ARBACHE Rémi
JUZEAU Thibaut

Mars 2021

Rapport de projet final - MultiProgramming

CONTEXTE

Le programme fourni permet de réaliser la synchronisation, à une fréquence donnée, d'un dossier local d'une machine vers un site distant FTP. Au fur et à mesure des synchronisations, seuls les fichiers dont les caractéristiques évoluent seront recopiés vers le site FTP.
Les opérations de synchronisation des dossiers étant sérialisées, notre rôle dans ce projet a été de mettre en place des mécanismes de « multiprogramming » afin d'optimiser le temps d'exécution tout en conservant le fonctionnement du programme initial.

Pour réaliser ce projet dans de bonnes conditions, nous avons collaboré autours d'un repository git, hébergé sur Github (github.com/Thiebosh/M1_S2_parallele_distribue). Nous avons adopté une démarche itérative, où chaque version propose une modification fonctionnelle du projet :

- La branche v0 héberge le projet initial avec quelques améliorations : try-except autour de la création des dossiers (pour pouvoir resynchroniser) et des fichiers (plus de plantage).
- La branche V1 regroupe tous nos mécanismes de parallélisation et de sécurité, afin que toutes les versions reposent sur la même implémentation.
- Pour les branches V2 à 4, nous avons apportés des modifications ciblées dans l'algorithme.

APPROCHES

VERSION 1 : PASSAGE AUX THREADS ASYNCHRONES

Notre première idée a été de combiner la souplesse de l'asynchrone et la force du multithreading. Nous voulions ne causer aucune charge processeur entre deux synchronisations et exécuter plusieurs opérations FTP en parallèle lors de ces synchronisations.

Par soucis de simplicité, nous avons conservé un fonctionnement très proche de l'algorithme initial : la transmission des tâches aux threads « travailleurs » se faisant via une queue, le thread « scanneur » attend aux points critiques (création de dossier ↔ création de fichier, suppression de fichier ↔ suppression de dossier) qu'elle soit vidée avant de continuer à parcourir le répertoire en quête de tâches FTP.

En revanche, cette version centralise (rétroactivement) **tous les mécanismes de parallélisation appliqués et développés** pour ce projet :

- Les méthodes `synchronize_directory`, `search_updates`, `any_removals` et `remove_all_in_directory` sont passées asynchrones. La première, pour attendre entre deux synchronisations sans impact CPU ; les trois autres, pour « mélanger » leur exécution et car nous avons utilisé les événements, verrou et queue de la librairie asyncio.

- Nous exécutons nos tâches FTP dans des threads « travailleurs », lancés au démarrage par la méthode `thread_pool`. Ces threads ont la particularité de posséder chacun leur propre boucle asynchrone.

- Pour partager un même verrou asynchrone, les travailleurs doivent l'attendre dans la boucle asynchrone principale (`run_coroutine_threadsafe`). Idem pour 'set' un événement.

- (Développé plus tard) Pour éviter les conflits aux points critiques, `search_updates` et `remove_all_in_directory` attendent que le dossier qu'ils doivent respectivement remplir ou supprimer soit effectivement respectivement créé ou vidé avant d'envoyer leur tâche.

- L'utilisation de threads implique de pouvoir les fermer correctement. Pour cela, `synchronize_directory` dédie un thread à l'écoute de la touche 'entrée' du clavier. Dès que cet événement survient, les travailleurs en sommeil asynchrone se réveillent pour s'arrêter et les autres finissent leur tâche avant de faire de même. `synchronize_directory` se réveille aussi, attend que tous les travailleurs se soient fermés et se ferme à son tour.

- En cas de fermeture forcée (CTRL+C), l'exception est interceptée et transformée en événements de fermeture (`evt_end.set()`).

- Le fait que le thread principal (« scanneur ») ne s'occupe que de lister les tâches à réaliser peut provoquer des décalages avec les travailleurs : le temps d'exécution n'est pas du même ordre de grandeur. Pour assurer la synchronicité entre tous les threads, nous avons développé un mécanisme inspiré du « 3-way handshake » des communications TCP (réseaux) :
    - SYN – le scanneur signale la fin de son activité (`evt_done_main.set()`) et attend confirmation (`await evt_done_workers.wait()`)
    - SYN ACK – le premier travailleur à n'avoir aucune tâche et cet événement confirme être libre pour la prochaine synchronisation (`evt_done_main.clear()`) et passe l'information aux prochains travailleurs (`evt_done_workers.set()`). Le scanneur et les travailleurs attendent donc la durée fixée par l'utilisateur entre deux synchronisations.
    - ACK – le scanneur sort de sommeil et coupe le signal (`evt_done_workers.clear()`).

- Le problème du système décrit précédemment est qu'un travailleur peut accumuler du retard. Pour pallier cela, le travailleur en charge du SYN ACK mémorise l'heure associée dans une variable partagée par tout le processus (`multiprocessing.Value`). Le temps de sommeil inter-scan est donc ajusté selon le retard du travailleur.

VERSION 2 : 2 GROUPES, 2 QUEUES

En se familiarisant avec l'algorithme et ses temps d'exécutions relatifs, nous avons remplacé l'attente aux points critiques par deux queues avec une notion de priorité : `queue_high` prend les tâches relatives aux créations de dossiers et suppression de fichiers (amont des points critiques) et `queue_low` prend les tâches de transfert de fichier et suppression de dossier (aval des points critiques). Cela dans le but de fluidifier l'exécution et de tirer parti d'un grand nombre de files d'exécution.

VERSION 3 : DES FICHIERS AUX POIDS VARIABLES

En exécution, il est apparu clairement que le plus long était de transférer les fichiers. Certains fichiers conséquents finissaient leur transfert bien après le 3-way handshake. La sérialisation des tâches faisait que le poids des fichiers importait peu, mais la parallélisation change la donne : envoyer les fichiers par taille décroissante permet d'avoir un remplissage plus uniforme des files d'exécutions.

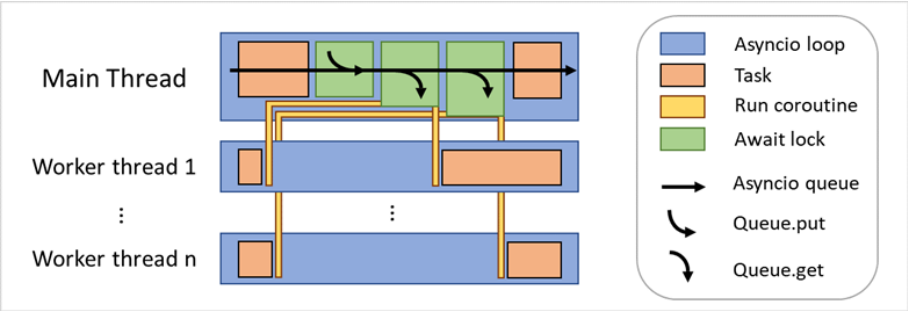
Nous avons donc remplacé l'envoi de fichiers dans `queue_low` par leur collecte et leur tri par poids décroissant avant de les envoyer aux travailleurs. Le temps perdu en exécution parallèle est retrouvé avec une fin d'exécution plus uniforme.

VERSION 4 : OPTIMISATION EMPIRIQUE

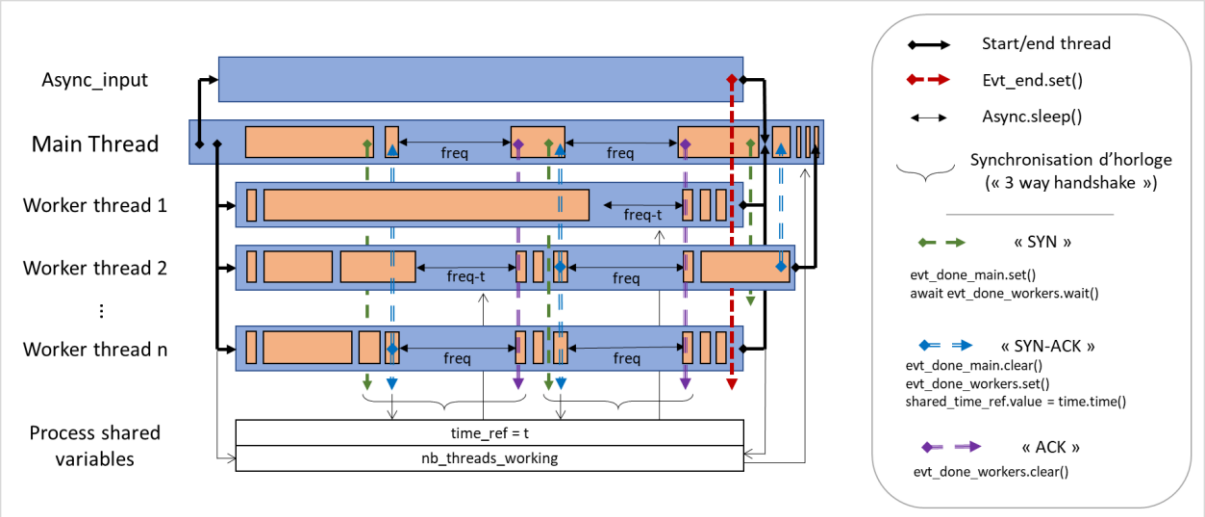
Après avoir effectué des mesures de temps d'exécution, il est apparu qu'un faible nombre de files réduisait les performances plutôt que de les améliorer. Nous avons donc décidé de sérialiser le comportement du programme si le paramètre `nb_multi` est inférieur à un seuil prédéfini. De la même manière, la suppression de fichiers semble être toujours plus lente lorsque le programme est sérialisé, même avec un nombre de threads élevé. Nous avons donc systématiquement sérialisé la suppression des fichiers dans le but d'obtenir de meilleures performances.

ARCHITECTURE :

Les deux schémas ci-dessous viennent résumer en image les mécanismes de parallélisation mis en œuvre pour ce projet :



Ce schéma illustre l'envoi de tâches dans la queue et leur récupération asynchrone.



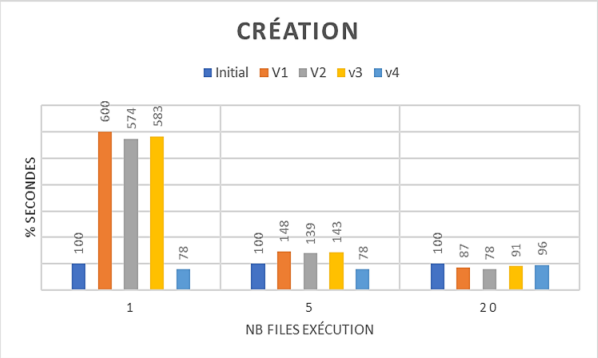
Ce schéma illustre les mécanismes utilisés / développés tels que décrits dans la V1

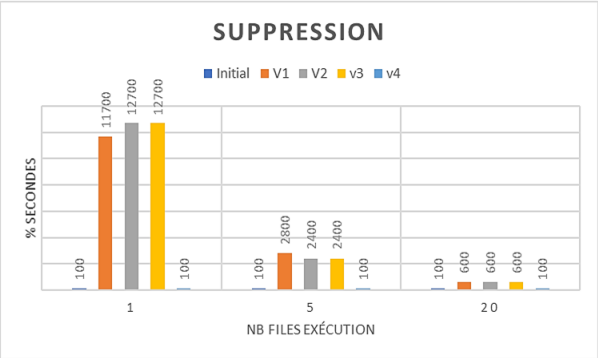
CONCLUSION

TEMPS D'EXECUTIONS :

Nous avons constitué une archive type, comprenant 90 fichiers de poids globalement équivalents répartis entre 24 dossiers, ainsi qu'une vidéo placée stratégiquement en fin d'arborescence. Pour nos mesures, nous placions l'archive dans le dossier avant démarrage et le temps relevé est l'écart entre l'heure du log « valid parameters » et de celui de la dernière tâche FTP.

Les temps passés par la version d'origine servent de référence (création : 23,6s / suppression : 0,5s).





Pour plus de précision, il faudrait faire la moyenne de plusieurs exécutions à chaque fois.

POINTS DELICATS ET DIFFICULTES RENCONTREES :

Une difficulté majeure a été de combiner les mécanismes de multithreading et d'asyncio, peu vus en cours. Il nous a notamment fallu se rendre compte que l'acquisition de l'asyncio lock et l'application de flag d'asyncio events doivent être faits dans une seule et même async loop, et trouver comment déporter ces choses dans d'autres async loops.

Un autre point peu abordé : nous n'avions vu que la transmission d'information sous forme de file, mais jamais de partage de variable entre plusieurs threads. Nous avons eu l'idée de passer par les variables de multiprocessing.

OBSERVATIONS :

De la même façon que l'application des mécanismes de multiprogramming représente un projet à part entière, un certain nombre de ressources sont dédiés à leur seul fonctionnement.

Notre approche cherche à limiter cet impact avec l'implémentation d'asyncio mais cela reste significatif lorsqu'on regarde les résultats en termes de temps d'exécution : il faut environ 5 threads pour réduire le facteur de temps en dessous de 2, et au moins une 15aine pour retrouver des temps équivalents (en création uniquement).

C'est avec ce constat en tête que nous avons développé notre V4, qui navigue entre sérialisation et parallélisation pour offrir des performances optimales selon la situation.

A noter qu'un avantage certain du multithreading, combiné à notre système de 3-way handshake, est que le système peut finir une tâche longue en arrière-plan sans bloquer les tâches futures.

Il est possible que l'aspect asynchrone du lock et de la queue ralentisse l'ensemble du système (awaits).