Cours 3 : Parallélisme (Erlang)

March 12, 2024

Séance 10 (S+3) ; Cours N+3 : Parallélisme en Erlang

0) Introduction au parallélisme
1) Proposition Erlang
2) Processus
3) Communication
4) Exemples de programme

1 Introduction au parallélisme

1.1 Histoire (depuis les origines de l'informatique il y a du parallélisme !)

- circuits logiques (calculs tout le temps : en parallèles, synchrone ou asynchrone)
- Système temps partagé
- Pseudo-parallélisme (programmation concurrente)
- Algorithmique répartie ou distribuée (internet, synchrone ou asynchrone avec transaction ou algorithme/automate de poignée de main)
- Architecture multi-cœur (processeur et/ou carte graphiques), algorithmique collaborative

Parmi les langages récents :

- Rust (2006), peut faire du prallélisme, c'est l'un des objectifs principaux du langage, mais dans les livres et sur internet, cela n'apparait pas tout de suite ! (quand cela apparait, il faut parfois attendre les derniers chapitres). Et attention, Rust n'est pas un langage simple ! Pour le parallélisme, idem.
- Golang (2009), peut faire du prallélisme, c'est l'un des objectifs principaux du langage, mais dans les livres et sur internet, cela n'apparait pas tout de suite ! (quand cela apparait, il faut parfois attendre les derniers chapitres). Cela utilise des "goroutine" facile d'utilisation (la facilité d'utilisation est l'un des autres objectifs du langage) mais beaucoup de choses doivent être faites "à la main" (mutex, synchronisation, etc.)
- Kotlin (2011), peut faire du parallélisme, ce n'est pas l'objectif principal de Kotlin, cela utilise de la mémoire partagée et c'est à base de coroutine et channels (stream)

1.2 Taxonomie de Flynn (1966)

En matière d'ordinateur, Flynn a proposé en 1966 cette classification de différentes formes de parallélisme (cf. https://fr.wikipedia.org/wiki/Taxonomie_de_Flynn) :

- SISD : architecture de von Neumann (single thread)

- SIMD : un même programme (synchrone) pour plusieurs données, ex. : les processeurs vectoriels
- MISD : plusieurs programmes partageant une seule données (implémentations plutôt rare, ex. : traitement de données)
- MIMD : avec des variantes selon qu'il y a des mémoires partagées ou pas (modèles des machines actuelles)

1.3 Topologie

Et il ne faudrait pas réduire l'informatique au flux de contrôle des instructions (en ajoutant un opérateur de calcul en parallèle), l'informatique c'est aussi des données. Pour le parallélisme, il y a au niveau des données, de l'information, une question qui s'ajoute : la question de la **communication** entre processus.
Dans ce domaine, on peut réfléchir aux topologies de communication :

- DataFlow (ad'hoc pour un algo, orienté données)
- Ligne, Bus, Anneau, grille 2D, grille 3D, treillis, hypercube, complet
- Etoiles, flocon, Constellation
- Maître-esclave, Arbre, 3 tiers, N tiers, en couche

1.4 Surtout : le parallélisme c'est compliqué ! Problèmes et vocabulaire

L'algorithmique classique séquentielle ressemble assez à la façon de penser nos actions individuelles basée sur l'organisation selon une ligne de temps claire des actions à faire. La récursivité déstabilise déjà cette façon de penser d'une certaine manière (par emboîtement de différentes lignes de temps). Pour le parallélisme, il faut penser aux actions d'un groupe, voire d'une communauté. Il y a une complexité qui augmente encore et une intuition à mobiliser différente, peut-être moins riche, moins entraînée (à penser à la place des autres).

Pour autant, le parallélisme est complètement naturel et courant.

Il est donc permis d'espérer trouver des manières de programmer en parallèle.

En attendant, preuves de cette difficulté de la programmation parallèle, voici quelques exemples (et du vocabulaire pour en parler) :

- Race condition (situation de compétition, ou conditions de courses) : 2 processus exécutent le même programme simple (N<-N+1) sur la même donnée N initialement à 0, quel est le résultat finale ? N=1, N+2 ? Et s'il y en avait 10 processus ?
- Interblocage (deadlock) : 2 processus veulent accéder à 2 ressources (imprimante, données), chacun bloque la première ressource qu'il obtient et attend que l'autre se libère ! Il faudrait un algo d'allocation de ressources sûres (algo du banquier)
- Famine (starvation) : quand les interblocages et race condition ont été évités, par des algorithmes d'exclusions mutuelles (mutex, sections critiques (synchronized), verrou, sémaphore, moniteur ou des protocoles spécifiques) et une hiérarchie de droit (Maitre-esclave), alors certains processus peuvent s'en sortir et d'autres souffrir ou dépérir ou ne pas pouvoir travailler (c'est dommage). Ex. un producteur produit des données dans un buffer de taille N, mais N+1 processus cherchent à consommer ces données. En moyenne il y aura toujours un processus inactif, au pire c'est toujours le même, il ne faudrait pas que cela soit un processus spécifique, unique et essentiel.

- Au final, quand tous les problèmes (majeurs) sont réglés, on peut réfléchir encore aux algos et améliorer ou garantir :
  - équité
  - vivacité
  - sureté

Pour votre sagacité, quelques problèmes de la vie courante (?) sur le parallélisme :

- le repas des philosophes chinois (sur le partage de ressources, où l'aléatoire peut aider ! le parallélisme est un domaine, comme l'algorithmique réseau ou les programmes avec un peu d'aléatoire peuvent obtenir de meilleurs résultats que les algorithmes purement déterministes)
- le problème des horloges (comment (r)établir une chronologie d'évènements entre des lieux avec des horloges farfelues)
- le problème des généraux byzantins (comment décider du début des activités quand les communications sont incertaines)

1.5 Limites du parallélisme

Souvent, on utilise le parallélisme pour améliorer des algos séquentiels, cela a des limites :

- le nombre de ressources/processus disponibles (on ne pourra jamais aller 10 fois plus vite si on a seulement 2 processus disponibles)
- la part séquentielle des algos
- ce qui donne la loi de Amdahl : https://fr.wikipedia.org/wiki/Loi_d%27Amdahl (peu de processeur, peu de gain sur temps global, et limité par temps séquentiel) ou la loi de Gustafson https://fr.wikipedia.org/wiki/Loi_de_Gustafson (plus de données, plus de gain sur temps moyen par donnée)

2 Proposition Erlang

Pour essayer de résoudre par construction, en proposant des primitives dans le langage qui soient "bonnes" :

- Erlang ne permet pas le partage de mémoire/données (c'est cohérent avec la gestion des variables, l'instanciation unique, le passage de paramètre, la programmation fonctionnelle pure), les processus/fonctions ne peuvent s'échanger que des valeurs (constantes, connues)
- Les primitives Erlang ne bloquent pas ! Elles peuvent échouer, mais sans blocage. (c'est doit être cohérent avec la gestion des erreurs qui n'alourdit pas l'écriture des programmes avec une gestion des erreurs dans les algos, mais demande à ce qu'elle soit séparée)

2.1 Eléments de base de la proposition Erlang

Partie parallèle/collaborative, celle qui nous fait préférer Erlang à d'autres langages … :

- d'autres langages qui ne comportent pas de volet "parallélisme" ou "collaboratif" (ou alors ce n'est pas immédiat)
- d'autres langages qui comportent un volet "parallélisme" ou "collaboratif" mais pour lesquels, rien n'est simple (déjà !) et en plus pour lesquels le "collaboratif" est surtout source de nombreux pièges (famine, interbloccage, …) !

Il y a deux temps dans la programmation collaborative :

- pouvoir faire travailler en parallèle plusieurs processus,
- les faire communiquer.

2.2 Plusieurs processus en parallèle

Oubliez le fork C (c'est pourtant joli ; mais au départ, c'est un peu compliqué à prendre en main). Oubliez les pipes (pourtant c'est bien aussi ! et il y a des organisations modernes en stream qui ont repris le concept pour améliorer la fusion entre programmation classique séquentielle et parallèlisme).

En Erlang, c'est simple, on demande juste à faire naitre un processus pour exécuter une tâche, et une fois la demande effectée, celui qui a effectué la demande reprends le cours de son exécution normal (que le processus en parallèle ait réussi à se lancer et s'exécuter ou pas).

Le mot clé :

- spawn qui renvoie (une erreur ou) le nom du processus pour pouvoir communiquer avec lui

et pour utilisation pratique, le mot clé :

- self() (pour connaître et donner son propre nom de processus)

2.3 La communication

Pour la communication, c'est aussi très simple, il y a une primitive du langage pour envoyer un message, et une primitive pour recevoir. Entre l'envoi et la réception, c'est le système Erlang qui gère les messages (en particulier, avec des files d'attentes pour ne pas perdre de message ou rendre les communications bloquantes). C'est à l'image de la communication par courriel (ou sms).

Remarque, pour savoir à qui envoyer un message, il faut avoir repérer ses interlocuteurs avant, et pour que la personne puisse vous répondre, il vaut mieux aussi lui donner son nom. Typiquement, pour savoir qui on est il suffit de demander self(), et pour savoir à qui envoyer un message, soit on vous a donner son nom, soit vous avez créer ce processus par un spawn(), dans ce cas, il faut récupérer au passage son nom.

Les mots clés :

- ! (pour envoyer ProcessusAQuiOnEnvoie!MessageEnvoyé)
- receive … end (et aussi after)

3 Les Processus (détails et premiers programmes)

Prenons un exemple classique (sans création de processus) :

[79]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


hello(W) ->
```

```
  io:format("Bonjour ~p~n",[W]).

main([]) ->
  hello("L3Miage").
```

```
Overwriting prog.erl
```

[80]:
```
!escript prog.erl
```

```
Bonjour "L3Miage"
```

En ajoutant la création d'un processus :

[235]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


hello(W) ->
  io:format("Bonjour ~p~n",[W]).

main([]) ->
  spawn(prog, hello, ["L3Miage"]).
```

```
Overwriting prog.erl
```

[238]:
```
!escript prog.erl
```

On ne voit pas grande chose, ou des erreurs ?
Il y a 2 problèmes :

- Pour utiliser des processus (fonction indiquée dans un module), il faut compiler le module. Soit sur la ligne de commande (cf. commande ci-dessous), soit dans le code lui-même (code après la commande)
- Le programme principal va probablement se terminer en premier, et avec, le terminal e/s va être fermé, le processus fils affichera son résultat nulle part !

[239]:
```
!erlc prog.erl; escript prog.erl
```

mais cela ne change (presque) rien, à cause du second problème, sauf à ajouter un affichage tardif.

[240]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


hello(W) ->
  io:format("Bonjour ~p~n",[W]).
```

```
main([]) ->
  spawn(prog, hello, ["L3Miage"]),
  io:format(""),io:put_chars(<<>>).
```

```
Overwriting prog.erl
```

[243]:
```
!erlc prog.erl ; escript prog.erl
```

```
Bonjour "L3Miage"
```

Meilleure solution :

- avec compilation dans le code
- et petite attente en fin de programme principal

[244]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


hello(W) ->
  io:format("Bonjour ~p~n",[W]).

main([]) ->
  compile:file(prog),
  spawn(prog, hello, ["L3Miage"]),
  timer:sleep(1000).
```

```
Overwriting prog.erl
```

[245]:
```
!escript prog.erl
```

```
Bonjour "L3Miage"
```

Avec plusieurs processus :

[246]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


hello(W) ->
  io:format("Bonjour ~p~n",[W]).

main([]) ->
  compile:file(prog),
  spawn(prog, hello, ["L3Miage"]),
  spawn(prog, hello, ["M1mIAGE"]),
  spawn(prog, hello, ["m2EtBoulot"]),
```

```
  timer:sleep(1000).
```

```
Overwriting prog.erl
```

[251]:
```
!escript prog.erl
```

```
Bonjour "L3Miage"
Bonjour "M1mIAGE"
Bonjour "m2EtBoulot"
```

Pour simuler un peu d'aléatoire on peut ajouter un petit temps d'attente avant affichage ?

[299]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


hello(W) ->
  timer:sleep(round(timer:seconds(rand:uniform()))),
  io:format("Bonjour ~p~n",[W]).

main([]) ->
  compile:file(prog),
  spawn(prog, hello, ["L3Miage"]),
  spawn(prog, hello, ["M1mIAGE"]),
  spawn(prog, hello, ["m2EtBoulot"]),
  timer:sleep(1100).
```

```
Overwriting prog.erl
```

[300]:
```
!escript prog.erl
```

```
Bonjour "m2EtBoulot"
Bonjour "L3Miage"
Bonjour "M1mIAGE"
```

Remarques :

- les affichages ne se mélangent pas complètement (heuresement), chaque ligne commencée est finie avant de passer à une autre.

- s'il y a plusieurs affichages, sur plusieurs lignes, avec un peu de temps aléatoire entre chaque affichage, même si cela appartient à la même section séquentielle ...

[274]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).
```

```
hello(W) ->
  timer:sleep(round(timer:seconds(rand:uniform()))),
  io:format("Bonjour ~p~n",[W]),
  timer:sleep(round(timer:seconds(rand:uniform()))),
  io:format("Au revoir ~p~n",[W]).

main([]) ->
  compile:file(prog),
  spawn(prog, hello, ["L3Miage"]),
  spawn(prog, hello, ["M1mIAGE"]),
  spawn(prog, hello, ["m2EtBoulot"]),
  timer:sleep(2100).io:format("Au revoir ~p~n",[W]).
```

```
Overwriting prog.erl
```

[275]:
```
!escript prog.erl
```

```
Bonjour "L3Miage"
Bonjour "M1mIAGE"
Au revoir "M1mIAGE"
Au revoir "L3Miage"
Bonjour "m2EtBoulot"
Au revoir "m2EtBoulot"
```

Remarques :

- la création de processus est rapide
- la fonction exécutée est définie par l'appelant
- les paramètres sont donnés par l'appelant
- spawn renvoie le nom du processus (sera utile pour communiquer ou pour tuer le processus)
- le nom du processus courant est donné par self()

4 La communication (détails et premiers programmes)

Pour communiquer il faut être 2 (ou plus). Les programmes à suivre auront donc tous une partie de générations de processus avant les communications proprement dites.

Pour communiquer il faut savoir à qui parler. Les programmes à suivre auront donc tous une partie qui permet aux processus tardifs de connatre au moins le nom des processus précoces (pour engager la discussion).
La problématique du carnet d'adresse (avoir des noms de processus avec qui communiquer) sera vu plus tard.

[292]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

ping(IdProc) ->
  IdProc ! "Bonjour, moi c'est ping".
```

```
pong() ->
  receive
    X ->
      io:format("Bonjour ~p~n",[X]) end.

main([]) ->
  compile:file(prog),
  IdPong = spawn(prog, pong, []),
  _IdPing = spawn(prog, ping, [IdPong]),
  timer:sleep(100).
```

```
Overwriting prog.erl
```

[293]:
```
!escript prog.erl
```

```
Bonjour "Bonjour, moi c'est ping"
```

Pour que pong puisse répondre, il faudrait que ping lui ait donner son vrai nom (ou que le programme principal ait donner tous les noms à tout le monde, mais c'est une autre histoire.)

[291]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

ping(IdProc) ->
  IdProc ! ["Bonjour, moi c'est ",self()].

pong() ->
  receive
    [_Msg,IdProc] ->
      io:format("pong> Bonjour ~p~n",[IdProc]) end.

main([]) ->
  compile:file(prog),
  IdPong = spawn(prog, pong, []),
  _IdPing = spawn(prog, ping, [IdPong]),
  timer:sleep(100).
```

```
Overwriting prog.erl
```

[294]:
```
!escript prog.erl
```

```
Bonjour "Bonjour, moi c'est ping"
```

Maintenant, ping et pong peuvent communiquer :

[298]:
```
%%writefile prog.erl
-module(prog).
```

```
-compile([export_all,nowarn_export_all]).

ping(IdProc) ->
  IdProc ! ["Bonjour, moi c'est ",self()],
  receive _Msg ->
    io:format("ping> cool!~n",[]) end.

pong() ->
  receive [_Msg,IdProc] ->
    io:format("pong> Bonjour ~p~n",[IdProc]),
    IdProc !  "Comment ca va ?" end.

main([]) ->
  compile:file(prog),
  IdPong = spawn(prog, pong, []),
  _IdPing = spawn(prog, ping, [IdPong]),
  timer:sleep(100).
```

```
Overwriting prog.erl
```

[297]:
```
!escript prog.erl
```

```
pong> Bonjour <0.83.0>
ping> cool!
```

Remarques :

- receive peut être bloquant s'il n'y a pas de message reçu ou à recevoir (sauf si ajout d'un timeout)
- après receive, la syntaxe a des ressemblance avec la définition d'une nouvelle fonction :
  - on peut avoir plusieurs cas (raisonnement par cas)
  - chaque cas peut recevoir des gardes
  - on peut ajouter des cas spéciaux avec timeout (after 100 ->)
- la file des messages peut être vidée (si nécessaire)
- receive ne consomme qu'un seul message (s'il y en a un disponible et remplissant les conditions de garde, éventuellement en attendant un peu qu'un tel message arrive), pour consommer plusieurs messages il faut faire/prévoir plusieurs receive, c'est possible en bouclant (récursivement) sur l'appel du receive.

Exemple où pong serait un automate qui calcul des valeurs absolues :

[319]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

ping(IdProc) ->
  {ok, X} = io:read("Donne un nombre : "),
  IdProc ! [X,self()],
  receive
```

```
    Msg -> io:format("ping> Resultat ~p~n",[Msg])
  end.

pong() ->
  receive
    [X,W] when X < 0 -> W!(-X);
    [X,W] -> W!X
    after 100 -> done end.

main([]) ->
  compile:file(prog),
  IdPong = spawn(prog, pong, []),
  _IdPing = spawn(prog, ping, [IdPong]),
  timer:sleep(100).
```

```
Overwriting prog.erl
```

[323]:
```
!echo "-42." | escript prog.erl
```

```
Donne un nombre : ping> Resultat 42
```

Pong est un processus “one shot”, on peut imaginer qu’il s’exécute en boucle (tant qu’il y a des demandes) : après envoie de la réponse, il peut récursivement être ré-appelé.