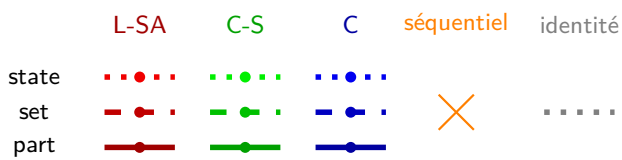
Accélération pour `kcfa_worst_case_40` et `qsort`



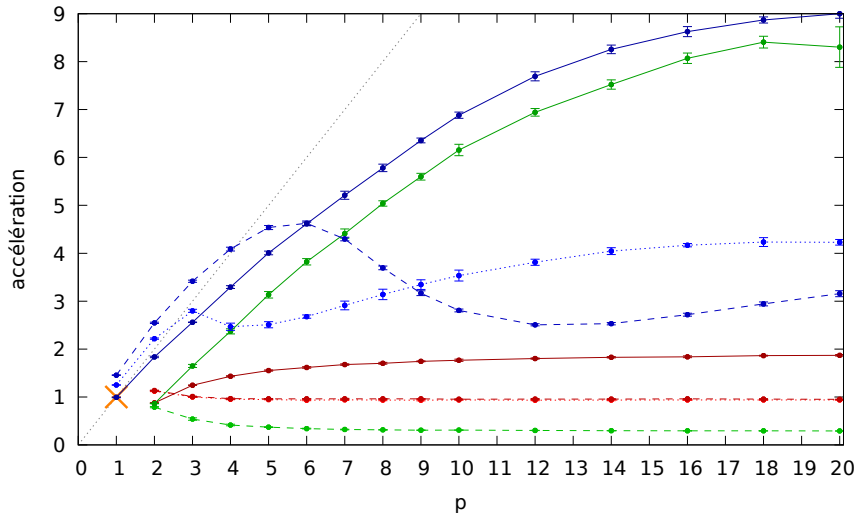


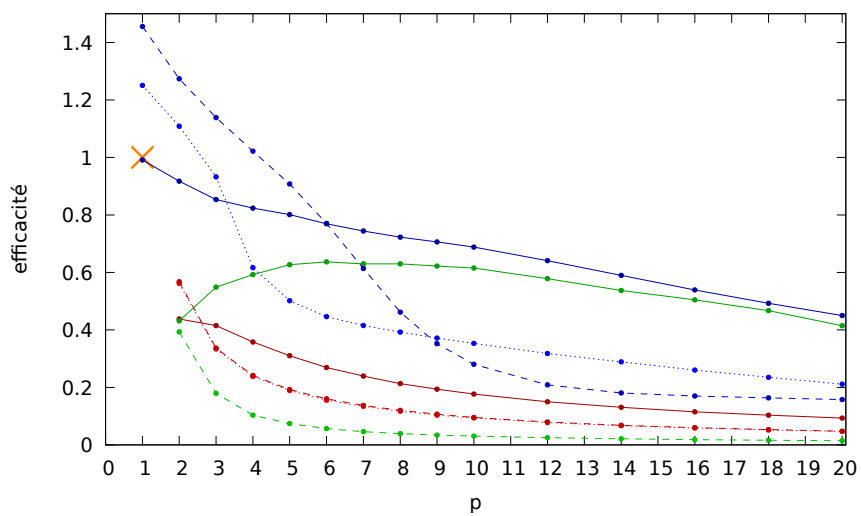
À gauche pour `kcfa_worst_case_40`, à droite pour `qsort`.

Les performances de l'algorithme C-S-set
varient énormément d'un programme Scheme à l'autre.



Accélération et efficacité pour `qsort` sur *Hydra*

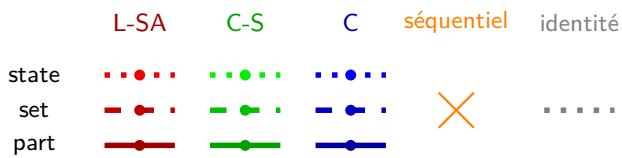




Les résultats qui suivent ont été calculés sur un ordinateur du cluster *Hydra* : 2 processeurs de 20 cœurs physiques.
Donc 40 cœurs physiques. (Mais uniquement 20 démandés.)

Confirme que le meilleur algorithme est C-part.
La différence avec C-S-part
est due à l'acteur en moins qui effectue les évaluations.

Ce sont les meilleurs résultats obtenus.
Accélération croissante jusque $p = 20$.



Résumé des performances

Par message envoie	Loop – SenderAggregator	Concurrent – Sender	Concurrent
un seul état	L-SA-state	C-S-state	C-state
ensemble d'états	L-SA-set	C-S-set	C-set
part "égale" d'états	L-SA-part	C-S-part	C-part
# acteur `eval`	$n = p - 1$	$n = p - 1$	$n = p$
Contrôle	barrière entre itérations	synchronisation	synchronisation
Envoi état(s) par	l'acteur spécifique envoie tout	l'acteur spécifique associe disponibilité états/acteurs	chaque `eval` associe disponibilité états/acteurs
Mise à jour par	l'acteur spécifique	chaque `eval`	chaque `eval`
Performances	mauvaises	variables	bonnes

De manière général les implémentations C-$*$ sont les meilleures.

Les résultats confirment que l'implémentation C-part,
supposée la meilleure selon les deux axes d'améliorations,
est bien la meilleure.

Les performances C-part sont bonnes lorsqu'une parallélisation est possible de part la structure du graphe d'états.
Et son surcout est minimisé dans les portions linéaires du graphe d'états.

Les résultats confirment aussi que les performances sont limitées par la structure du graphe d'états.

Résumé

L'objectif était d'accélérer l'interprétation abstraite en la parallélisant.

Le framework Scala-AM a servi de base
avec les acteurs de la bibliothèque Akka.

3 familles successives d'algorithmes ont été implémentées et évaluées :
la 1re L-SA-$*$ directement issue de l'algorithme séquentiel,
la 2e C-S-$*$ supprimant la barrière,
et la 3e C-$*$ n'ayant plus d'acteur spécifique.

Les résultats confirment les performances de la 3e C-$*$
et en particulier du meilleur algorithme supposé C-part.

Perspectives

- Utiliser les mesures statistiques implémentées dans Scala-Par-AM et les mettre en relation avec les performances.
  Afin de mieux comprendre la dynamique d'évaluation des algorithmes et tenter d'évaluer leur degré de parallélisation possible.
- Évaluer les performances avec d'autres abstractions, en pratiquant des analyses en particuliers.
- Évaluer les performances dans un cadre distribué (en excluant les versions *state*).
- Évaluer les meilleures implémentations sur de plus gros programmes Scheme, plus réalistes.
- Tester la structure hybride de la `worklist` sur de plus gros programmes Scheme.

Questions/réponses. . .

Références :

- **Cette présentation** sur SD :
  https://speakerdeck.com/opimedia/parallelisation-dun-interpreteur-abstrait-au-travers-du-modele-acteur
- **Mémoire** *Parallélisation d'un interpréteur abstrait, au travers du modèle acteur — Application à l'interpréteur abstrait Scala-AM* :
  https://bitbucket.org/OPiMedia/efficient-parallel-abstract-interpreter-in-scala/
- **Implémentation** :
  Scala-Par-AM – https://bitbucket.org/OPiMedia/scala-par-am/
- **Résultats** :
  http://www.opimedia.be/CV/2017-2018-ULB/MEMO-F524-Masters-thesis/benchmark-results/