

Module S4x

Programmation Concurrente

Introduction

Les bases

Programmation Concurrente

NOTE

- La programmation concurrente est un domaine de l'informatique complexe à maîtriser. Il demande des connaissances logicielles, matérielles et algorithmiques, parfois très poussées. Les programmes resultants sont souvent difficiles à déboguer.
- Dans ce module S4x de 2^{ème} de DUT, on va se focaliser sur les notions essentielles et applicables dans le cadre de vos projets courants et futurs.

INTRODUCTION à la programmation concurrente

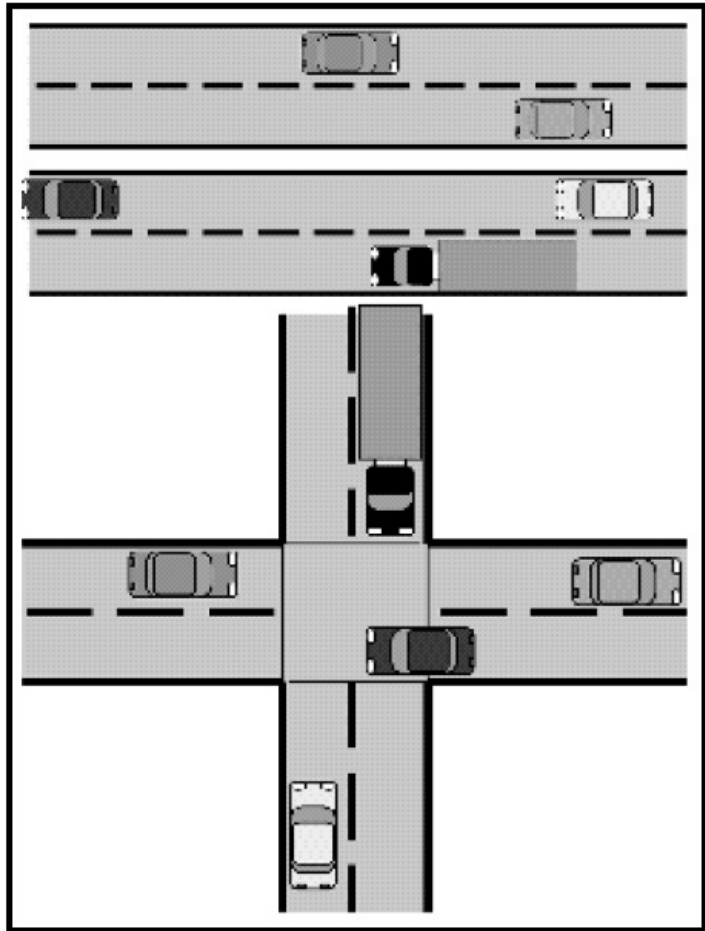
Parallélisme et Concurrency : notions

Gestion de la concurrence :
communication et synchronisation

Interblocage

Concurrence VS Parallélisme

Différence entre concurrence et parallélisme



Parallélisme

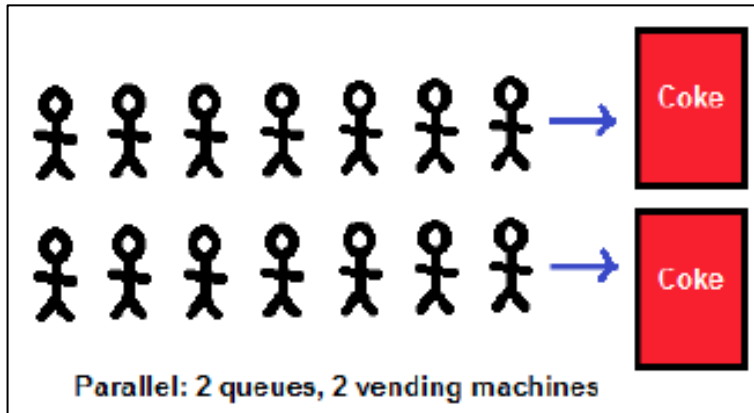
Dans la partie supérieure, des activités parallèles (dans ce cas, des voitures), qui n'interagissent pas les unes avec les autres, peuvent s'exécuter en même temps.

Concurrence

Dans la section inférieure, certaines tâches doivent attendre que d'autres terminent avant de pouvoir être exécutés.

Concurrence VS Parallélisme

Parallélisme



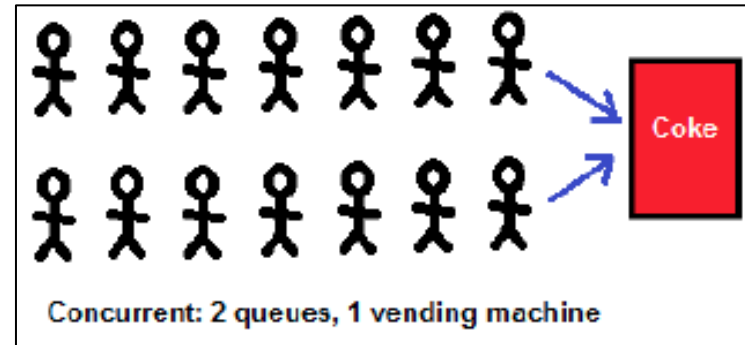
Parallélisme

art d'exécuter deux ou plusieurs actions simultanément

Indépendance

Il y a une notion d'indépendance. Ici, les deux files de personnes peuvent récupérer leurs bouteilles de coca en parallèle, indépendamment de l'autre file.

Concurrence



Concurrence

progresser sur deux choses ou plus en même temps

Ressources partagées

Avec la concurrence, la notion de ressources partagées est centrale. Ici, les deux files de personnes doivent mettre en place des règles pour accéder au distributeur de coca, partagé entre les deux files. Il s'agit de communiquer et/ou se synchroniser.

Concurrence VS Parallélisme

Propriétés des systèmes concurrents

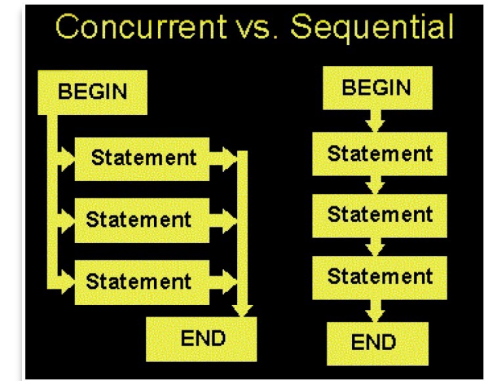
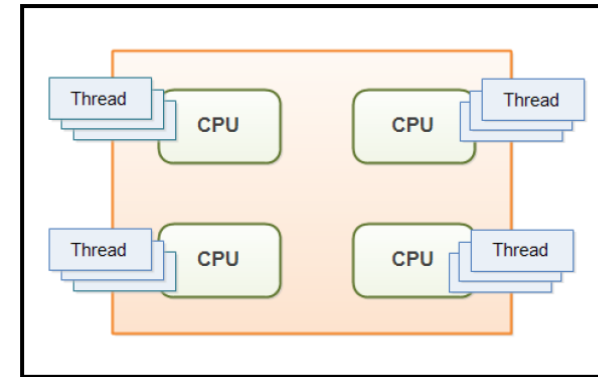
Acteurs multiples : différents *processus* et *threads* essayant tous de progresser activement sur leurs propres tâches. On peut avoir plusieurs processus contenant plusieurs threads essayant tous de s'exécuter en même temps.

Ressources partagées : la mémoire, le disque et les autres ressources que les acteurs du groupe précédent doivent utiliser pour effectuer ce qu'ils doivent faire.

Règles : il s'agit d'un ensemble strict de règles que tous les systèmes concurrents doivent suivre et qui définissent quand les acteurs peuvent et ne peuvent pas acquérir des verrous, accéder à la mémoire, modifier l'état, etc. Ces règles sont vitales pour que ces systèmes concurrents fonctionnent.

Différentes configurations matérielles

Ex : single-core single thread,
multi-threading,
multi-cores CPU,
GPU (cartes graphiques)

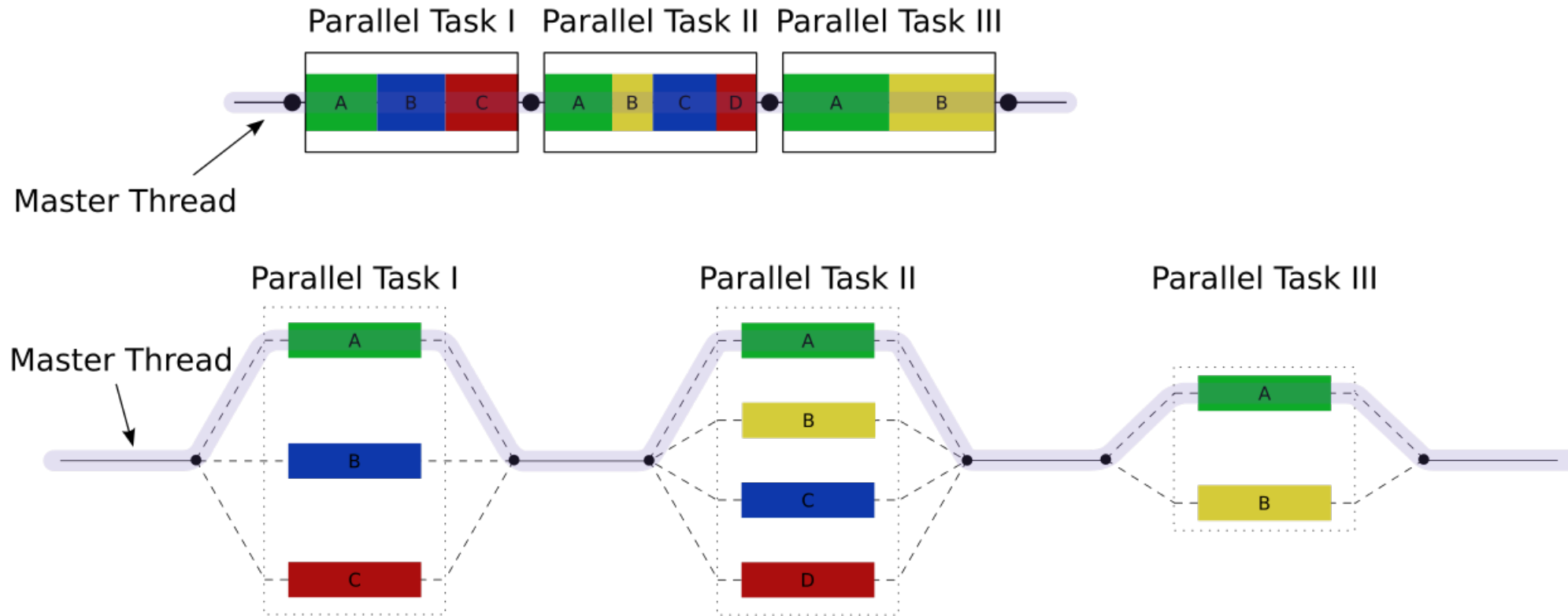


BUT principal du *parallélisme* et de la *concurrence* :

Dans les programmes, on souhaite obtenir un **gain de temps d'exécution**, comparé à une *approche séquentielle*. **Mais** tous les algorithmes, ou programmes, ne sont pas parallélisables. De plus, cela requiert plus de temps de développement, d'expertise, ainsi que d'autres coûts supplémentaires (communication et synchronisation).

Mise en place de la programmation parallèle et concurrente

Une illustration du [multithreading](#) où le thread principal crée un certain nombre de threads qui exécutent des blocks de code en parallèle.




En pratique, dans les programmes, on cherche les parties de code les plus longues à s'exécuter, on regarde si l'algorithme est parallélisable, puis si le coût de développement vaut le coup ou pas selon les contraintes de deadline, et les coûts supplémentaires de communication et de synchronisation (ex : coût du transfert de l'envoi de données sur GPU pour effectuer des calculs puis récupération des données).



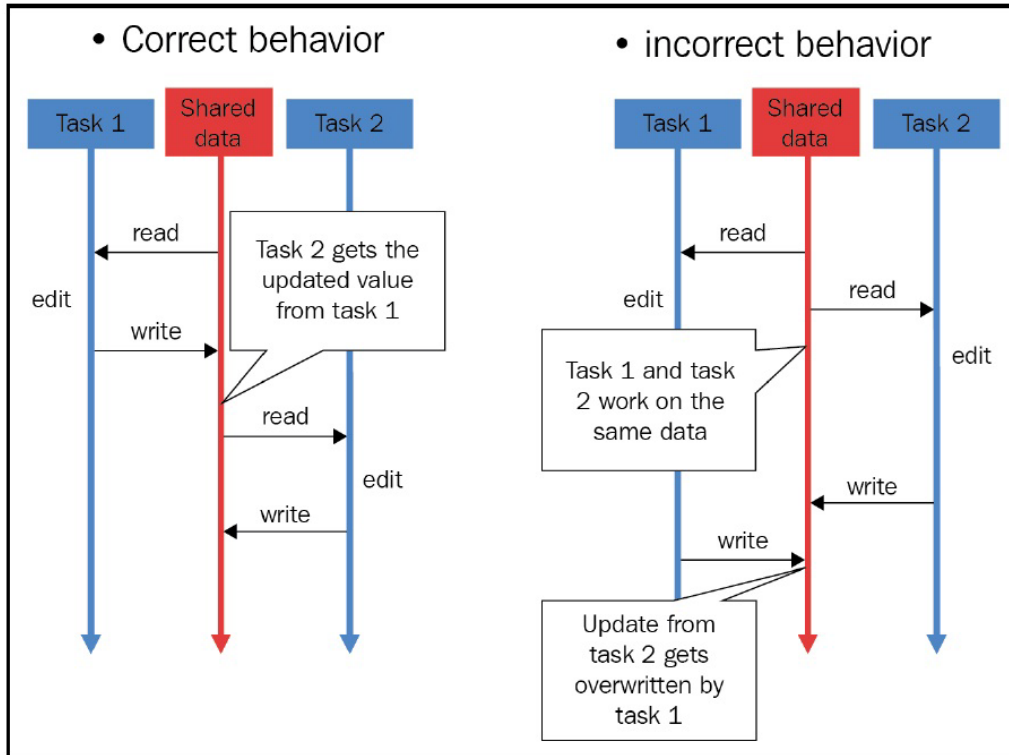
Gestion de la concurrence

race condition,
communication et synchronization,
interblocage



Accès concurrents incohérents à des ressources partagées :

Problème : ***Race Condition*** (situation de compétition)



Ex : application bancaire, 2 threads,
2 tâches parallèles : crédit et débit

Thread 1	Thread 2	Integer value
		0
read value	←	0
increase value		0
write back	→	1
	read value	← 1
	increase value	1
	write back	→ 2

Correct

Thread 1	Thread 2	Integer value
		0
read value	←	0
	read value	← 0
increase value		0
	increase value	0
write back	→	1
	write back	→ 1

Incorrect

Race condition

On parle de ***section critique*** de code à protéger (par des verrous), ici le “***increase value***” (qui incrémente la valeur).

Besoin de mécanismes de communication et de synchronisation.

Différents mécanismes de communication et de synchronisation

Il existe un ensemble de mécanismes de **communication** et de **synchronization** permettant de gérer de manière sécurisée les accès concurrents aux ressources partagées, et ainsi protéger les sections critiques de code. Par exemple : les **verrous**, mutex, **sémaphores**, **conditions**, **événements**, **barrières**.

Plusieurs langages de programmation proposent des implémentations de ces différents mécanismes et éléments, comme Python et C++. On va apprendre à les utiliser en TP et TD sur de petits exemples applicatifs concrets.

Verrou : **Lock**

Mutex : **Mutex**

Sémaphore : **Semaphore**

Condition : **Condition**

Événement : **Event**

Barrière : **Barrier**

Interblocage

Le problème du dîner des philosophes

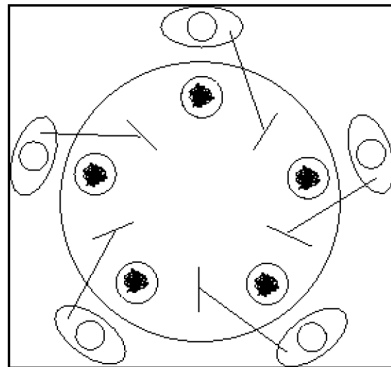
Le problème du « dîner des philosophes » est un cas d'école classique sur le partage de ressources en informatique système. Il concerne l'ordonnancement des processus et l'allocation des ressources à ces derniers et a été énoncé par Edsger Dijkstra.

La situation est la suivante : cinq philosophes (initialement mais il peut y en avoir beaucoup plus) se trouvent autour d'une table ; chacun des philosophes a devant lui un plat de spaghettis; à gauche de chaque plat de spaghettis se trouve une fourchette.

Un philosophe n'a que trois états possibles : penser pendant un temps indéterminé ; être affamé pendant un temps déterminé et fini (sinon il y a famine) ; manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation : quand un philosophe a faim, il va se mettre dans l'état « affamé » et attendre que les fourchettes soient libres ; pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette) ; si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour. En pratique, on arrive très vite à une situation de blocage où tous les philosophes ont une fourchette dans une main, et aucun ne peut la relâcher avant d'avoir l'autre. On parle d'**interblocage**.



La solution de Dijkstra a été d'utiliser des sémaphores.

Ce type de problème d'interblocage peut être difficile à voir, et peut demander une expertise. Dans le cadre du module S4x, on se va concentrer sur la pratique du parallélisme, et comment mettre en place des mécanismes de gestion de la concurrence sur des cas simples.



Approche du Module S4x

Approche projet du module S4x

On va essayer d'appliquer les concepts vu en cours par la programmation de petits exercices concrets, avec un langage simple donnant accès à tous ces éléments. On choisit Python, même s'il n'est pas optimal (on verra le problème du GIL, global interpreter lock), car il est très simple et contient tous les objets et primitives de synchronisation dont on va avoir besoin.

Python

Ensuite, on utilisera un langage de plus bas niveau comme C++, Java ou C#, qui sont plus utilisés dans le cadre de projets industriels. Ici, on choisit C++ car il contient tout ce qu'il faut et permettra également d'utiliser une autre librairie de programmation concurrente dans le même programme, OpenMP (un langage à base de directives de pré-compilation).

C++

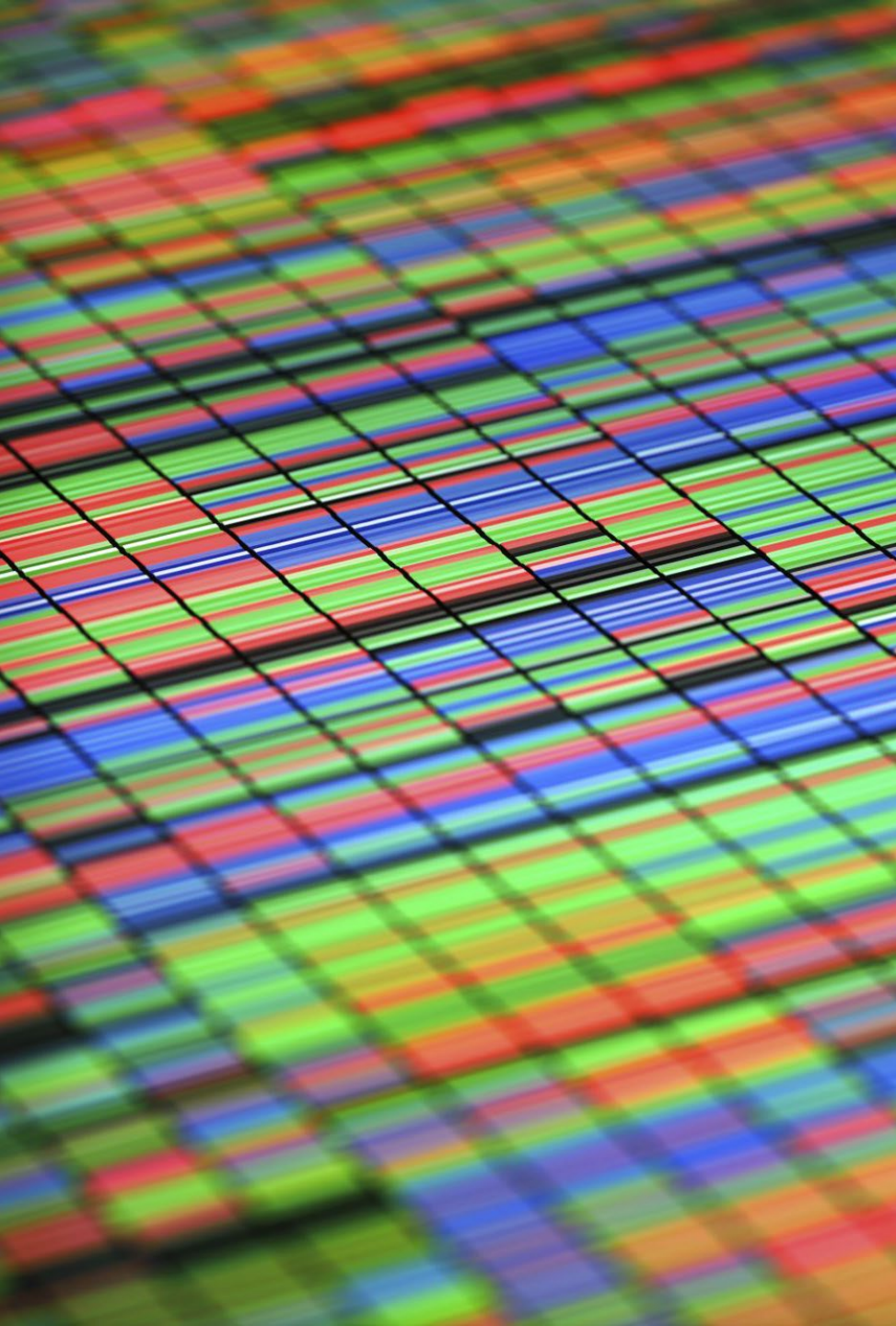
OpenMP

Puis, nous verrons une initiation à la programmation sur GPU, vu comme un accélérateur matériel : le GPU Computing. CUDA et OpenCL sont les deux API de base, mais peu de gens savent que l'on peut faire des choses équivalentes avec une API de programmation graphique comme OpenGL, ce qui permet de mélanger facilement graphique et GPU computing. Ceci permettra de créer une application contenant à la fois du C++, OpenMP et du GPU Computing, dans le cadre d'un projet de traitements d'images. On passera en mode projet intégrateur qui va permettre de faire le lien avec vos cours de synthèse d'images et de traitement d'images. Le projet pourra également intégrer des aspects réseaux (ex : accès/téléchargements d'images dans des BDD sur internet/serveurs).

GPU

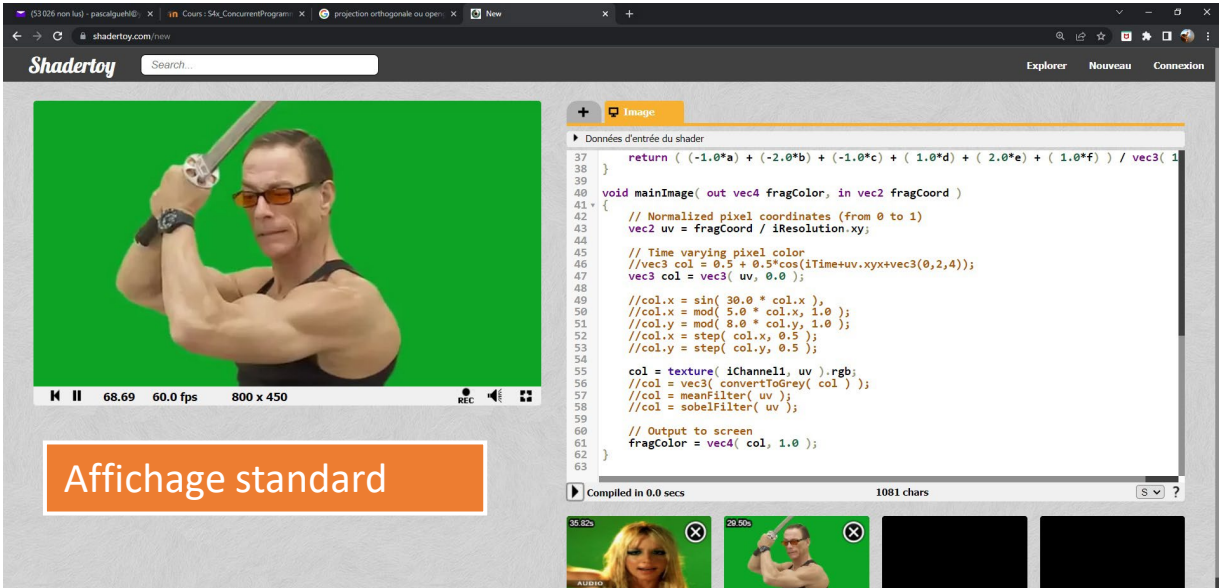
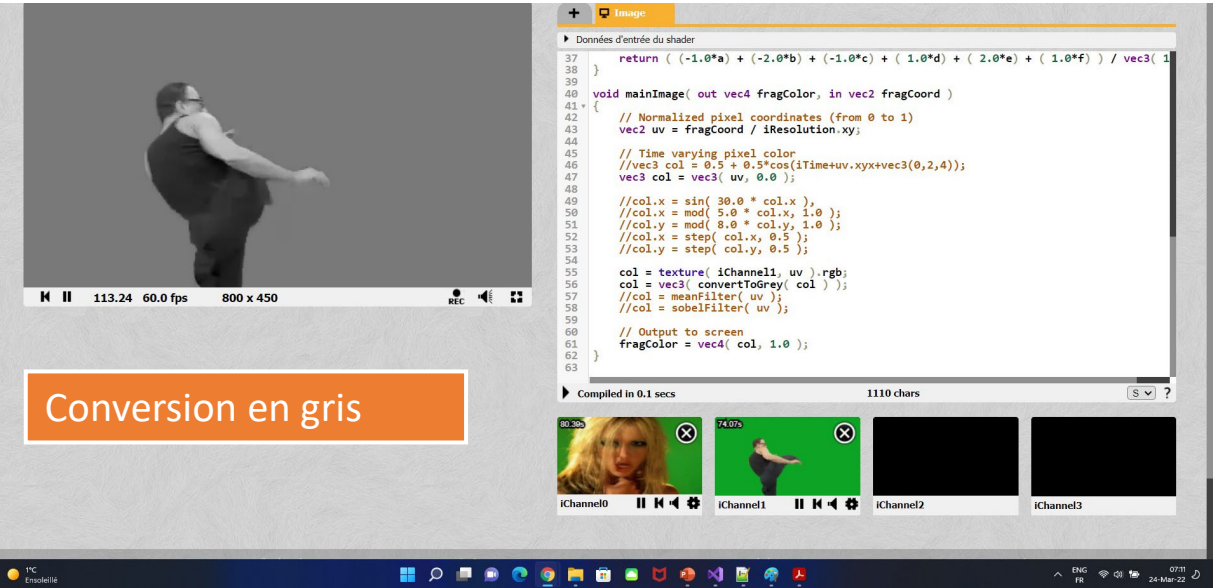
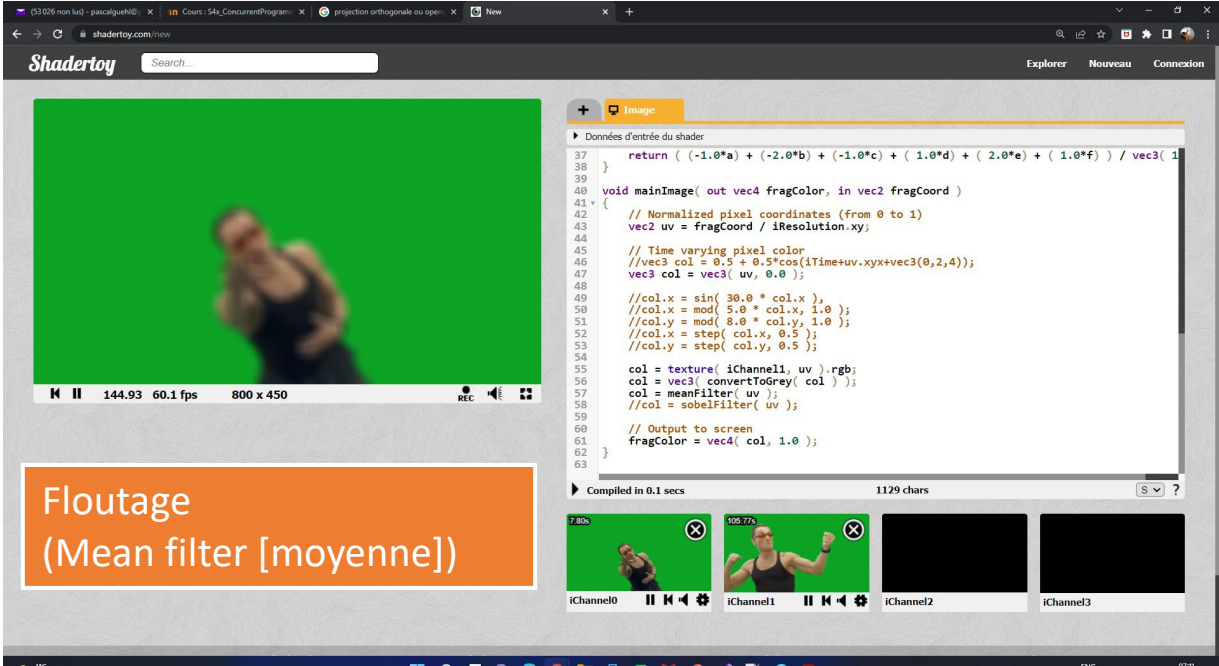
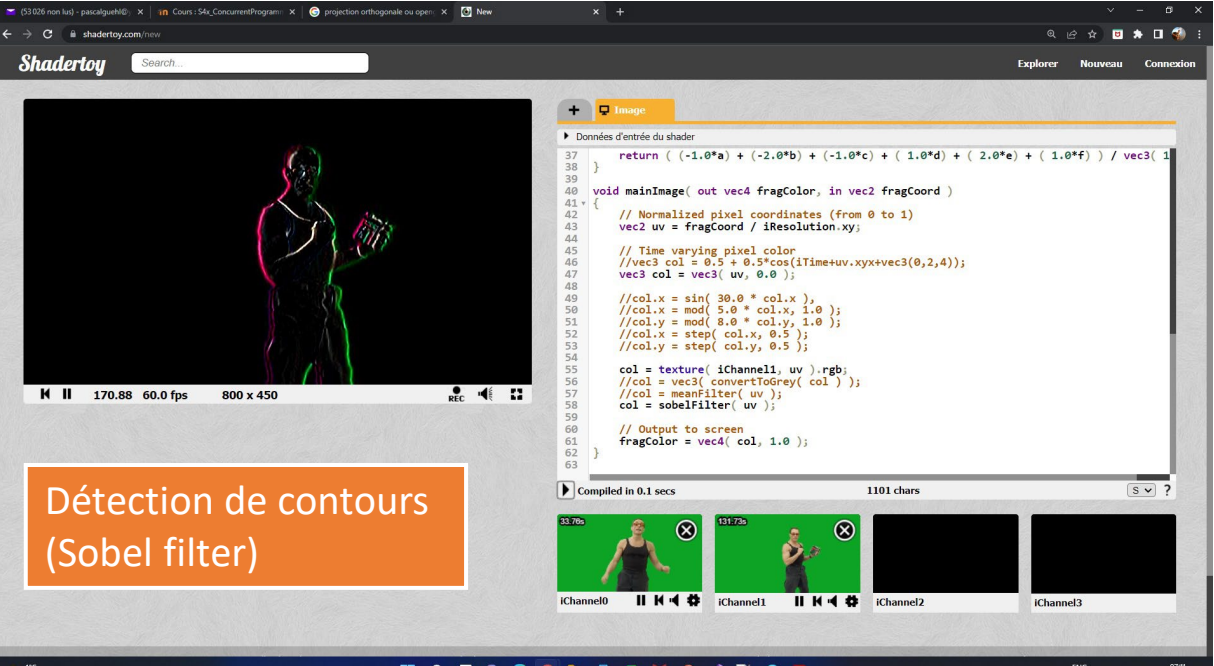
Shadertoy

OpenGL



- Application
- Image Processing

SHADERTOY : quelques exemples de traitement d'images temps-reel que l'on va coder en TP sur des flux vidéos (ou images)



PROJET : un screenshot de l'application de traitement d'images que l'on va développer ensemble avec C++, OpenMP et OpenGL

S4x: Concurrent Programming

File

Application average 16.662 ms/frame (60.0 FPS)

▶ Application Settings

Image: 341x512 pixels - 4 channels

▼ Hardware / Device

[Renderer Info]

GL Vendor: NVIDIA Corporation

GL Renderer: NVIDIA GeForce GTX 1060 with Max-Q Design

GL Version (string): 4.6.0 NVIDIA 471.41

GL Version (integer): 4.6

GLSL Version: 4.60 NVIDIA

▼ Implementation Type

☐ sequential

☐ C++ threads

☐ OpenMP threads

☒ GPU threads

Parameters:

Blocks grid: 10x16x1

Threads block: 32x32x1

Timing: 0.176128 ms

▼ Filter Type

☐ convert to grey

☒ threshold


☐ blur

☐ procedural generation


Parameters:

0.378 threshold

▼ Input Image Window



▼ Output Image Window



Module S4x

Programmation Concurrente

On a vu les notions et concepts
de bases suffisantes pour
aborder leur mise en pratique
par la programmation.