Principes et Pratiques de la Programmation Concurrente en π-calcul

Frédéric Peschanski

UPMC – LIP6 – Equipe APR

Journées Francophones des Langages Applicatifs - 2010

Plan

Plan

La concurrence

Pourquoi ?

The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software
Herb Sutter, Dr Dobb's Mai 2005

La concurrence

Pourquoi ?

The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software
Herb Sutter, Dr Dobb's Mai 2005

- L'avénement du multi-coeurs replace la concurrence au centre des préoccupations
- mais il y a du neuf (un peu) :
  - nouveau *hardware* : multicœurs généralistes, *hyper-threading*, GPU, etc.
  - nouvelles techniques : algorithmes *lock-free/wait-free*, mémoires transactionnelles, etc.
  - et une **théorie de la concurrence** : algèbres de processus, P/T nets, etc.

Splendeur et misère du π-calcul

Splendeur

- Un langage **minimal**
  pour décrire des systèmes concurrents et dynamiques
- Un langage très **expressif**
- Un (trop ?) large **corpus théorique**

Splendeur et misère du π-calcul

Splendeur

- Un langage **minimal**
  pour décrire des systèmes concurrents et dynamiques
- Un langage très **expressif**
- Un (trop ?) large **corpus théorique**

Misère

- Une théorie à géométrie variable (early, late, open, etc.)
- Manque d'outils de modélisation et de vérification
- Manque d'implémentations (stables, fidèles, efficaces, etc.)

Plan

Le langage : syntaxe

Definition		**def** $D(x_1,\ldots,x_n) = P$		Définition
Process	P	$::=$	**end**	Terminaison
			$\mid \sum_i [g_i]\alpha_i, P_i$	Choix gardés
			$\mid D(v_1,\ldots,v_n)$	Appel
Prefix	α	$::=$	**tau**	Pas interne
			$\mid c\,!v$	Emission
			$\mid c\,?(x)$	Réception
			$\mid$ **new**(c)	Création de canal
			$\mid$ **spawn**$\{P\}$	Création de thread

Le langage : syntaxe

Definition		**def** $D(x_1,\ldots,x_n) = P$		Définition
Process	P	$::=$	**end**	Terminaison
			$\mid \sum_i [g_i]\alpha_i, P_i$	Choix gardés
			$\mid D(v_1,\ldots,v_n)$	Appel
Prefix	α	$::=$	**tau**	Pas interne
			$\mid c\,!v$	Emission
			$\mid c\,?(x)$	Réception
			$\mid$ **new**(c)	Création de canal
			$\mid$ **spawn**$\{P\}$	Création de thread

\+ sucre syntaxique : **if-then-else**, opérateur $||$, etc.

Le langage : syntaxe

Definition		**def** $D(x_1,\ldots,x_n) = P$		Définition
Process	P	$::=$	**end**	Terminaison
			$\mid \sum_i [g_i]\alpha_i, P_i$	Choix gardés
			$\mid D(v_1,\ldots,v_n)$	Appel
Prefix	α	$::=$	**tau**	Pas interne
			$\mid c\,!v$	Emission
			$\mid c\,?(x)$	Réception
			$\mid$ **new**(c)	Création de canal
			$\mid$ **spawn**$\{P\}$	Création de thread

+ sucre syntaxique : **if-then-else**, opérateur $||$, etc.

+ système de type

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r !m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
new(c :chan<int>), spawn{Fib(5,0,1,c)},c ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
new(c :chan<int>), spawn{Fib(5,0,1,c)},c?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
spawn{Fib(5,0,1,ĉ)},ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
spawn{Fib(5,0,1,ĉ)},ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(5,0,1,ĉ),
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[5=0] ĉ!0,end + [true] tau,Fib(4,1,0,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[5=0] ĉ!0,end + [true] tau,Fib(4,1,0,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(4,1,0,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[4=0] ĉ!1,end + [true] tau,Fib(3,1,1,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(3,1,1,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(2,2,1,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(1,3,2,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
Fib(0,5,3,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[0=0] ĉ!5,end + [true] tau,Fib(-1,8,5,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r!m,end
  + [true] tau,Fib(n-1,m+p,m,r)
```

```
[0=0] ĉ!5,end + [true] tau,Fib(-1,8,5,ĉ)
|| ĉ?(n),print(n)
```

Exemple : récursion terminale

Skip

```
def Fib(n m p :int, r : chan<int>)=
  [n=0] r !m,end
  + [true] tau,Fib(n-1,m+p,m,r)

---

end
|| print(5)
```

Récursion non-terminale : encodage

```
def Ack(n p : int, r : chan<int>) =
  [n=0] r!(p+1)
  + [p=0] Ack(n-1,1,r)
  + new(r1 :chan<int>), [ r1?(pp),Ack(n-1,pp,r) || Ack(n,p-1,r1) ]
```

Récursion non-terminale : encodage

```
def Ack(n p : int, r : chan<int>) =
  [n=0] r!(p+1)
  + [p=0] Ack(n-1,1,r)
  + new(r1 :chan<int>), [ r1?(pp),Ack(n-1,pp,r) || Ack(n,p-1,r1) ]
```

Peu efficace ?

Récursion non-terminale : encodage

```
def Ack(n p : int, r : chan<int>) =
  [n=0] r!(p+1)
  + [p=0] Ack(n-1,1,r)
  + new(r1 :chan<int>), [ r1?(pp),Ack(n-1,pp,r) || Ack(n,p-1,r1) ]
```

Peu efficace ? pas si sûr (canaux linéaires . . .)

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\mathrm{TaskPool}(2,\hat{e},\hat{l}) \parallel \mathrm{Task}(\hat{e}) \parallel \mathrm{Task}(\hat{e}) \parallel \mathrm{Task}(\hat{e}) \parallel \mathrm{Task}(\hat{e})$

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

```
l̂ ?, TaskPool(3,ê,l̂)
+ [2>0] ê ?(release), Permit(release,l̂) || TaskPool(1,ê,l̂)
|| new(rel :chan<>),ê !(rel), /* task behavior */, rel !
|| new(rel :chan<>),ê !(rel), /* task behavior */, rel !
|| new(rel :chan<>),ê !(rel), /* task behavior */, rel !
|| new(rel :chan<>),ê !(rel), /* task behavior */, rel !
```

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| $\hat{e}$!($\hat{r1}$), /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{l}$?, TaskPool(3,$\hat{e}$,$\hat{l}$)
+ [2>0] $\hat{e}$?(release), Permit(release,$\hat{l}$) || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| $\hat{e}$!($\hat{r1}$), */* task behavior */*, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\mathsf{Permit}(\hat{r1},\hat{l}) \parallel \mathsf{TaskPool}(1,\hat{e},\hat{l})$
$\parallel$ **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
$\parallel$ ** task behavior **/, $\hat{r1}$!
$\parallel$ **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
$\parallel$ **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| */* task behavior */*, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !
|| **new**(rel :chan<>),$\hat{e}$!(rel), */* task behavior */*, rel !

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || TaskPool(0,$\hat{e}$,$\hat{l}$)
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior **/, rel !
|| /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r2}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| **new**(rel :chan<>),$\hat{e}$!(rel), ** task behavior */*, rel !
|| /* task behavior */, $\hat{r1}$!
|| **new**(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r2}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| $\hat{e}$!($\hat{r3}$), /* task behavior */, $\hat{r3}$!
|| /* task behavior */, $\hat{r1}$!
|| new(rel :chan<>),$\hat{e}$!(rel), /* task behavior */, rel !
|| /* task behavior */, $\hat{r2}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| $\hat{e}$!($\hat{r3}$), /* task behavior */, $\hat{r3}$!
|| /* task behavior */, $\hat{r1}$!
|| $\hat{e}$!($\hat{r4}$), /* task behavior */, $\hat{r4}$!
|| /* task behavior */, $\hat{r2}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
              || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r2}$?,$\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| $\hat{e}$!($\hat{r3}$), /* task behavior */, $\hat{r3}$!
|| /* task behavior */, $\hat{r1}$!
|| $\hat{e}$!($\hat{r4}$), /* task behavior */, $\hat{r4}$!
|| $\hat{r2}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{l}$! || $\hat{l}$?, TaskPool(1,$\hat{e}$,$\hat{l}$) + [0>0] ...
|| $\hat{e}$!($\hat{r3}$), /* task behavior */, $\hat{r3}$!
|| /* task behavior */, $\hat{r1}$!
|| $\hat{e}$!($\hat{r4}$), /* task behavior */, $\hat{r4}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || TaskPool(1,$\hat{e}$,$\hat{l}$)
|| $\hat{e}$!($\hat{r3}$), ** task behavior **/, $\hat{r3}$!
|| ** task behavior **/, $\hat{r1}$!
|| $\hat{e}$!($\hat{r4}$), ** task behavior **/, $\hat{r4}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}$?,$\hat{l}$! || $\hat{r4}$?,$\hat{l}$! || TaskPool(0,$\hat{e}$,$\hat{l}$)
|| $\hat{e}$!($\hat{r3}$), /* task behavior */, $\hat{r3}$!
|| /* task behavior */, $\hat{r1}$!
|| /* task behavior */, $\hat{r4}$!

Un exemple concurrent

Skip

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
  leave ?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter ?(release), Permit(release,leave)
                || TaskPool(nb-1,enter,leave)

def Permit(release leave : chan<>) = release ?,leave !

def Task(enter : chan<>) =
  new(rel :chan<>),enter !(rel), /* task behavior */, rel !
```

$\hat{r1}\,?,\hat{l}\,!\;||\;\hat{r4}\,?,\hat{l}\,!\;||\;$ TaskPool$(0,\hat{e},\hat{l})$
$||\;\hat{e}\,!(\hat{r3}),$ ** task behavior */*, $\hat{r3}\,!$
$||\;$ ** task behavior */*, $\hat{r1}\,!$
$||\;$ ** task behavior */*, $\hat{r4}\,!$

etc ...

Plan

Les π-threads et la PCM

Objectifs

- Fidélité (ex. : pas de + dans Pict)
- Efficacité (CML, GHC, ...)
- Parallélisme (GPH, Erlang, ...)

Les π-threads et la PCM

Objectifs

- Fidélité (ex. : pas de + dans Pict)
- Efficacité (CML, GHC, ...)
- Parallélisme (GPH, Erlang, ...)

Les π-threads (machine abstraite) et la PCM (machine virtuelle)

- Architecture sans pile
- Ordonnancement en «$O(1)$»
- GC concurrent

Concurrence et pile : sœurs énemies

Approches courantes

heavyweight 1 pile par thread (ex. Java threads)

lightweight 1 pile pour tous les threads (ex. Go threads)

Concurrence et pile : sœurs énemies

Approches courantes

heavyweight 1 pile par thread (ex. Java threads)

- ... mais *heavyweight* justement

lightweight 1 pile pour tous les threads (ex. Go threads)

Concurrence et pile : sœurs énemies

Approches courantes

heavyweight 1 pile par thread (ex. Java threads)

- ... mais *heavyweight* justement

lightweight 1 pile pour tous les threads (ex. Go threads)

- ... mais (très) complexe (découpage/migration de pile, etc.)

Concurrence et pile : sœurs énemies

Approches courantes

heavyweight 1 pile par thread (ex. Java threads)

- ... mais *heavyweight* justement

lightweight 1 pile pour tous les threads (ex. Go threads)

- ... mais (très) complexe (découpage/migration de pile, etc.)

Notre approche : pas de pile ! (buzz : *stackless*)

Concurrence et pile : sœurs énemies

Approches courantes

heavyweight 1 pile par thread (ex. Java threads)

- ... mais *heavyweight* justement

lightweight 1 pile pour tous les threads (ex. Go threads)

- ... mais (très) complexe (découpage/migration de pile, etc.)

Notre approche : pas de pile ! (buzz : *stackless*)

Pourquoi une pile ?

Concurrence et pile : sœurs énemies

Approches courantes

heavyweight 1 pile par thread (ex. Java threads)

- ... mais *heavyweight* justement

lightweight 1 pile pour tous les threads (ex. Go threads)

- ... mais (très) complexe (découpage/migration de pile, etc.)

Notre approche : pas de pile ! (buzz : *stackless*)

Pourquoi une pile ?

- appels non-terminaux
- environnement lexicaux imbriqués

Structure d'un π-thread

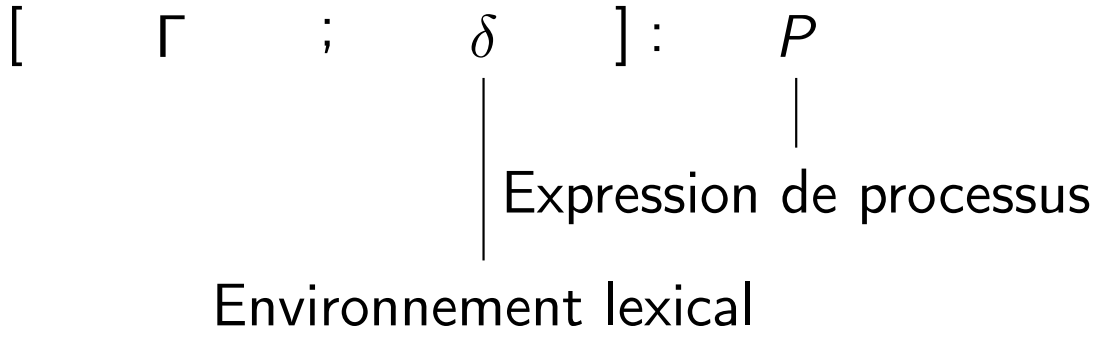
$$[\quad \Gamma \quad ; \quad \delta \quad] : \quad P$$

Structure d'un π-thread

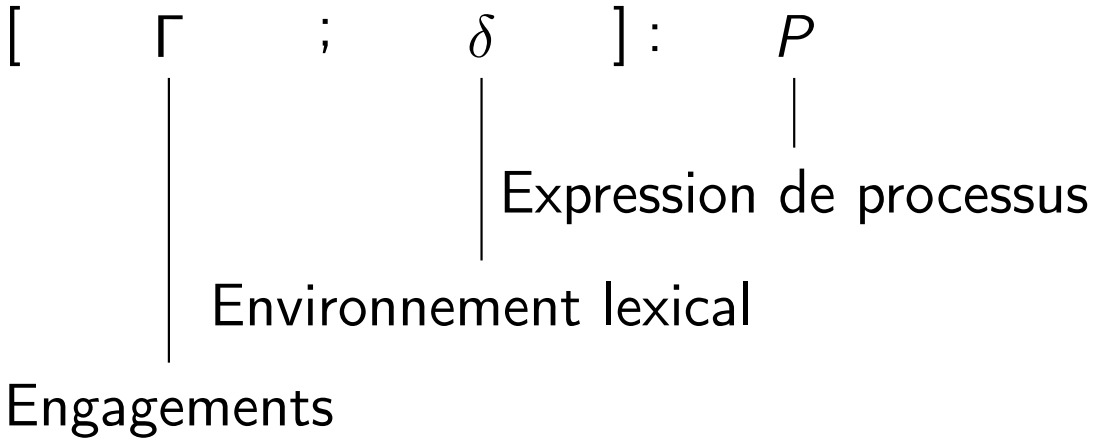
$$[\quad \Gamma \quad ; \quad \delta \quad] : \quad P$$

Expression de processus

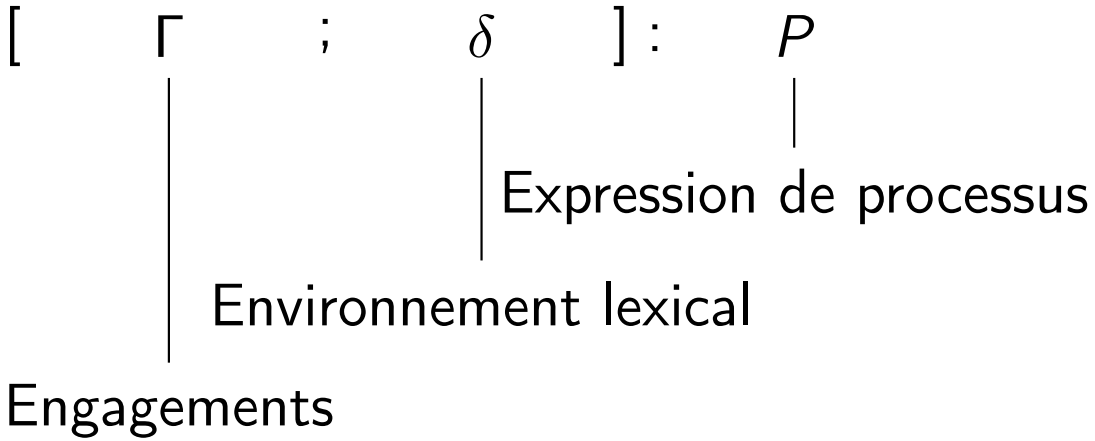
Structure d'un π-thread



Structure d'un π-thread

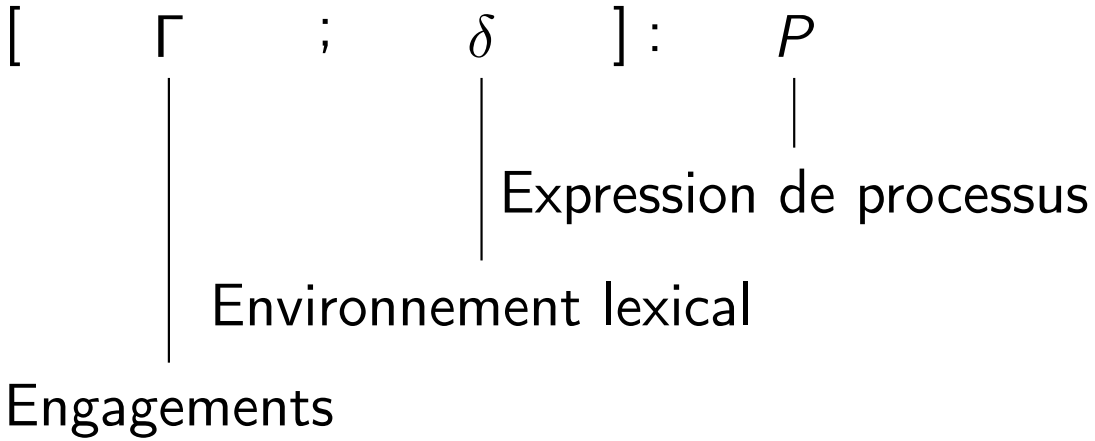


Structure d'un π-thread



Env. lex. «plat» (+ appels terminaux) $\Longrightarrow$ pas de pile [VEE06]

Structure d'un π-thread



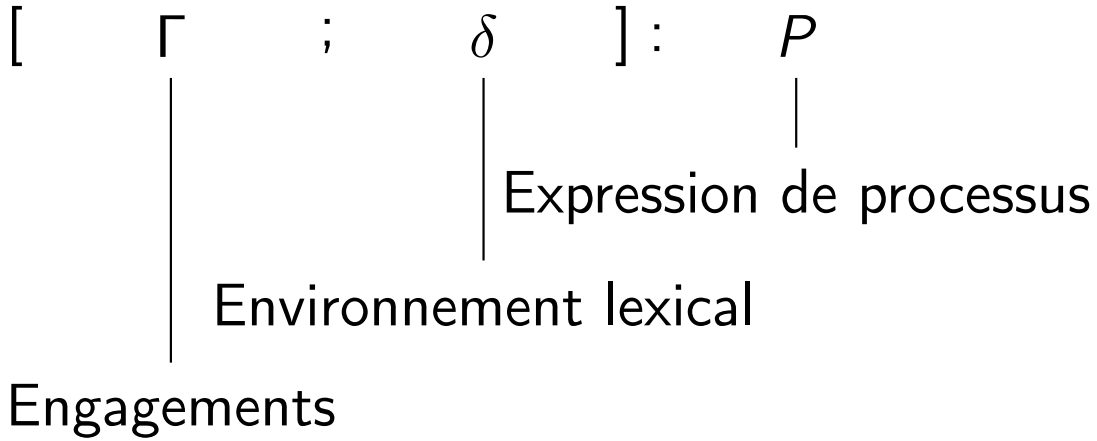
Env. lex. «plat» (+ appels terminaux) $\Longrightarrow$ pas de pile [VEE06]

Engagements

Emission $\hat{c} \Leftarrow v : Q$ canal $\hat{c}$ valeur v (ou expression e en *lazy*) et continuation Q

Réception $\hat{c} \Rightarrow x : Q$ canal $\hat{c}$ variable x et continuation Q

Structure d'un π-thread



Env. lex. «plat» (+ appels terminaux) $\Longrightarrow$ pas de pile [VEE06]

Engagements

Emission $\hat{c} \Leftarrow v : Q$ canal $\hat{c}$ valeur v (ou expression e en *lazy*) et continuation Q

Réception $\hat{c} \Rightarrow x : Q$ canal $\hat{c}$ variable x et continuation Q

Engagements explicites $\Longrightarrow$ ordonnancement en $O(1)$

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

Traçage parallélisation (très) complexe

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

- ... mais ramassage des cycles (très) complexe
- et problème d'efficacité (temps de comptage)

Traçage parallélisation (très) complexe

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

- ... mais ramassage des cycles (très) complexe
- et problème d'efficacité (temps de comptage)

Traçage parallélisation (très) complexe

- ... mais efficace

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

- ... mais ramassage des cycles (très) complexe
- et problème d'efficacité (temps de comptage)

Traçage parallélisation (très) complexe

- ... mais efficace

Notre approche : comptage de références globales sur les canaux

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

- ... mais ramassage des cycles (très) complexe
- et problème d'efficacité (temps de comptage)

Traçage parallélisation (très) complexe

- ... mais efficace

Notre approche : comptage de références globales sur les canaux

- parallélisation aisée

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

- ... mais ramassage des cycles (très) complexe
- et problème d'efficacité (temps de comptage)

Traçage parallélisation (très) complexe

- ... mais efficace

Notre approche : comptage de références globales sur les canaux

- parallélisation aisée
- efficacité

Ramasse-miettes

Ramasse-miettes concurrents

Comptage de références parallélisation aisée

- ... mais ramassage des cycles (très) complexe
- et problème d'efficacité (temps de comptage)

Traçage parallélisation (très) complexe

- ... mais efficace

Notre approche : comptage de références globales sur les canaux

- parallélisation aisée
- efficacité
- ramassage des cycles

Ramasse-miettes

Prédicats élémentaires

$$\mathrm{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\mathrm{def}}{=} \exists x \in \mathrm{dom}(\delta),\ \delta(x) = \hat{c}$$

$$\mathrm{wait}([\Gamma; \delta] : P) \stackrel{\mathrm{def}}{=} \mathrm{true}\ \textsf{si}\ P \equiv \sum \textbf{wait},\ \mathrm{false}\ \textsf{sinon}$$

Ramasse-miettes

Prédicats élémentaires

$$\text{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \text{dom}(\delta),\ \delta(x) = \hat{c}$$

$$\text{wait}([\Gamma; \delta] : P) \stackrel{\text{def}}{=} \text{true si } P \equiv \sum \textbf{wait},\ \text{false sinon}$$

Références globales

$$\text{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

Ramasse-miettes

Prédicats élémentaires

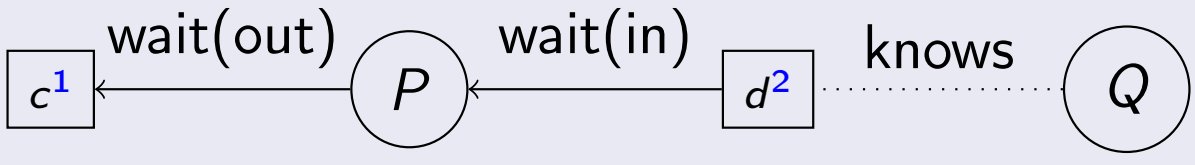
$$\text{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \text{dom}(\delta),\ \delta(x) = \hat{c}$$

$$\text{wait}([\Gamma; \delta] : P) \stackrel{\text{def}}{=} \text{true si } P \equiv \sum \textbf{wait},\ \text{false sinon}$$

Références globales

$$\text{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

Représentation



Ramassage des canaux

Règle

$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \to \Delta \vdash A} \ (\mathit{reclaim})$$

Ramassage des canaux

Règle

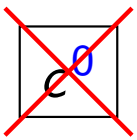
$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \rightarrow \Delta \vdash A} \ (\mathit{reclaim})$$

$\boxed{c^0}$

Ramassage des canaux

Règle

$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \rightarrow \Delta \vdash A} \ (\mathit{reclaim})$$



Ramassage des processus

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Ramassage des processus

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\begin{array}{l}\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \\ \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j\end{array}} \ (\mathit{stuck})$$

Principes

Ramassage des processus

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Principes

- Détection d'une clique de processus en attente ...

Ramassage des processus

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

Principes

- Détection d'une clique de processus en attente ...
- ... sur des canaux connus uniquement de la clique

Ramassage des processus

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\begin{array}{l}\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \\ \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j\end{array}} \ (\mathit{stuck})$$

Principes

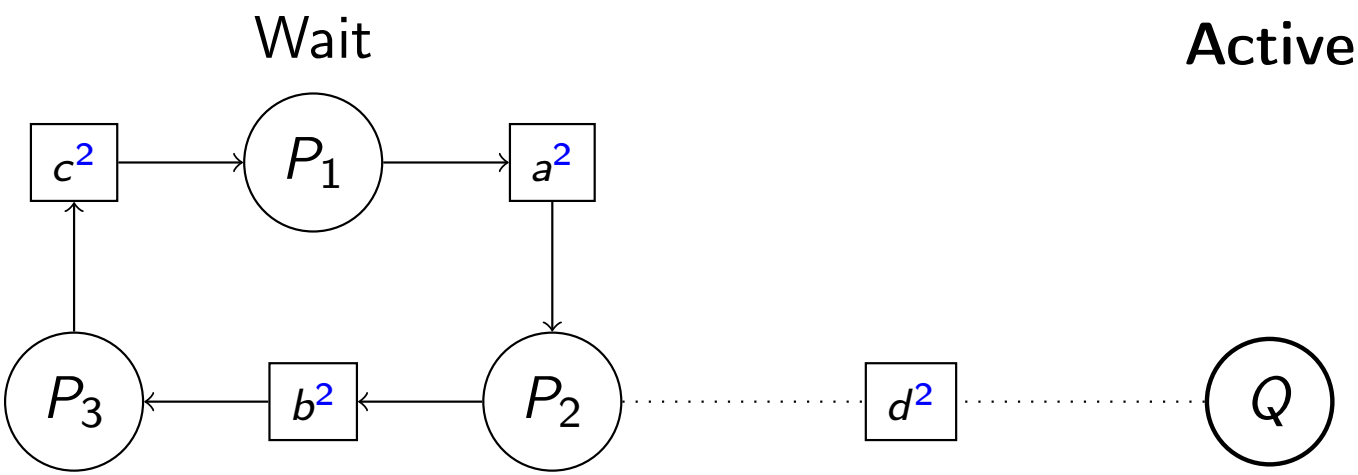
- Détection d'une clique de processus en attente ...
- ... sur des canaux connus uniquement de la clique

Remarque : pas de détection explicite de cycle

Ramassage des processus : cas positif

Règle

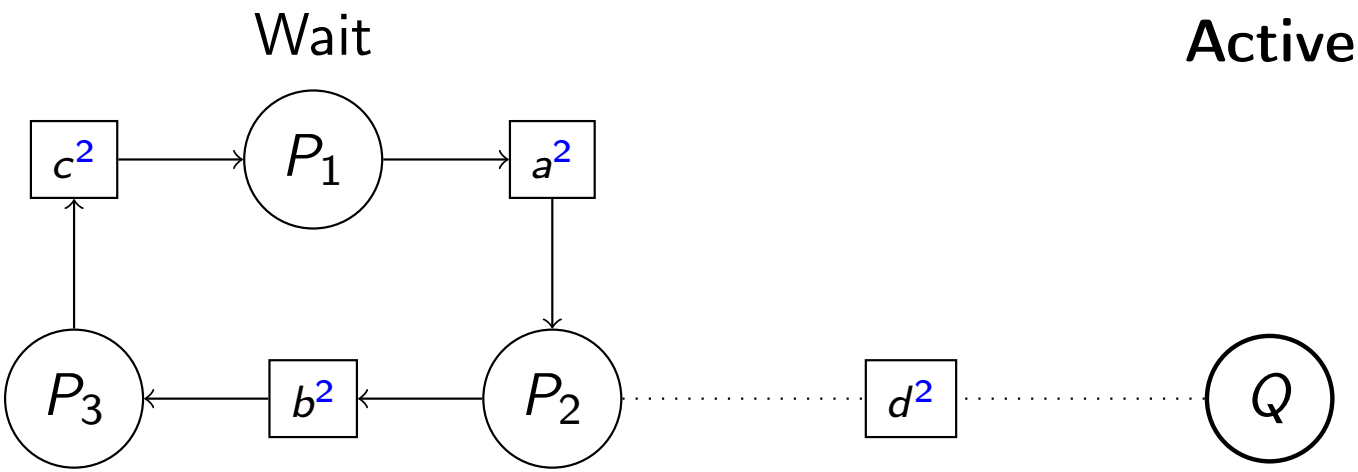
$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$



Ramassage des processus : cas positif

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

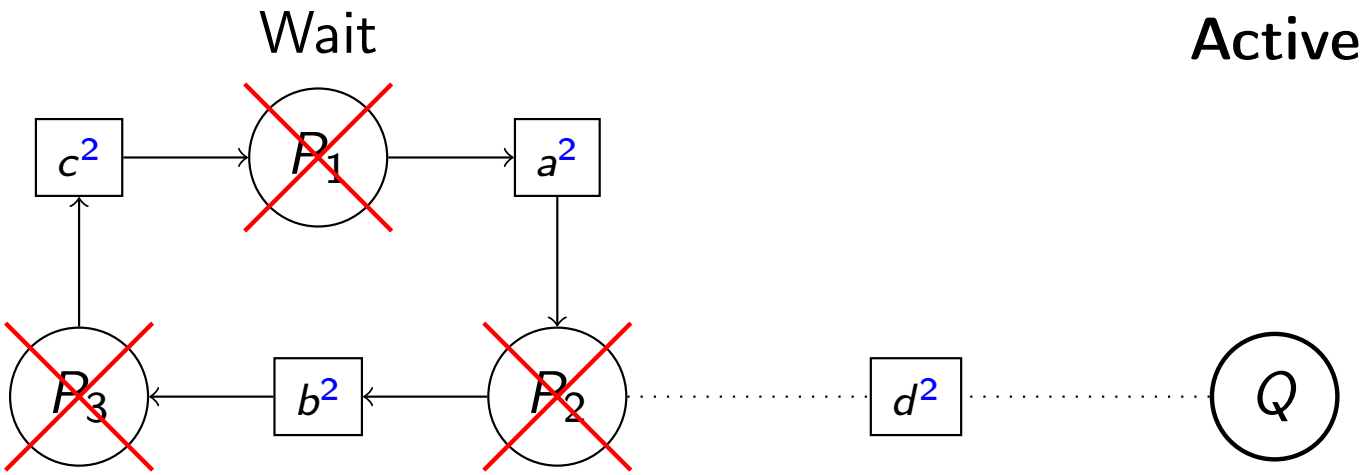


Ramassage des processus : cas positif

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$

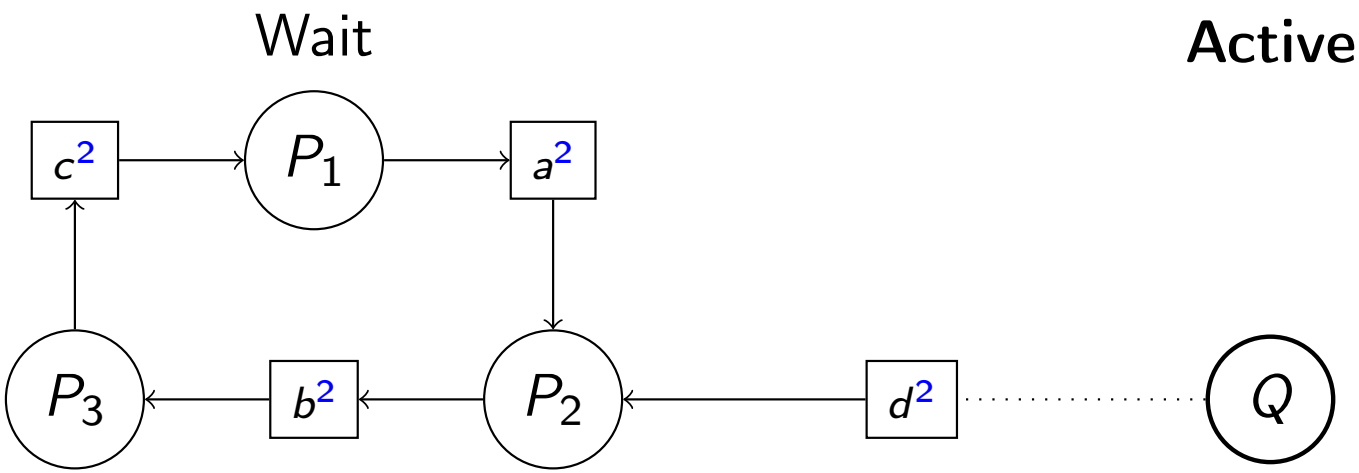
$$\rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j$$



Ramassage des processus : cas négatif

Règle

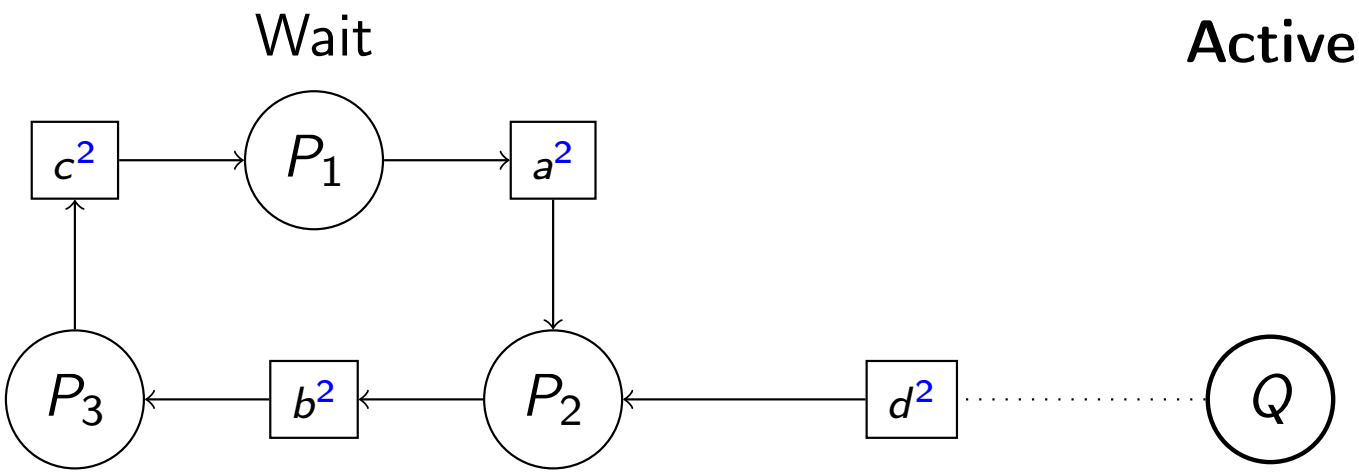
$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$



Ramassage des processus : cas négatif

Règle

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) \neq 0}{\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j} \ (\mathit{stuck})$$



Résumé des règles

$$\mathrm{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \mathrm{dom}(\delta),\ \delta(x) = \hat{c}$$

$$\mathrm{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if } \mathrm{knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \to \Delta \vdash A}\ (\textit{reclaim})$$

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\begin{array}{l}\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \\ \to \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j\end{array}}\ (\textit{stuck})$$

Plan

Implémentations

Disponibles

- CubeVM : interprète *stackless* du papier VEE06
- LuaPi : bib. au dessus des coroutines Lua (invité PUC-Rio été/hiver 2007, hiver/été 2008)
  - Engagements explicites, Ordonnanceur O(1)
    + extensions : broadcast, join patterns
- JavaPi : bib. au dessus des Java Threads
  - Adapté au multi-cœurs, algorithmes *lock-free*, détecteur de terminaison (GC simplifié)
- Picc : compilateur natif
  - Ref. globales uniquement, Nouveau GC (en cours)

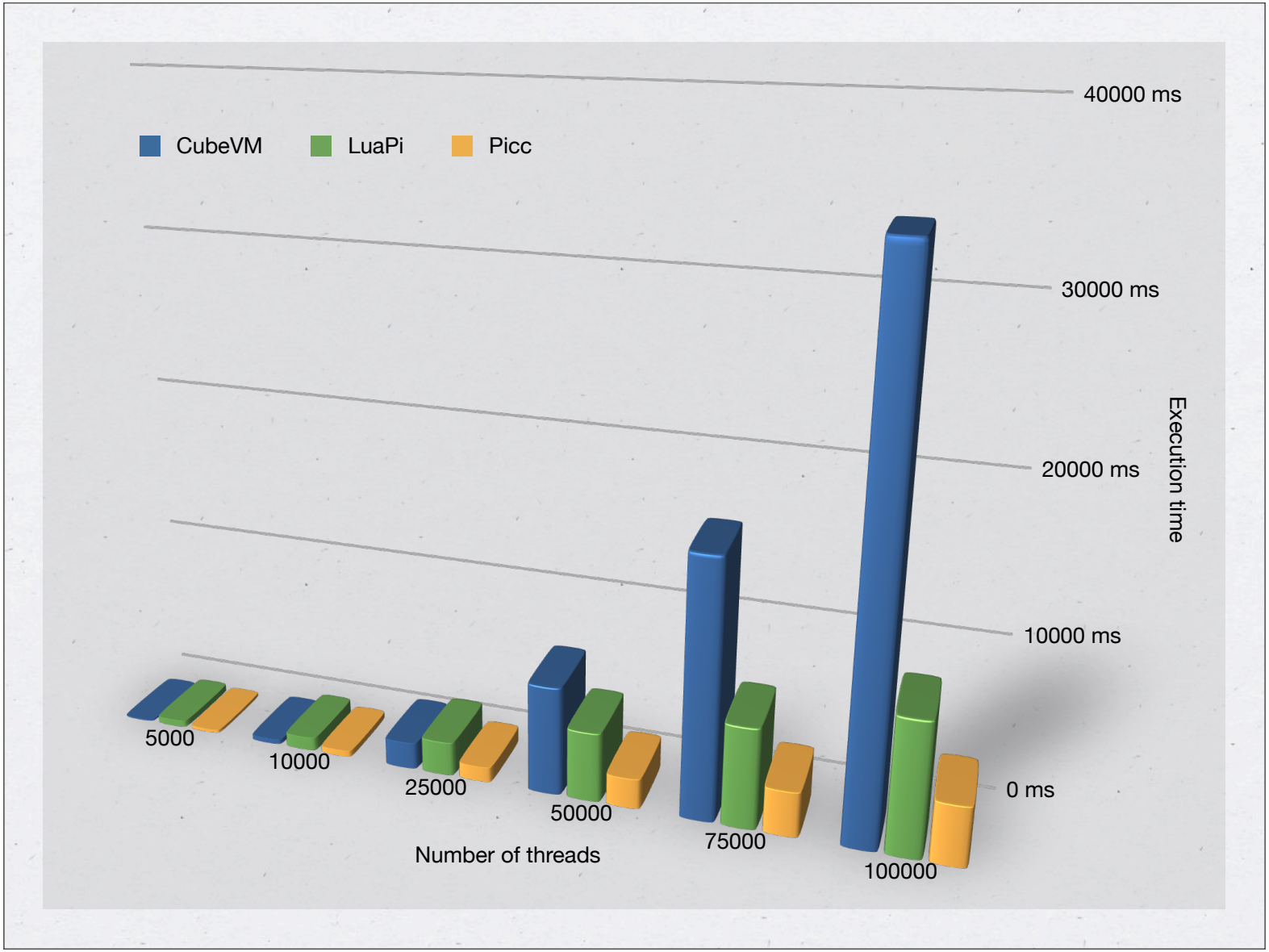
Implémentations

Disponibles

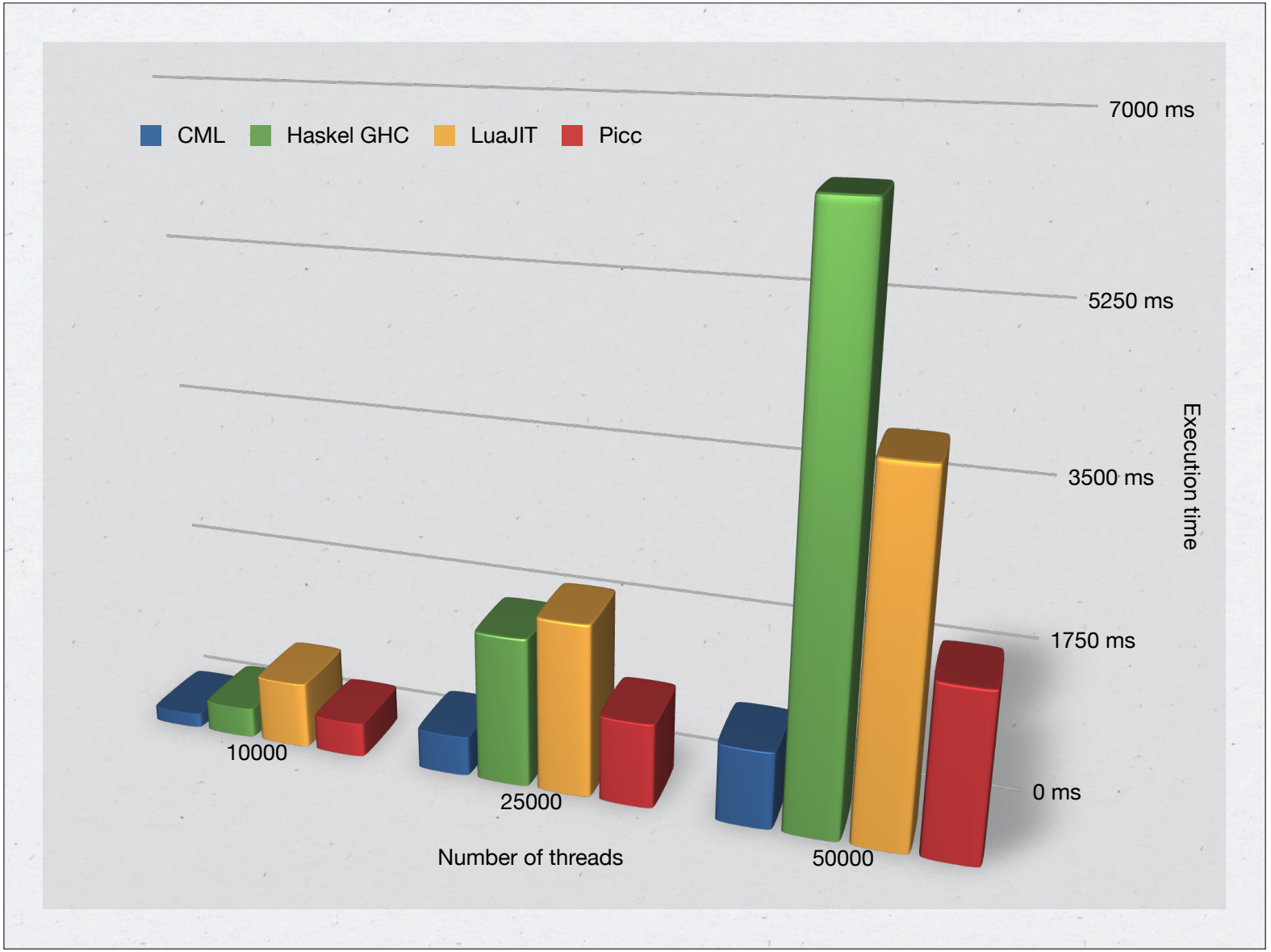
- CubeVM : interprète *stackless* du papier VEE06
- LuaPi : bib. au dessus des coroutines Lua (invité PUC-Rio été/hiver 2007, hiver/été 2008)
  - Engagements explicites, Ordonnanceur O(1)
    + extensions : broadcast, join patterns
- JavaPi : bib. au dessus des Java Threads
  - Adapté au multi-cœurs, algorithmes *lock-free*, détecteur de terminaison (GC simplifié)
- Picc : compilateur natif
  - Ref. globales uniquement, Nouveau GC (en cours)

+ d'infos sur `https://www-poleia.lip6.fr/~pesch/pithreads`

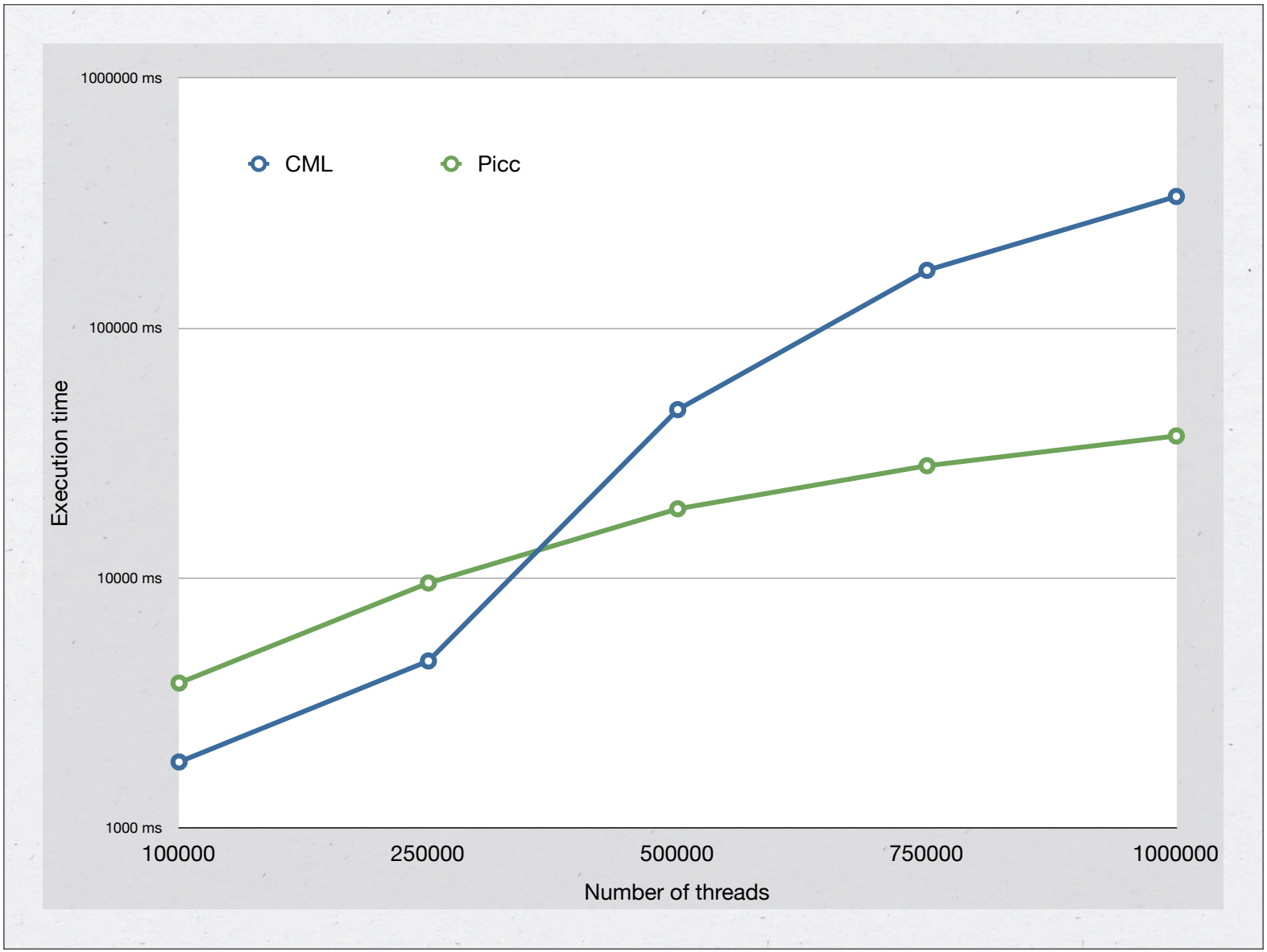
Benchmark 1 : implémentations des Pi-Threads



Benchmark 2 : Langages concurrents



Benchmark 3 : Passage à l'échelle



Travaux en cours

Compilateur : système de type v2

- Processus comme types (cf. Kobayashi)
- Approximation des recursions (interpretation abstraite)
- Injection automatique de pile : inference de canaux linéaires (cf. Sangiorgi)

Travaux en cours

Compilateur : système de type v2

- Processus comme types (cf. Kobayashi)
- Approximation des recursions (interpretation abstraite)
- Injection automatique de pile : inference de canaux linéaires (cf. Sangiorgi)

Runtime

- Backend multicœurs pour Picc + auto-threading
- Références locales+globales $\Longrightarrow$ globales uniquement
- GC de seconde génération $\Longrightarrow$ parallélisation

Travaux en cours

Compilateur : système de type v2

- Processus comme types (cf. Kobayashi)
- Approximation des recursions (interpretation abstraite)
- Injection automatique de pile : inference de canaux linéaires (cf. Sangiorgi)

Runtime

- Backend multicœurs pour Picc + auto-threading
- Références locales+globales $\Longrightarrow$ globales uniquement
- GC de seconde génération $\Longrightarrow$ parallélisation

$\Rightarrow$ Questions ?