INF5171 Programmation concurrente et parallèle : Introduction

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay

5 septembre 2017

Aperçu

1. Paradigme et paradigme de programmation

La notion de paradigme

Paradigme = Modèle théorique de pensée qui oriente la recherche et la réflexion scientifique.

Source: «Le Petit Larousse», 1997

La notion de paradigme

Un **paradigme** est une représentation du monde, une manière de voir les choses, un modèle cohérent de vision du monde qui repose sur une base définie (matrice disciplinaire, modèle théorique ou courant de pensée).

Le mot paradigme tire son origine du grec ancien *paradeigma*, qui signifie «modèle» ou «exemple».

Source: F. Goetghebeur, 2008

Paradigme de programmation

Programming paradigms *are heuristics used for algorithmic problem solving.*

Paradigme de programmation

Programming paradigms *are heuristics used for algorithmic problem solving.*

A **programming paradigm** *formulates a solution for a given problem by breaking the solution down to specific building blocks and defining relationship among them.*

Source: Stolin & Hazzan, 2007

Paradigme de programmation

Programming paradigms *are heuristics used for algorithmic problem solving.*

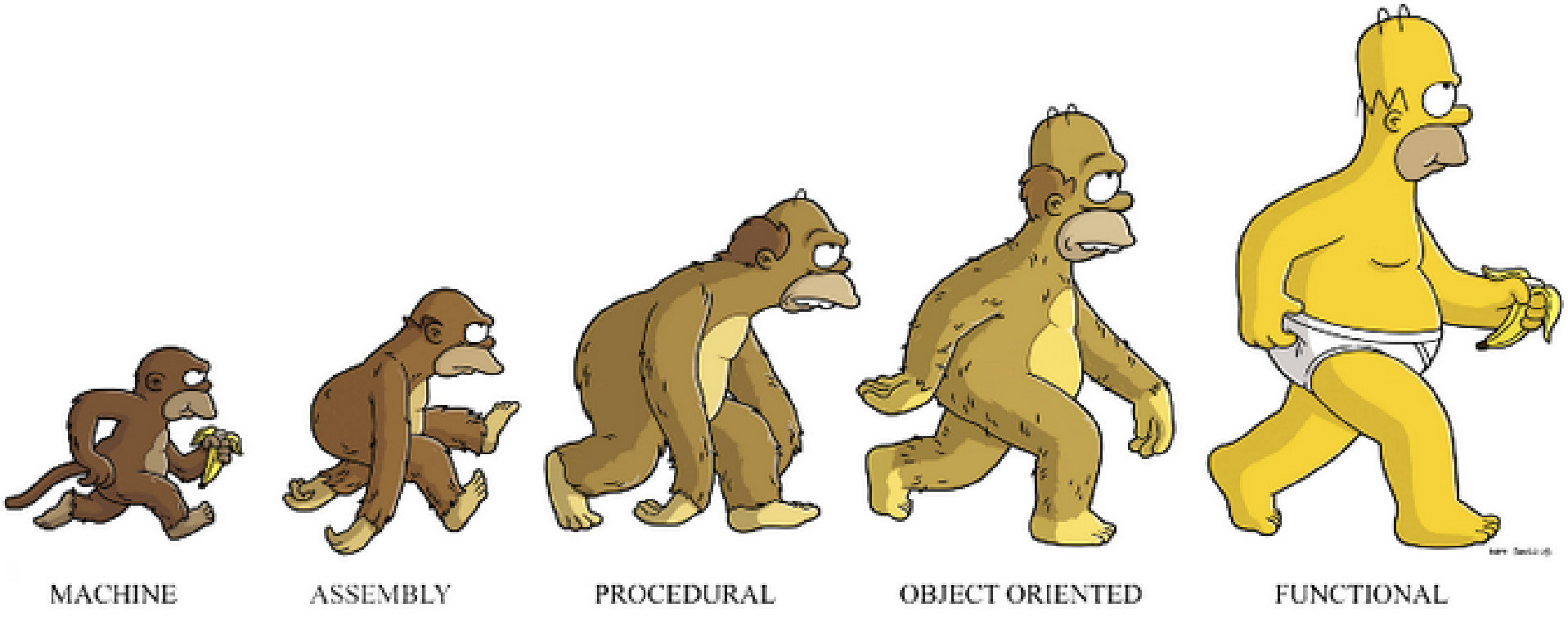
A **programming paradigm** *formulates a solution for a given problem by breaking the solution down to specific building blocks and defining relationship among them.*

Source: Stolin & Hazzan, 2007

Paradigme de programmation

Paradigme de programmation $\approx$ Façon d'aborder un problème de programmation, à l'aide de langages qui supportent bien **certains mécanismes d'abstraction et de modularité**

Les principaux paradigmes de programmation et langages étudiés au bac. à l'UQAM



Les principaux paradigmes de programmation et langages étudiés au bac. à l'UQAM

Impératif		**Déclaratif**	
Procédural	**Objet**	**Fonctionnel**	**Autre**
FORTRAN	Simula	Lisp/Scheme	**SQL**
Algol	Smalltalk	ML	**Prolog**
Pascal	**C++**	Miranda	
C	**Java**	**Haskell**	
bash	Ruby	Elixir	

Paradigme procédural	$\Rightarrow$	procédures sous-routines
Paradigme objet	$\Rightarrow$	objets classes
Paradigme fonctionnel	$\Rightarrow$	valeurs fonctions

Le paradigme comme «façon de voir le monde»

«Si le marteau est le seul outil que vous avez, vous voyez des clous partout !»

Le paradigme comme «façon de voir le monde»

«Si le marteau est le seul outil que vous avez, vous voyez des clous partout !»

Le paradigme comme «façon de voir le monde»

Autres variantes :

- *«I call it the law of the instrument, and it may be formulated as follows : Give a small boy a hammer, and he will find that everything he encounters needs pounding.»* (Kaplan, 1964)
- *«I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.»* (Maslow, 1966)
- *«If all you have is a hammer. . . eventually you'll hit a nail !»*

Changer de paradigme $\Rightarrow$ Utiliser d'autres outils

2. Deux exemples illustrant l'idée de paradigme. . . implicite

2.1 Exemple : La recherche du maximum

Le paradigme dans lequel on travaille est généralement **implicite**

Problème : On veut trouver la valeur maximum parmi une séquence (un tableau) d'éléments

Le paradigme dans lequel on travaille est généralement **implicite**

Problème : On veut trouver la valeur maximum parmi une séquence (un tableau) d'éléments

Dans ce qui suit, on va regarder différentes solutions. . .

Solution impérative itérative — Pseudocode

```
FONCTION maximum( s: array{int}; i, j: nat ): int
# i ≤ j && 0 ≤ i, j < size(s)
DEBUT
    m ← s[i]
    POUR k ← i+1 A j FAIRE
        m ← max( m, s[k] )
    FIN

    RETOURNER m
FIN
```

Solution impérative itérative — Pseudocode

```
FONCTION maximum( s: array{int}; i, j: nat ): int
# i ≤ j && 0 ≤ i, j < size(s)
DEBUT
    m ← s[i]
    POUR k ← i+1 A j FAIRE
        m ← max( m, s[k] )
    FIN

    RETOURNER m
FIN
```

Question : Quel est le temps d'exécution **asymptotique** pour une séquence `s` de longueur n ?

Solution impérative itérative — Pseudocode

```
FONCTION maximum( s: array{int}; i, j: nat ): int
# i ≤ j && 0 ≤ i, j < size(s)
DEBUT
    m ← s[i]
    POUR k ← i+1 A j FAIRE
        m ← max( m, s[k] )
    FIN

    RETOURNER m
FIN
```

Question : Quel est le temps d'exécution **asymptotique** pour une séquence `s` de longueur n ?

Question : Est-il possible de faire mieux ? Si oui, comment ? (Pour simplifier, supposons $n = 2^k$.)

Solution impérative récursive — Pseudocode

```
FONCTION maximum( s: array{int}; i, j: nat ): int
DEBUT
  SI i = j ALORS
    RETOURNER s[i]
  SINON
    m ← (i+j) / 2
    max1 ← maximum( s, i,   m )
    max2 ← maximum( s, m+1, j )

    RETOURNER max( max1, max2 )
  FIN
FIN
```

Solution impérative récursive — Pseudocode

```
FONCTION maximum( s: array{int}; i, j: nat ): int
DEBUT
  SI i = j ALORS
    RETOURNER s[i]
  SINON
    m ← (i+j) / 2
    max1 ← maximum( s, i,   m )
    max2 ← maximum( s, m+1, j )

    RETOURNER max( max1, max2 )
  FIN
FIN
```

Question : Quel est le temps d'exécution **asymptotique** pour une séquence `s` de longueur n ($n = 2^k$) ?

Solution impérative récursive — Pseudocode

```
FONCTION maximum( s: array{int}; i, j: nat ): int
DEBUT
  SI i = j ALORS
    RETOURNER s[i]
  SINON
    m ← (i+j) / 2
    max1 ← maximum( s, i,   m )
    max2 ← maximum( s, m+1, j )

    RETOURNER max( max1, max2 )
  FIN
FIN
```

Question : Quel est le temps d'exécution **asymptotique** pour une séquence `s` de longueur n ($n = 2^k$) ?
Question : Est-il possible de faire mieux ? Si oui, comment ?
(Pour simplifier, on suppose $n = 2^k$.)

Et si on sortait du **cadre habituel** où tout s'exécute de façon *séquentielle ?*

Solution parallèle — Pseudocode

Et si on sortait du **cadre habituel** où tout s'exécute de façon *séquentielle ?*

Solution parallèle — Pseudocode

Supposons que les appels de fonction se font en parallèle.

```
FONCTION maximum( s: array{int}; i, j: nat ): int
DEBUT
  SI i = j ALORS
    RETOURNER s[i]
  SINON
    m ← (i+j) / 2
    EN PARALLELE
       max1 ← maximum( s, i,   m )
       max2 ← maximum( s, m+1, j )
    FIN

    RETOURNER max( max1, max2 )
  FIN
FIN
```

Et si on sortait du **cadre habituel** où tout s'exécute de façon *séquentielle ?*

Solution parallèle — Pseudocode

Supposons que les appels de fonction se font en parallèle.

```
FONCTION maximum( s: array{int}; i, j: nat ): int
DEBUT
  SI i = j ALORS
    RETOURNER s[i]
  SINON
    m ← (i+j) / 2
    EN PARALLELE
       max1 ← maximum( s, i,   m )
       max2 ← maximum( s, m+1, j )
    FIN

    RETOURNER max( max1, max2 )
  FIN
FIN
```

Question : Quel est le temps d'exécution **asymptotique** pour une séquence `s` de longueur n ($n = 2^k$) ?

Solution parallèle récursive — MPD

```
procedure maximum( int s[*], int i, int j ) returns int
{
  if (i == j) {
    leMax = s[i];
  } else {
    m = (i+j) / 2;
    int max1, max2;
    co  max1 = maximum( s, i,   m );
    //  max2 = maximum( s, m+1, j );
    oc
    leMax = max( max1, max2 );
  }
}
```

Solution parallèle récursive — Ruby/pruby

```ruby
def maximum( s, i, j )
  if i == j
    s[i]
  else
    m = (i+j) / 2
    max1 = max2 = nil
    PRuby.pcall(
      -> { max1 = maximum( s, i,   m ) },
      -> { max2 = maximum( s, m+1, j ) }
    )

    [max1, max2].max
  end
end
```

Solution parallèle récursive — Ruby/pruby (bis)

```ruby
def maximum( s, i, j )
  if i == j
    s[i]
  else
    m = (i+j) / 2
    max1 = PRuby.future { maximum( s, i,   m ) }
    max2 = PRuby.future { maximum( s, m+1, j ) }

    [max1.value, max2.value].max
  end
end
```

Solution parallèle récursive — Ruby/pruby (bis)

```ruby
def maximum( s, i, j )
  if i == j
    s[i]
  else
    m = (i+j) / 2
    max1 = PRuby.future { maximum( s, i,   m ) }
    max2 = PRuby.future { maximum( s, m+1, j ) }

    [max1.value, max2.value].max
  end
end
```

Question : Peut-on faire mieux ? Si oui, comment ?

Solution parallèle récursive — Ruby/pruby (bis)

```
def maximum( s, i, j )
  if i == j
    s[i]
  else
    m = (i+j) / 2
    max1 = PRuby.future { maximum( s, i,   m ) }
    max2 = PRuby.future { maximum( s, m+1, j ) }

    [max1.value, max2.value].max
  end
end
```

Question : Peut-on faire mieux ? Si oui, comment ?
Que fait le *thread* parent pendant que les enfants travaillent ?

Solution parallèle récursive — Ruby/pruby (ter)

```ruby
def maximum( s, i, j )
  if i == j
    s[i]
  else
    m = (i+j) / 2
    max1 = PRuby.future { maximum( s, i,   m ) }
    max2 = maximum( s, m+1, j )

    [max1.value, max2].max
  end
end
```

Autre solution «diviser-pour-régner», mais sans récursion — Ruby/pruby

On suppose qu'on cherche le maximum pour `s` au complet

```ruby
def maximum( s )
  # s.size doit etre une puissance de 2
  n = s.size
  s = s.clone

  for i in 0..(Math.log2 n)-1
    dist = 2**i
    (dist-1...n).step(2*dist).to_a.peach do |j|
      s[j+dist] = [s[j], s[j+dist]].max
    end
  end

  s.last
end
```

Solution parallèle avec exactement NB_THREADS *threads* — Ruby

On suppose qu'on cherche le maximum pour s au complet

```ruby
def maximum( s )
  m = Array.new( NB_THREADS )
  n = s.size

  threads = []
  (0...NB_THREADS).each do |k|
    threads << Thread.new do
      ma_tranche = s[inf(k, n)..sup(k, n)]
      m[k] = ma_tranche.max
    end
  end
  threads.map(&:join)

  m.max
end
```

Solution parallèle avec exactement `NB_THREADS` *threads* (bis) — Ruby/pruby

On suppose qu'on cherche le maximum pour `s` au complet

```
def maximum( s )
  m = Array.new( NB_THREADS )
  n = s.size

  PRuby.pcall( 0...NB_THREADS,
    ->( i ) { m[i] = s[inf(i, n)..sup(i, n)].max }
  )

  m.max
end
```

Solution avec un nombre limité de *threads* — Ruby/pruby

On suppose qu'on cherche le maximum pour `s` au complet

```
def maximum( s )
  s.preduce(s[0]) { |m, v| [m, v].max }
end
```

2.2 Exemple : Le problème de Jackson

Le problème de Jackson de transformation d'un fichier de caractères

Soit N un entier positif.

- Entrée = série de lignes contenant un nombre variable de caractères
- Sortie = les lignes d'entrée, mais formatées pour avoir exactement N caractères avec les « $\star\star$ » remplacés par « $\hat{}$ »

Le problème de Jackson de transformation d'un fichier de caractères

Soit N un entier positif.

- Entrée = série de lignes contenant un nombre variable de caractères
- Sortie = les lignes d'entrée, mais formatées pour avoir exactement N caractères avec les « $\star\star$ » remplacés par « ^ »

Plus précisément, on veut...

1. paqueter/dépaqueter les caractères de façon à produire des lignes de N caractères — sauf (peut-être ?) pour la dernière ligne
2. que les paires « $\star\star$ » soient remplacées par « ^ »

Exemple de fichier d'entrée et de fichier de sortie

Exemple pour `n=4`

```
Entree:
-------
abc ** dsds cssa
ssdsx
fssfdfdfdfdfdf
s.s.**xtx*zy
```

```
Sortie:
-------
abc
^ ds
ds c
ssas
sdsx
fssf
dfdf
dfdf
dfs.
s.^x
tx*z
y
```

Solutions

Solution séquentielle

Très (!) difficile 

Solutions

Solution séquentielle

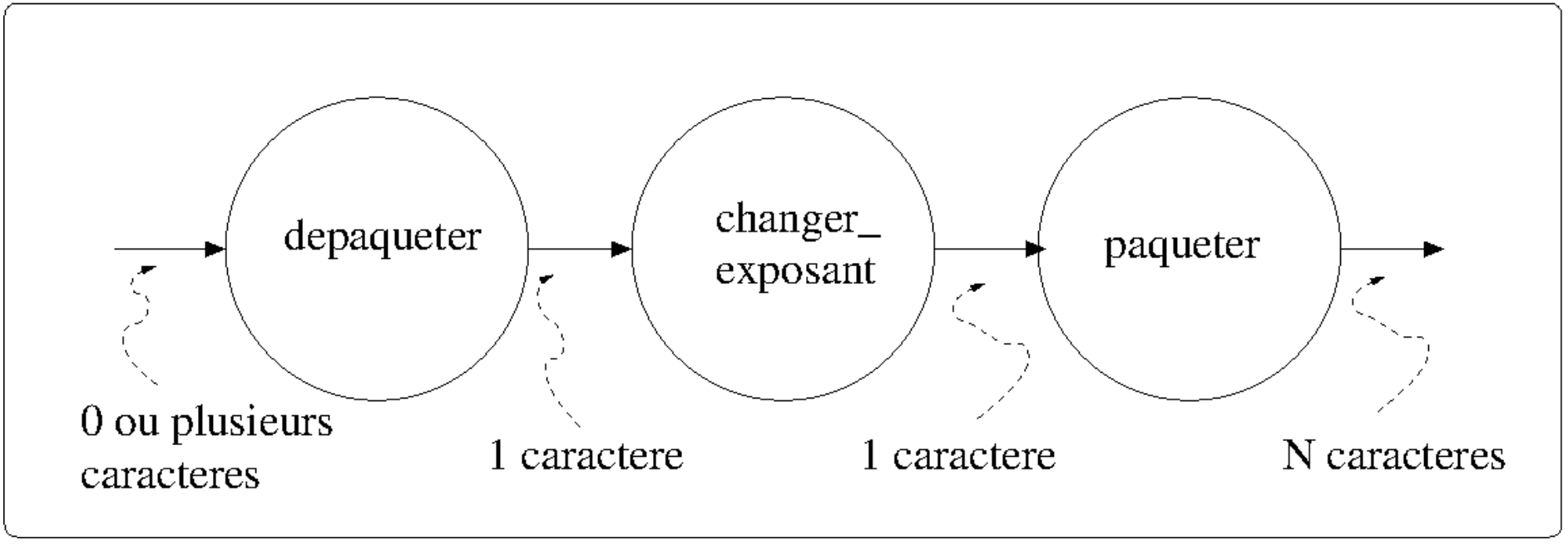
Très (!) difficile 

Solution concurrente

Facile 

La topologie des processus d'une solution concurrente avec filtres et pipelines

Pour simplifier, on ignore la lecture/écriture des fichiers



Solution avec parallélisme de flux, pipelines et canaux — Ruby/pruby

Les processus pour `depaqueter` et `paqueter`

```
depaqueter = lambda do |cin, cout|
  cin.each do |ligne|
    ligne.each_char { |c| cout << c }
  end
  cout.close
end

paqueter = lambda do |cin, cout|
  ligne = ''
  cin.each do |c|
    ligne << c
    (cout << ligne; ligne = '') if ligne.size == N
  end
  cout << ligne unless ligne.empty?
  cout.close
end
```

Solution avec parallélisme de flux, pipelines et canaux — Ruby/pruby

Le processus pour `changer_exposant` et le programme principal

```
changer_exposant = lambda do |cin, cout|
  cin.each do |c|
    (c = '^'; cin.get) if c == '*' && cin.peek == '*'
    cout << c
  end
  cout.close
end


# Activation des processus en pipeline!
depaqueter | changer_exposant | paqueter
```

Question

Question : Pourquoi parle-t-on ici d'une solution «concurrente» et non d'une solution «parallèle» ?

3. Qu’est-ce que la programmation concurrente et parallèle ?

Ce qu'est la programmation concurrente et parallèle

Programme séquentiel

= Programme défini par une *séquence* d'actions
$\Rightarrow$ une instruction après l'autre

= Un processus, une tâche, un *thread*

Note : *thread* = fil d'exécution

Ce qu'est la programmation concurrente et parallèle

Programme concurrent

= Programme qui contient plusieurs *threads* qui *coopèrent*

Coopération $\Rightarrow$ Communication, échange d'information

Ce qu'est la programmation concurrente et parallèle

Programme concurrent

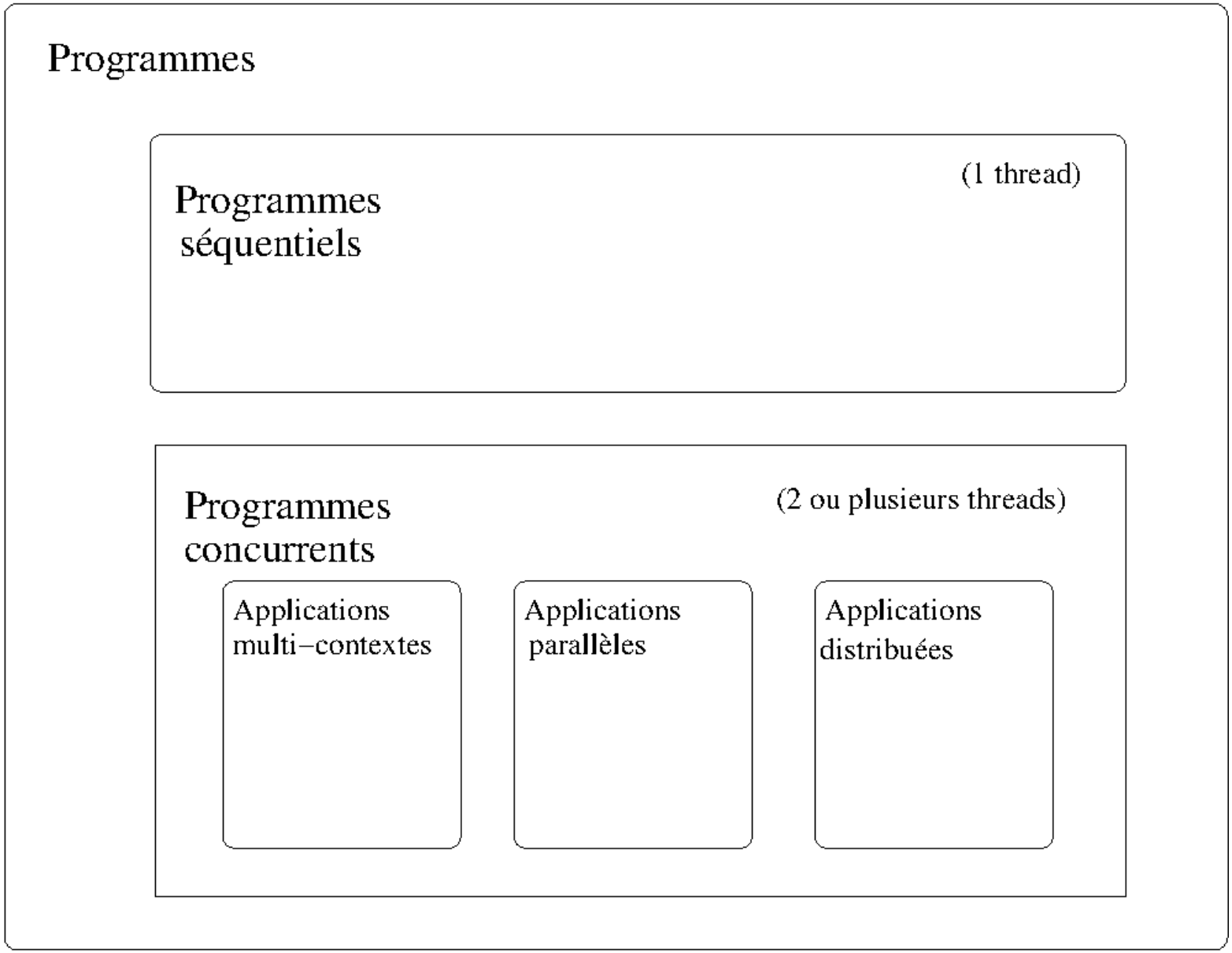
= Programme qui contient plusieurs *threads* qui *coopèrent*

Coopération $\Rightarrow$ Communication, échange d'information

Deux principales façons de coopérer/communiquer

- Par l'intermédiaire de variables *partagées*
- Par l'échange de messages

Différents types d'applications concurrentes



Application multi-contextes

Application multi-contextes (*multi-threaded*) $\Rightarrow$ contient plusieurs *threads*

Note : On considère généralement un *thread* comme étant un processus léger (*lightweight thread*)

Utilisations : Pour mieux structurer une application ($\Rightarrow$ meilleure modularité)

- Système d'exploitation multi-tâches
- Fureteurs multi-tâches
- Interface personnes-machines vs. application
- ...

Application parallèle

Application parallèle = chaque *thread* s'exécute sur son propre processeur

Utilisations : Pour résoudre plus rapidement un problème ou pour résoudre un problème plus gros

- Prévisions météorologiques
- Prospection minière
- Physique moderne
- Bio-informatique (génomique)
- ...

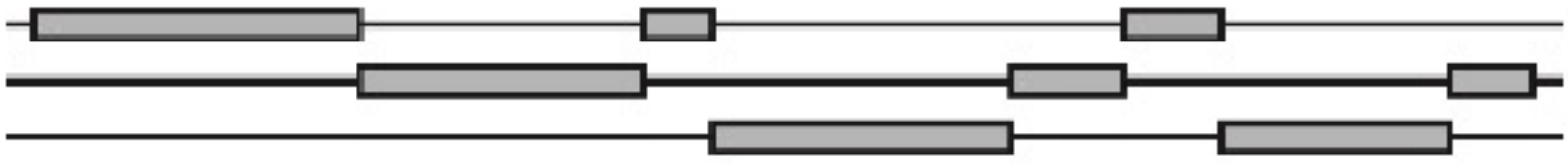
Application distribuée

Application distribuée = les processus communiquent entre eux par l'intermédiaire d'un réseau ($\Rightarrow$ délais plus longs)

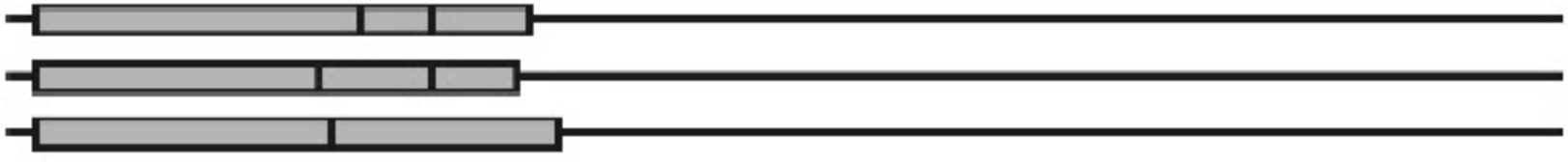
Utilisations :

- Serveurs de fichiers
- Accès à distance à des banques de données

Exécution concurrente vs. exécution parallèle



Concurrent, non-parallel execution



Concurrent, parallel execution

Pourquoi le parallélisme est-il de plus en plus important ?

Évolutions importantes au niveau matériel :

- Machine uniprocesseur : boîte contenant un seul processeur

- Multi-processeurs : boîte contenant plusieurs processeurs
- Multi-ordinateurs : plusieurs boîtes inter-connectées

- Processeurs multicoeurs
- Processeurs graphiques hautement parallèles (GPU)

Configuration de mon MacBook Air (modèle de base)

```
Device 0
Name: Intel(R) Core(TM) i5-4260U CPU @ 1.40GHz
Vendor: Intel
Compute Units: 4
Global Memory: 4294967296
Local Memory: 32768
Workgroup size: 1024

Device 1
Name: HD Graphics 5000
Vendor: Intel
Compute Units: 280
Global Memory: 1610612736
Local Memory: 65536
Workgroup size: 512
```

Le Tianhe 2

L'ordinateur le plus puissant au monde, classement de Novembre 2015

Le Tianhe 2

L'ordinateur le plus puissant au monde, classement de Novembre 2015

TIANHE-2 (MILKYWAY-2) - TH-IVB-FEP CLUSTER, INTEL XEON E5-2692 12C 2.200GHZ, TH EXPRESS-2, INTEL XEON PHI 31S1P

Site:	National Super Computer Center in Guangzhou
Manufacturer:	NUDT
Cores:	3,120,000
Linpack Performance (Rmax)	33,862.7 TFlop/s
Theoretical Peak (Rpeak)	54,902.4 TFlop/s
Nmax	9,960,000
Power:	17,808.00 kW
Memory:	1,024,000 GB
Processor:	Intel Xeon E5-2692v2 12C 2.2GHz
Interconnect:	TH Express-2
Operating System:	Kylin Linux
Compiler:	icc
Math Library:	Intel MKL-11.0.0
MPI:	MPICH2 with a customized GLEX channel

Le Sunway TaiHuLight

L'ordinateur le plus puissant au monde, classement de Novembre 2016

Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway

Site:	National Supercomputing Center in Wuxi
Manufacturer:	NRCPC
Cores:	10,649,600
Linpack Performance (Rmax)	93,014.6 TFlop/s
Theoretical Peak (Rpeak)	125,436 TFlop/s
Nmax	12,288,000
Power:	15,371.00 kW (Submitted)
Memory:	1,310,720 GB
Processor:	Sunway SW26010 260C 1.45GHz
Interconnect:	Sunway
Operating System:	Sunway RaiseOS 2.0.5

4. INF5171 Programmation concurrente et parallèle

Accent du cours

Étude des principaux paradigmes de **programmation concurrente** *et* **parallèle avec utilisation de variables partagées**

Préalables

- Unix/Linux, Java
- C + Java (langage impératif + langage objet)

Préalables

- Unix/Linux, Java
- C + Java (langage impératif + langage objet)

- Aimer programmer

Préalables

- Unix/Linux, Java
- C + Java (langage impératif + langage objet)

- Aimer programmer

- Aimer apprendre de nouveaux langages

Caractéristique du cours

Apprentissage et utilisation de plusieurs langages !

Caractéristique du cours

Apprentissage et utilisation de plusieurs langages !

The limits of my language mean the limits of my world.

L. von Wittgenstein

Contenu du cours I

Contenu du cours II

4. Programmation parallèle et concurrente avec **mémoire partagée** :
   - Concepts de base :
     processus vs. *thread* vs. tâche ; atomicité, exclusion mutuelle et synchr. cond. ; situation de compétition ; graphe de dépendances ;
   - Patrons de programmation parallèle en **PRuby** :
     parallélisme *fork–join*, de boucles, de données, coordonnateur/travailleurs, de flux de données ;
   - Patrons d'algorithmes parallèles :
     parallélisme de tâches, récursif, de données, de flux (filtres et pipelines, *streams*) ;

Contenu du cours III

- Programmation concurrente et mécanismes de coopération en **Ruby** :
  threads, verrous, moniteurs et variables de condition, queues ;

Contenu du cours IV

- Exemples de langages :
  Ruby/pruby
  Java
  C/OpenMP
  C++/Threading Building Blocks
  C/Pthreads
  C/OpenCL

Des nouveaux langages nous fournissent de nouveaux concepts, de nouveaux outils ⋆

A programmer can think quite well in just about any language. Many of us cut our teeth in BASIC, and simply learning how to think computationally allowed us to think differently than we did before. But then we learn a radically different or more powerful language, and suddenly we are able to think new thoughts, thoughts we didn't even conceive of in quite the same way before.

It's not that we need the new language in order to think, but when it comes along, it allows us to operate in different ways.
New concepts become new tools.

Source: `http://www.cs.uni.edu/~wallingf/blog/archives/monthly/2016-12.html#e2016-12-16T14_14_26.htm`

Contenu du cours IV

5. Mesures de performance : temps, coût, travail, accélération, efficacité ; loi d'Amdhal ;

Approche pédagogique

- Quelques cours du mardi se donneront en laboratoire (local à déterminer)

Approche pédagogique

- Quelques cours du mardi se donneront en laboratoire (local à déterminer)

- **INF5171 est un cours... de programmation !**

Évaluation

- Un intra (mardi 24 octobre, `13:30-16:30`) : 25 %
- Un final (mardi 12 décembre, `13:30-16:30`) : 30 %

- Trois (3) devoirs : 45 %

- Examens à «livre» ouvert
- Seuil de 50 % pour examens et devoirs

- Devoirs seul ou avec une (1) autre personnne
- Pénalité de retard : 10 % par jour

- Troisième devoir = Projet avec sujet et langage au choix (sujet à approbation)

Matériel pédagogique

- Notes de cours :
  `www.labunix.uqam.ca/~tremblay/INF5171`

Questions ?