

Paradigme de concurrence

En utilisant Elixir

2025-04-03

Guillaume Dunant & Edwin Häffner

HEIG-VD

2025

ICSL

Table des matières

I.	Introduction	3
II.	Motivations du choix de la concurrence	3
III.	Paradigme de concurrence	3
IV.	Pourquoi Elixir est la solution ?	3
	Qu'est ce que c'est BEAM ?	3
	Gestion de la concurrence	4
	Planificateurs multicœurs efficaces	4
	Tolérance aux pannes grâce à la gestion des erreurs	4
	Collecte de déchets par processus	4
	Paradigme fonctionnel	5
	Processus légers	5
V.	La concurrence dans Elixir	5
	Les outils de base	5
VI.	Cahier des charges prévisionnel	5
VII.	Recherches effectuées	5
	13-03-25	5
	27-03-25	5
VIII.	Bibliographie	5
	BEAM	5
IX.	Feature	6
	Customized items	6

I. Introduction

L'idée derrière le paradigme de concurrence est de pouvoir efficacement gérer plusieurs tâches en parallèle sur une même machine, ou même sur plusieurs. L'objectif final est d'optimiser l'utilisation des ressources disponibles, de réduire les temps d'attente et d'accélérer l'exécution des programmes en répartissant les calculs ou les opérations entre plusieurs processus ou threads qui s'exécutent simultanément.

Un exemple simple serait d'avoir une application qui utilise un thread du CPU pour afficher la partie GUI (interface graphique) et un autre thread pour effectuer des calculs en arrière-plan. Cela éviterait de bloquer l'affichage graphique lorsqu'un calcul complexe est demandé, améliorant ainsi la fluidité et la réactivité de l'application.

II. Motivations du choix de la concurrence

La concurrence est devenue essentielle dans le développement de logiciels modernes. Les CPU actuels disposent d'un nombre croissant de cœurs, même dans le marché des processeurs abordables, par exemple, un processeur à moins de 100 francs offre déjà au moins 6 cœurs.

Donc il nous faut pouvoir utiliser ces cœurs maintenant si abondant pour pouvoir faire des applications efficace.

La concurrence est aussi très importante lorsqu'on interagit avec des API web et d'autres machines. La communication n'est pour l'instant pas instantanée et il faut alors avoir un mécanisme d'attente qui ne fait pas arrêter le programme. Les opérations d'entrée/sortie (I/O), comme les requêtes réseau ou les accès disque, sont particulièrement concernées car elles comportent des temps d'attente réel.

III. Paradigme de concurrence

La programmation concurrente est donc une façon de programmer en tenant compte de l'existence de ces threads et processus. Elle implique la conception de programmes où plusieurs séquences d'instructions peuvent s'exécuter simultanément ou en alternance rapide (préemption de plusieurs processus entre eux par exemple).

Ce paradigme introduit des concepts spécifiques comme la synchronisation, les verrous, les sémaphores, les variables atomiques, et les files de messages, qui permettent de coordonner l'exécution des différents processus ou threads et d'éviter les problèmes classiques de concurrence comme les race conditions, les deadlocks ou les famines.

IV. Pourquoi Elixir est la solution ?

Elixir tourne sur la machine virtuelle d'Erlang qui est nommée BEAM. Plus précisément le code Elixir est compilé dans un bytecode qui tourne sur BEAM.

Qu'est ce que c'est BEAM ?

BEAM, qui veut dire en français "La machine abstraite Erlang de Bogdan" est la machine virtuelle qui exécute le bytecode compilé des programmes Erlang et Elixir.

Vu que cette machine virtuelle a été développée pour l'écosystème Erlang/OTP (Open Telecom Platform), elle est spécialement conçue pour répondre aux exigences liées à la télécommunication, la haute disponibilité, la tolérance aux pannes et la concurrence.

Voici ses caractéristiques :

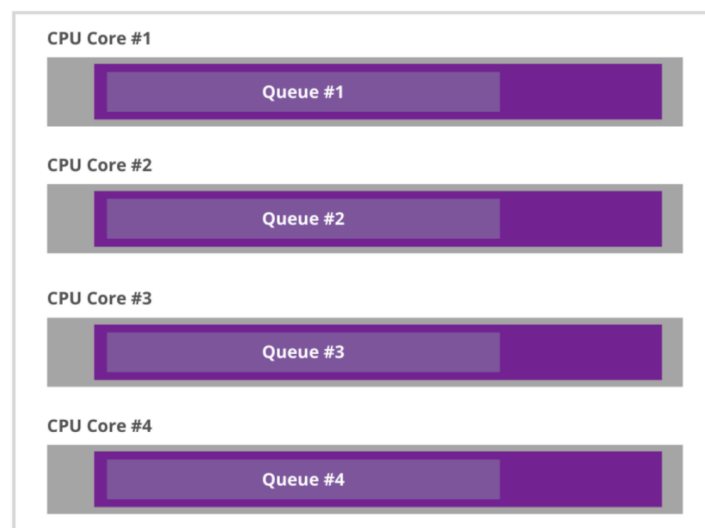
Gestion de la concurrence

BEAM exécute des processus légers qui ne partagent pas de mémoire. Ces processus communiquent uniquement via un passage de **messages asynchrone**, ce qui permet d'éviter les problèmes courants liés aux accès concurrents comme les conditions de course et les verrous. Cette isolation entre les processus est la base de la tolérance aux pannes : si l'un des processus échoue, il n'impactera pas les autres.

Planificateurs multicœurs efficaces

Initialement, BEAM utilisait une seule file d'attente d'exécution (run queue). Aujourd'hui, elle attribue une file d'attente à chaque cœur de processeur disponible, ce qui permet de paralléliser les programmes de manière optimale dynamiquement selon le nombre de cœurs de la machine.

Figure 1 — Files d'executions sur chaque cœurs.



Tolérance aux pannes grâce à la gestion des erreurs

Contrairement à un langage comme Java où l'erreur est fatale, Erlang prône la philosophie du "let it crash". Comme dit précédemment, cela signifie que les processus sont conçus pour échouer de manière isolée et sans impact sur le reste du système. Lorsqu'un processus rencontre une erreur, il peut simplement se planter sans essayer de corriger l'exception, tandis qu'un autre processus, souvent supervisé par un superviseur OTP, prendra en charge sa relance.

Collecte de déchets par processus

BEAM utilise une gestion automatisée de la mémoire avec un système de garbage collection par processus, ce qui permet de maintenir des temps de réponse constants (de l'ordre de la milliseconde), sans impact négatif sur la performance globale.

Paradigme fonctionnel

Outre le paradigme de concurrence, il nous semblait important de pour une fois effectuer un projet en entier en utilisant un langage fonctionnel. Dans le cadre de notre cours de “Paradigme et Language de Programmation”, on a pu se mouiller les mains avec Haskell, mais outre les mini programmes que nous avons pu réaliser, nous n’avions jamais fait de “grand” projet du début à la fin en utilisant un langage fonctionnel.

Dans la programmation fonctionnelle, une variable est toujours immuable, il n’y a pas de système de reassignments de valeurs, donc les problèmes liés à la synchronisation disparaissent.

Processus légers

En parlant de BEAM, on a pu parler de l’utilisation de processus légers, mais que sont-ils ?

V. La concurrence dans Elixir

Les outils de base

En elixir, il est assez simple de créer un nouveau processus.

VI. Cahier des charges prévisionnel

VII. Recherches effectuées

Cette rubrique est pour l’instant temporaire, mais nous permet de nous mettre à jour sur ce que nous avons fait lors des jours. A refactor plus tard de façon plus propre

13-03-25

Recherche sur l’initialisation de projets phoenix, notamment en utilisant cette ressource ci : https://hexdocs.pm/phoenix/up_and_running.html

De plus, on a pu trouver un site qui permet de faire des exercices liés à Elixir ici un lien sur la section de la concurrence : https://elixirschool.com/en/lessons/advanced/otp_concurrency

27-03-25

Début du rapport intermédiaire !

VIII. Bibliographie

<https://www.erlang-solutions.com/blog/comparing-elixir-vs-java/>

<https://medium.com/flatiron-labs/elixir-and-the-beam-how-concurrency-really-works-3cc151cddd61>

<https://medium.com/@ck3g/introduction-to-otp-genservers-and-supervisors-cf1358d545>

BEAM

[https://en.wikipedia.org/wiki/BEAM_\(Erlang_virtual_machine\)](https://en.wikipedia.org/wiki/BEAM_(Erlang_virtual_machine))

<https://www.erlang.org/blog/a-brief-beam-primer/>

<https://www.erlang.org/blog/beam-compiler-history/>

<https://www.erlang-solutions.com/blog/the-beam-erlangs-virtual-machine/>

https://elixirschool.com/en/lessons/advanced/otp_supervisors

IX. Feature

Customized items

Figures are customized but this is settable in the template file. You can of course reference them : Listing 1.

Listing 1 — Code example

```
fn main() {  
  println!("Hello Typst!");  
}
```