Ordonnancement automatique et parallèle du flux de données appliqué à la radio logicielle et notamment au logiciel AFF3CT

Diane Orhan

Mémoire de stage

Ordonnancement automatique et parallèle du flux de données appliqué à la radio logicielle et notamment au logiciel AFF3CT

Université de Bordeaux
M2 CALCUL INTENSIF ET SCIENCES DES DONNÉES
2021 – 2022

Rédigé par
Diane ORHAN

Encadrants :
Denis Barthou, Laércio Lima Pilla, Olivier Aumage

Table des matières

Table des figures

Liste des tableaux

Ce mémoire de stage concerne le stage que j'ai effectué du 7 février au 15 juillet 2022 dans le cadre de mon année de Master 2 Informatique parcours Calculs Intensifs et Sciences des Données (CISD). Il s'est déroulé au sein de l'équipe de recherche STORM (Optimisation statique, Méthode d'exécution) au centre Inria de l'Université de Bordeaux et encadrée par Denis Barthou, Laercio Lima Pilla et Olivier Aumage. Mon stage s'intéressait à l'ordonnancement automatique et parallèle de flux de données dans le cadre de la radio logicielle et notamment dans le cadre du logiciel dédié AFF3CT. Dans ce cadre-là, j'ai pu également rencontrer certains membres de l'équipe CSN de l'IMS qui travaille en collaboration avec l'équipe STORM pour le développement de ce logiciel.

J'ai choisi ce stage, car il se situe à la convergence de deux domaines : le calcul haute performance et la télécommunication. Venant d'une formation mécatronique, j'aime particulièrement les problématiques aux interfaces de plusieurs disciplines. De plus, le choix d'un stage dans un environnement recherche était motivé par l'envie de continuer le stage par une thèse dans la même problématique.

Environnement du stage

Équipe STORM du centre de recherche Inria

L'Inria est un établissement public à caractère scientifique et technologique français. Sa mission principale est le développement et la valorisation de la recherche dans le domaine de l'informatique. Elle possède plusieurs centres de recherche en France dont celui de Bordeaux dont les axes prioritaires de recherche tournent autour du calcul haute performance [9], du Big Data, de l'apprentissage automatique et de la santé numérique. Cet institut de recherche national travaille dans le cadre de projets avec de nombreux partenaires, que ce soit d'autres universités, des centres de recherches ou des entreprises.

L'équipe dans laquelle j'ai pu travailler est l'équipe STORM dont le responsable est Denis Barthou. Il s'agit d'une équipe qui rassemble des membres, en plus de l'Inria, du CNRS, de l'université de Bordeaux et du groupe Bordeaux INP. Cette équipe se concentre sur la notion de parallélisme dans le cadre du domaine du calcul haute performance ou HPC (*High Performance Computing*). Ce domaine vise à obtenir des performances élevées en termes de débit, latence, efficacité, faible énergie consommée ou autres métriques en prenant en compte l'électronique, l'architecture de la machine, les logiciels, les langages de programmations, l'algorithmique sur les machines utilisées. C'est dans cet objectif-là que l'équipe possède plusieurs axes de recherche avant l'exécution du programme (*compilation*), pendant l'exécution (*runtime*) et après l'exécution (*analyse*). C'est notamment le cas pour le développement de DSL (*Domain Specific Language*) qui est un langage spécifique à un domaine comme c'est le cas avec le logiciel AFF3CT. Celui-ci fournit une bibliothèque de codage canal, mais également un simulateur de chaîne de télécommunication. Dans ce cadre-là, j'ai également pu participer à la collaboration de l'équipe STORM avec l'équipe CSN de l'IMS (Laboratoire de l'Intégration du Matériau au Système) qui travaille sur l'adéquation algorithme-architecture. Des réunions entre les deux équipes de recherches, auxquelles j'ai participé, sont organisées chaque mois pour discuter des avancements autour d'AFF3CT. D'autre part, j'ai pu également assister à des conférences du Groupement de Recherche ISIS (Information, Signal, Image et ViSion) sur l'*Implémentations, outils et applications émergent pour la radio logicielle* du 7 mars 2022 à Jussieu, Paris.

Travail demandé

Le travail demandé est de concevoir des algorithmes d'ordonnancement statique de graphes de tâches issus de chaîne de télécommunication. Il peut s'agir de chaîne d'envoi ou de réception de flux continu de données (*streaming*), comme ce qui représenté sur la figure 2 et qui sera expliquée plus en détail dans la partie 2.2. Pour effectuer le traitement, ces chaînes disposent d'un nombre limité de ressources et sont supposées toutes identiques. Le but est donc de construire un algorithme donnant une répartition automatique

tâche-ressource en fonction du nombre de ressources disponibles et permettant de maximiser le débit de traitement de ces chaînes lors de leur exécution.

Dans cette optique-là, j'ai développé un simulateur en Python permettant de valider ou non mes choix algorithmiques. Les opérations possibles pour l'organisation des tâches étaient limitées aux suivantes : pipeline (découpage en étages) et duplication d'étage, ce qui a permis de contraindre un peu l'étude au vu du temps imparti. Les chaînes considérées sont uniquement des séquences de tâches comme celles implémentées dans le logiciel AFF3CT et ont permis d'évaluer les différents algorithmes développés. Il est normalement possible de rencontrer des tâches de type itérative et conditionnelle, mais celles-ci ne sont pas considérées dans ce travail, elles pourront faire l'objet d'une autre étude.

Pour cela, j'ai eu à disposition le code source du logiciel AFF3CT [1], des ressources bibliographiques, des exemples de chaînes de communication réellement implémentées et utilisées dans l'industrie, et enfin, j'ai pu bénéficier des conseils avisés de mes encadrants.

Planning

Je n'ai pas eu de planning précis de la part de mes encadrants pendant mon stage. Il est cependant possible de distinguer trois grandes phases :

1. Prise en main du sujet :
   (a) faire de la bibliographie ;
   (b) se former au logiciel AFF3CT.
2. Développement d'une heuristique d'ordonnancement automatique de chaîne :
   (a) poser un modèle avec ses contraintes ;
   (b) tester des solutions naïves ou issues de la littérature ;
   (c) développer un modèle plus complet par rapport au problème posé ;
   (d) tester et évaluer les heuristiques à l'aide d'un simulateur Python.
3. Rédaction du rapport.

En plus de mon stage, il y a eu également des tâches annexes comme le rendu de rapport lié à l'UE Numerics, la préparation de dossier de bourses de thèse, la préparation de présentation visant à l'obtention de bourse de thèse, diverses réunions, la participation à des conférences et séminaires. L'ensemble de ces évènements additionnels au stage m'ont permis également m'immerger dans la vie d'une équipe de recherche.

1. github.com/aff3ct/aff3ct.git

1 Introduction

La communication est l'action ou le fait de communiquer, de transmettre quelque chose. Elle est régie par un ensemble de règles définies au préalable entre deux interlocuteurs pour pouvoir échanger de l'information. Par exemple, en France, lorsque l'on reçoit un appel téléphonique, il faut d'abord décrocher puis dire «Allô» puis attendre que l'interlocuteur parle : il s'agit du protocole de mise en relation de deux interlocuteurs. Ce même principe de règles de communication pour pouvoir transmettre de l'information se retrouve dans les technologies de communication radio ou téléphonique et ces règles sont contraintes par le matériel utilisé pour ces technologies. La partie 1.1 présente l'intérêt d'avoir une implémentation logicielle de la radio par rapport à une implémentation sur matériel dédié. Puis il sera montré d'un point de vue historique pourquoi cette idée n'était pas réalisable avant dans la partie 1.2 et enfin, vis-à-vis ce qui est fait aujourd'hui, qu'est ce qui pourrait être encore amélioré dans la partie 1.3.

1.1 Les standards de communication numérique

Tout comme pour la communication orale humaine, il existe des standards de communication pour les réseaux informatiques et les télécommunications, cela rassemble l'ensemble des protocoles, des règles à suivre pour transmettre de l'information. Ces standards ont évolué au cours du temps, de la 1G jusqu'à l'apparition de la 5G maintenant. La figure 1 résume l'ensemble de l'évolution des réseaux de communication. Cette sous partie présente cette évolution des standards.

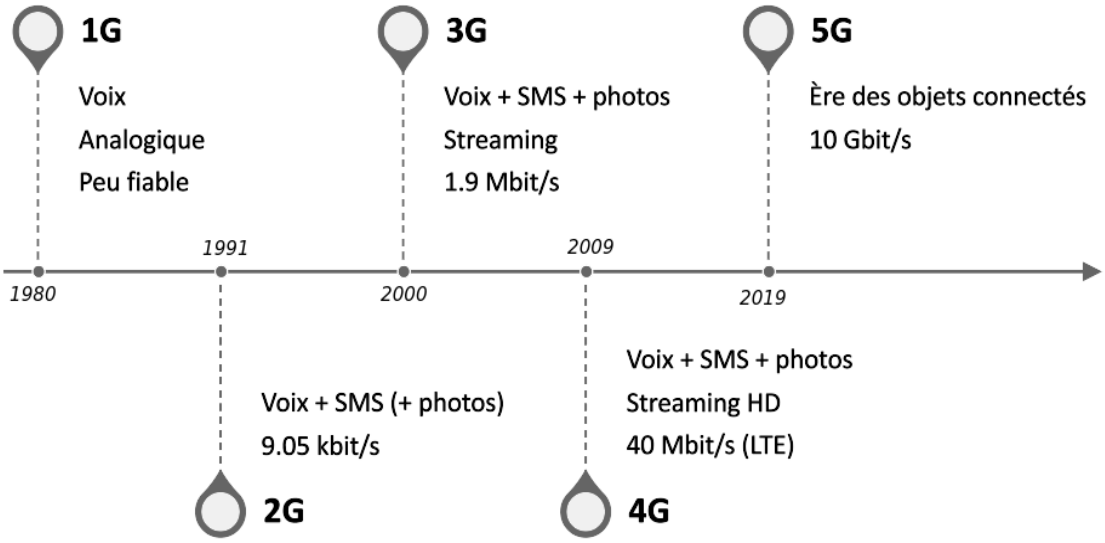


FIGURE 1 – Évolution des réseaux de communication.

La figure 1 résume l'ensemble de l'évolution des réseaux de communication. La première génération de réseaux mobiles 1G, dédié uniquement à la voix [10], repose sur l'analogique uniquement. Il existe plusieurs standards de communication utilisant ce réseau selon les pays et région du monde. Les principales faiblesses de ce réseau sont la mauvaise qualité du son et le manque de sécurité. Puisque les données n'étaient pas encryptées, elles pouvaient être lues en clair par n'importe quel individu récupérant les données. Dès les années 90, le réseau 2G fait son apparition et commence à utiliser la technologie numérique, elle permet de transmettre non seulement la voix, mais aussi des SMS puis des photos également. Les données transmises sont maintenant encryptées, les données sont par exemple arrangée dans un ordre précis connu seulement

de l'émetteur et du récepteur de façon à ce que ce soit les seuls à pouvoir lire les informations du message, un individu interceptant le message ne sera pas en capacité de le comprendre. Ainsi, la communication est donc plus fiable. L'arrivée de la 3G, au début des années 2000, permet d'atteindre des débits de communication de plusieurs centaines de Kbit/s (haut débit) soit environ quatre fois plus élevé que la 2G. L'utilisation mobile d'Internet se généralise, il est alors possible d'envoyer des vidéos, de regarder des vidéos en streaming et de bénéficier du GPS. Dès 2009, la 4G fait son arrivée et permet d'accéder au très haut débit (100 Mbit/s). Avec ce type de débit, il est possible de regarder des vidéos en streaming en haute définition, de faire de la visioconférence en meilleure qualité. Ce nouveau réseau en général améliore la qualité de l'expérience utilisateur en plus de permettre de transférer des fichiers plus rapidement. La 5G propose des débits de communication toujours plus élevés de l'ordre de la dizaine de Gbit/s et a but de diversifier l'usage des communications, que ce soit des communications mobiles à grande portée et des communications à plus faible portée comme c'est le cas pour les objets connectés.

Chacune des générations pallie les faiblesses de la génération précédente et apporte de nouvelles fonctionnalités tout en offrant un débit toujours plus élevé. Historiquement, à chaque standard correspond un matériel spécifique, c'est-à-dire des cartes électroniques spécialement conçues pour cette utilisation et correspond à des performances spécifiques en débit, latence, consommation énergétique et puissance instantanée. Ce qui signifie qu'environ tous les dix ans, avec l'apparition d'un nouveau standard, il faut changer le matériel au niveau des antennes et stations relais, ce qui peut représenter un coût important tant financier et temporel. C'est ainsi qu'il y a un intérêt à réaliser une implémentation logicielle des standards de communication sur du matériel programmable.

1.2 Des avancées technologiques au service de la communication

L'implémentation logicielle des standard de communication n'a pu se faire avant que les processeurs génériques puissent fournir des performances similaires à celle du matériel dédié spécifiquement. Cette partie, en retraçant l'historique de la radio diffusion en parallèle de celle des processeurs génériques, explique comment il a été possible matériellement que les deux domaines s'allient pour former la radio logicielle.

Les premières radiodiffusions ont lieu dès les années 1890 [10]. Il faudra attendre un siècle pour que l'armée américaine crée, dans les années 90, la première radio utilisant des processeurs générique avec le système SPEAKeasy [8], dont l'idée est d'avoir toutes les fonctions de la radio programmable sauf l'antenne et le convertisseur analogique-numérique. L'idée est d'avoir un système très portable pouvant tourner sur n'importe quel ordinateur contenant des CPU. Cependant, les CPU ne contiennent que des opérations d'addition ou de décalage et pas de multiplication, alors que l'opération principalement utilisée dans les traitements de radio sont des multiplications. Le fait de réaliser une opération de multiplication est donc très coûteuse en temps, ce qui peut poser un problème pour ce type de système temps réel. De plus, les processeurs standards chauffent beaucoup et consomment beaucoup plus qu'avec une architecture non reconfigurable.
Les architectures de CPU ont évolué depuis, il est maintenant possible de paralléliser les traitements à plusieurs échelles, ce qui permet d'augmenter le débit de calcul. Il est possible de réaliser plusieurs flots d'instruction en même temps grâce aux architectures multi coeurs et plus exactement multi threads. De plus, au sein d'une même unité de calcul, il est possible de réaliser plusieurs fois la même opération en parallèle sur des données différentes, c'est-à-dire de faire du SIMD *single instruction multiple data*. L'utilisation de GPU utiliserait ce genre de parallélisme, mais les coûts de communications pour déplacer les données seraient beaucoup trop importants, c'est pour cela que ce type d'architecture n'est pas envisagé dans ce travail. D'autre part, il faut pouvoir exploiter cette architecture depuis les traitements et fonctions à réaliser par la chaîne de communication, c'est-à-dire déterminer quelle fonction peut être parallélisée et de quelle façon. Il appartient à l'utilisateur·ice de déterminer cela manuellement, ce qui peut se révéler extrêmement fastidieux. Les bibliothèques de radio logicielles exploitent aujourd'hui la parallélisation à l'échelle des unités de calcul, comme c'est le cas d'AFF3CT qui embarque une bibliothèque d'instructions

vectorisées MIPP [2] [4] à cet effet. Il est alors possible d'atteindre des débits de traitement similaire à ce qui existe avec des architectures dédiées et implémentation purement matérielle.

1.3 Vers une meilleure exploitation du matériel générique

L'aspect de parallélisation à l'échelle des différents threads n'est pas ou très peu exploité aujourd'hui, avec l'attribution d'un unique thread par tâche, en particulier dans GNU Radio [13]. Cette sous partie explique quels seraient les avantages à tirer parti de cet aspect des processeurs génériques et de quelles façons.

Augmenter encore le degré de parallélisme et d'optimisation pour l'exécution des standard de communication permettrait d'améliorer encore les performances, notamment de débit de traitement de l'information. Ce qui est en accord avec l'évolution des différends standard, il est en effet possible de remarquer que d'une génération à l'autre (voir figure 1) que le débit peut être multiplié entre 4 et 1000 fois. Pour exploiter cet aspect de parallélisme, il s'agirait de choisir à la compilation dans quel ordre effectuer les tâches et par quelle(s) ressource(s), donc de réaliser un ordonnancement statique des tâches, qui serait fait à la compilation. Il n'est pas possible d'envisager un ordonnancement dynamique, fait lors de l'exécution de la chaîne de communication. En effet, si l'allocation de ressources et le choix de tâches à exécuter se fait à l'exécution, il faut rajouter des surcoûts de calcul pour ce choix-là et éventuellement des surcoûts de communication des données. Or, pour ce type d'application, il est important d'avoir un débit le plus élevé possible pour l'utilisateur·ice final·e qui préférera pouvoir télécharger le plus rapidement possible des fichiers ou photos par exemple. L'ordonnancement statique permet d'éviter ce problème-là. Étant donné que l'idée de la radio logicielle est de pouvoir être porté sur tout type d'architecture générique, il faudrait refaire ce travail pour chaque système sur lequel se fait la communication et chaque nouveau standard.

C'est ainsi que je vais présenter mon travail effectué dans le cadre de mon stage dont l'objectif est de travailler sur l'**ordonnancement statique et automatique de tâche dans le cadre de la radio logicielle**. Dans la section 2 est décrit ce qu'est la radio logicielle, quel est le fonctionnement d'une chaîne de communication numérique pour mieux en comprendre les enjeux en 2.1, en 2.2 est présenté le modèle de graphe adopté pour ce type d'application puis la partie 2.3 quelles sont les bibliothèques de radio logicielle existantes et leurs caractéristiques et en particulier le logiciel **AFF3CT**. La section 3 présente la **modélisation du problème** que j'ai pu effectuer avec notamment un exemple. Puis la section 4 présente mes contributions sur le **développement d'heuristiques** avec tout d'abord une version utilisant l'aléatoire en 4.1, puis un algorithme optimal de pipeline pour l'équilibrage des charges dans la partie 4.2 et enfin la partie 4.3 traite de l'heuristique principale développée mêlant **pipeline** et **duplication**. La section 5 est dédiée à la mise en pratique des algorithmes d'ordonnancement avec la présentation du simulateur Python que j'ai réalisé et **l'évaluation expérimentale sur une chaîne existante d'AFF3CT** en comparaison aux heuristiques de la section 4.

2 La radio logicielle

La radio logicielle désigne des éléments de transmission radio, comme des émetteurs ou des récepteurs, réalisés de manière logicielle avec du matériel a priori non dédié (processeurs standards) à cette application en particulier. Lors de la transmission et la réception de données via la radio, il existe un certain nombre d'étapes par lesquelles doit passer l'information pour pouvoir être transmise puis réceptionnée. L'ensemble des traitements de la chaîne radio sont détaillés dans la partie 2.1. Il est possible de modéliser ces chaînes de communication ainsi que les différents dataflows possibles, certains de ces modèles issus de la littérature

2. github.com/aff3ct/MIPP.git

sont détaillés dans la partie 2.2. Il existe par ailleurs des bibliothèques ayant fait l'implémentation logicielle de ces traitements pour pouvoir facilement en faire leur exploitation comme le présente la partie 2.3.

2.1 Chaînes

Une chaîne typique de communication est celle décrite par la figure 2. Elle consiste à faire transiter un message d'information entre un émetteur (*transmitter*) vers un récepteur (*receiver*) [15] via un canal de transmission (*channel*). Ce dernier peut être de différente nature : câble coaxial, guide d'ondes, fibre optique, l'air, etc. Chacun de ces supports est caractérisé par une bande passante différente, c'est-à-dire que l'intervalle de fréquence sur lequel peut transiter un signal diffère selon la nature du canal. Le signal dans un canal est toujours de nature analogique même si l'information contenue est numérique. De ce fait, le signal peut être affecté, lors du passage dans un canal, par des perturbations électromagnétiques pouvant provoquer des erreurs. Ce peut être par phénomène d'absorption dans le milieu de propagation atténuant le signal ou encore par phénomène de dispersion qui provoque une distorsion du signal transmis. Chaque canal possède des caractéristiques qui lui sont propres, ainsi il existe plusieurs modélisations de canal. La plus courante est celle du canal **AWGN** *additive white gaussian noise*, c'est-à-dire que le bruit est **additif**, il se rajoute au signal existant, **blanc**, le bruit est réparti de manière homogène sur toute la bande de fréquence du canal et **gaussien**, la distribution du bruit est normale et centrée en zéro. Pour l'ensemble de ce rapport, c'est ce modèle de canal qui sera pris.

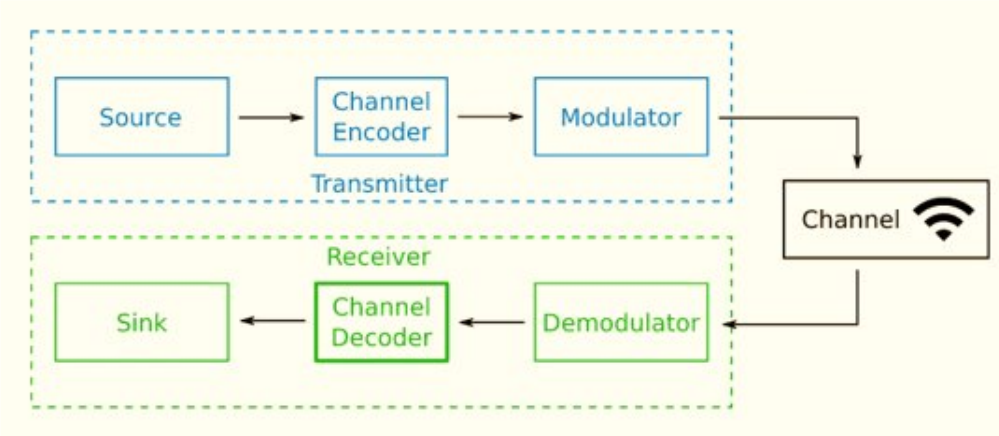


FIGURE 2 – Chaîne de communication
source : aff3ct.readthedocs.io.

Le but de l'opération de **modulation** du signal est en premier lieu d'adapter le signal au canal à une bande de fréquence favorable à la transmission [12]. Cette opération permet également de réduire la sensibilité du signal aux bruits et interférences décrites plus haut. Un signal analogique $s(t)$ peut s'écrire de la manière suivante :

$$s(t) = A \times cos(2\pi ft \ + \ \Phi)$$

Avec :

- f : la fréquence du signal ;
- A : l'amplitude du signal ;
- Φ : la phase du signal.

Il existe donc trois variables qui peuvent porter l'information utile. Si c'est la fréquence $f(t)$ qui code l'information, alors c'est une modulation FSK (*Frequence Shift Keying*). Cette modulation peut s'observer sur la figure 3 : le signal binaire contient l'information avec les bits à envoyer, l'onde porteuse correspond

au signal $s(t)$ de fréquence f_0, enfin le signal modulé est obtenu en encodant la donnée à envoyer par la fréquence $f(t) = f_0 \pm \delta f$. Dans cet exemple, pour envoyer '1' la fréquence prend la valeur $f_0 + \delta f$ et pour envoyer '0' la fréquence prend la valeur $f_0 - \delta f$.

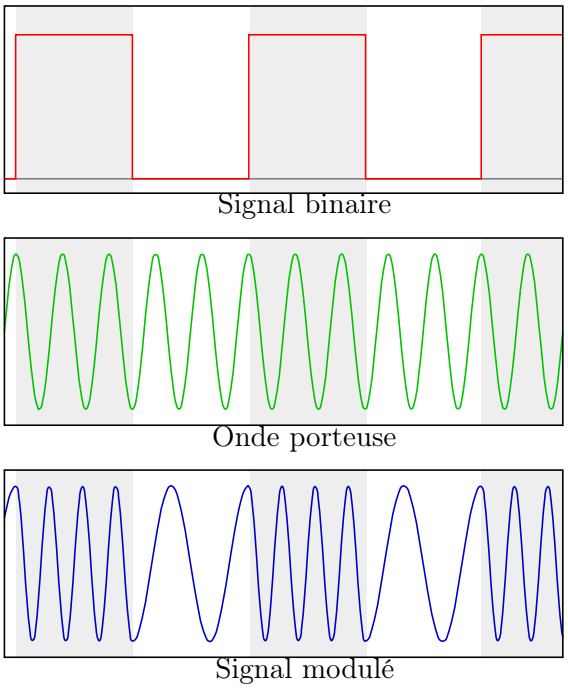


FIGURE 3 – Modulation numérique de fréquence FSK
source : Par Ktims-commons.wikimedia.org, CC BY-SA 3.0.

Si c'est l'amplitude du signal $A(t)$ qui porte l'information, alors c'est une modulation ASK (*Amplitude Shift Keying*), typiquement un bit est codé par une valeur d'amplitude. La figure 4 illustre cette modulation d'amplitude, ici le signal peux prendre deux valeurs d'amplitudes distinctes traduisant les bits d'information. Enfin, si c'est la phase $\Phi(t)$ qui code l'information, alors c'est une modulation PSK (*Phase Shift Keying*). La figure 5 présente cette dernière modulation, il est possible par exemple d'observer des sauts de phase à $t = 1$, $t = 2$ et $t = 4$ sur le graphique. Chacun de ses sauts de phase correspond à un changement de bits, par exemple si une série de '1' est envoyée, au premier '0' rencontré il y aura un changement de phase.

Dans les faits, l'information n'est pas envoyée bit à bit, mais par paquets de plusieurs bits. Ces paquets

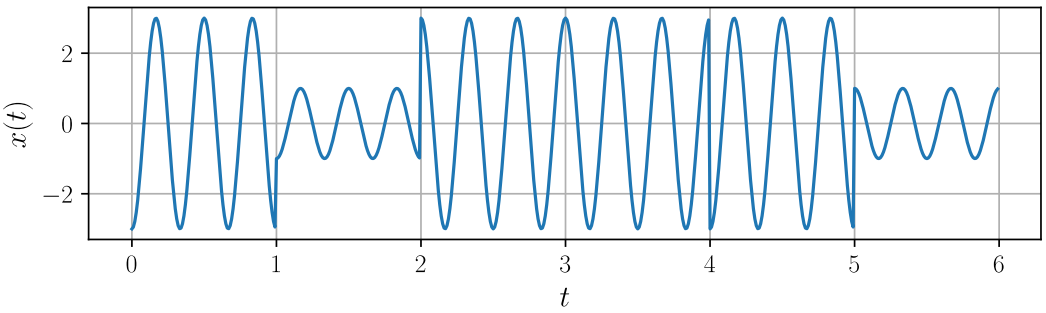


FIGURE 4 – Signal modulé, modulation d'amplitude ASK
source : vincmazet.github.io

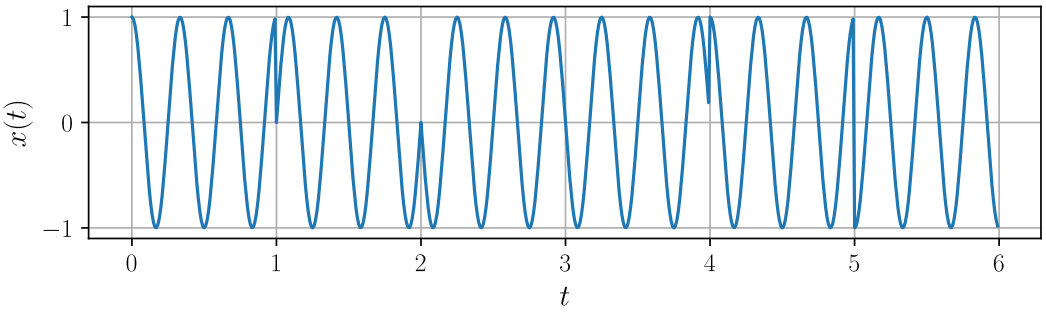


FIGURE 5 – Signal modulé, modulation de phase PSK
source : vincmazet.github.io

constituent des symboles. Par exemple, si le choix est fait d'encoder sur 2 bits, les symboles qui peuvent être envoyés sont '00', '01', '10' et '11'. Dans ce cas, il faut quatre valeurs d'amplitude, de fréquence ou de phase pour pouvoir transmettre cette information. Cette étape d'encodage permet d'augmenter le débit d'information, chaque symbole envoyé contient deux fois plus d'information. Comme ce type de codage permet de transmettre quatre symboles différents, le nom donné à la modulation sera 4ASK, 4FSK ou 4PSK selon le type choisi. Il existe aussi d'autre type de modulation, notamment la modulation QAM qui permet de transporter l'information sur deux signaux distincts, l'avantage de cette modulation est de pouvoir porter un plus grand nombre de symboles là où les modulations ASK et PSK sont plus limitées. En effet, pour pouvoir différencier les symboles en ASK, il faut une grande plage de variation de l'amplitude qui augmente avec le nombre de symboles à envoyer. L'énergie nécessaire à une telle configuration est conséquente alors que la modulation QAM concentre l'information et demande moins d'énergie pour envoyer plus de symboles.

La modulation, même si elle a pour but secondaire de limiter les interférences et les dégradations sur le signal, n'est pas suffisante. Les erreurs observées à la réception sont principalement de deux types : perte partielle de morceau de l'information ou erreur de symboles/bits ('0' à la place de '1' par exemple). Ainsi, pour réduire les erreurs de transmission, l'information est encodée lors de l'étape du **codage canal**, basé sur de la redondance, rallongeant ainsi la longueur du message transmis. Il existe deux grandes familles de codes correcteurs d'erreurs : les codes par blocs qui consistent à découper le message en blocs à chacun duquel est appliqué un code et les codes convolutifs où le message entier est codé d'un seul bloc. Parmi tous les codes existants, il y a notamment [3] :

- les codes de parité à faible densité (**LDPC**) : il s'agit de code linéaire par blocs, aujourd'hui ils servent dans plusieurs standards de communication comme le Wi-Fi, DVB-S2, le transport de données 5G ;
- les **codes polaires** : ce sont des codes linéaires par blocs, ils sont utilisés dans le standard 5G pour les canaux de contrôle ;
- les **turbo codes** : ces codes utilisent du codage convolutionnel, ils sont utilisés dans de nombreux standards comme la 3G ou la 4G ;
- il existe de nombreux autres codes comme les codes de Hamming, de Gallois, etc.

Pour des codes binaires (symbole de taille 1), les opérations qui peuvent être appliquées par les codes correcteurs d'erreur sont la multiplication et l'addition. L'opération d'addition correspond à l'opération logique *ou exclusif* et la multiplication correspond à un *et* logique, comme présenté dans la table 1. Or, le passage du signal en analogique puis par le canal a pour effet de traduire l'opération $\oplus$ en une opération plus complexe. Soient a et b (binaires) et a' et b' leur image par le canal de transmission, l'opération $\oplus$ se traduit par :

$$a \oplus b \implies 2\tanh^{-1}(\tanh(\frac{a'}{2}).\tanh(\frac{b'}{2}))$$

a	b	a⊕b	ab
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLEAU 1 – Opérations *ou exclusif* et *et* sur des symboles binaires

C'est pour cette raison que dans une chaîne de transmission, les décodeurs sont les traitements les plus gourmands en calcul derrière les encodeurs.

Dans l'ensemble de ce rapport, chacune de ses fonctions représentera un bloc de calculs indépendants. Il est possible de modéliser les interactions entre ces blocs, mais aussi de modéliser la façon dont les données sont gérées dans l'ensemble de la chaîne, ce qui peut permettre d'aider à la compréhension du problème et au développement d'algorithme d'ordonnancement.

2.2 Depuis une chaîne de communication vers un graphe d'exécution

Une chaîne de traitement de signal numérique peut être décrite par un graphe de flux de données, dont les nœuds ou blocs sont les différentes fonctions de la chaîne et les arêtes orientées représentent les dépendances de données. Les arêtes orientées représentent le transfert de données de la sortie d'un bloc A vers l'entrée d'un bloc B. Chacun des blocs sait en amont de son exécution la quantité de données dont il a besoin pour pouvoir effectuer son traitement. Ce qui correspond à un flux de données synchrone d'après la taxonomie des flux de données. La figure 6 présente un graphe de flux de données synchrone, par exemple le nœud α consomme b données et produit c données sur l'arrête le reliant au nœud β et d données sur l'arrête le reliant au nœud γ.

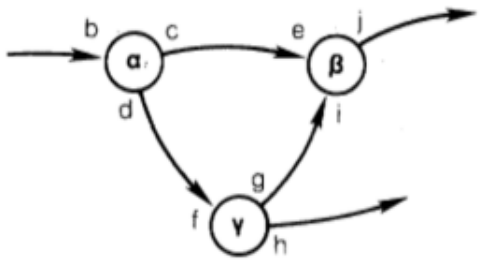


FIGURE 6 – Exemple d'un graphe de flux de données synchrone [6]

L'intérêt un flux de données synchrone est d'avoir une exécution déterministe [5], la dépendance et le parallélisme entre les tâches peut être facilement détecté. Grâce à cela, il est possible de réaliser un ordonnancement du graphe d'exécution de la chaîne. Cet ordonnancement pourrait être fait dynamiquement, mais il y a de forte chance qu'il y ait des surcoûts dû à la supervision. L'ordonnancement est donc fait de manière statique à la compilation [6]. Si l'exécution est faite sur du matériel parallèle, alors l'ordonnancement revient à faire l'assignation des nœuds aux différentes ressources. Un ordonnancement est réalisable s'il garantit qu'à l'exécution, lorsqu'un bloc est appelé, il a suffisamment de données en entrée pour pouvoir faire le traitement [11].

Ce modèle permet d'avoir un modèle du flux de données pour du traitement de signal numérique et surtout, il permet d'extraire facilement les informations sur le parallélisme possible entre tâche. L'autre avantage de ce modèle est de savoir que lorsque le graphe respecte certaines conditions, alors, il est possible de

déterminer un ordonnancement. Il existe certaines limitations à ce modèle [6], le modèle ne considère pas de signal de contrôle, c'est-à-dire qu'une chaîne peut changer de configuration à l'exécution à l'échelle d'une tâche ou d'un ensemble de tâches et le signal contrôlant ce changement n'est pas pris en compte. De plus, les tâches (nœuds) considérées sont sans état interne alors que pour des étapes de synchronisation par exemple, il est nécessaire d'avoir un état interne. D'autre part, certains types de tâches sont totalement exclus comme les tâches conditionnelles (*if*) ou les tâches exécutant des boucles (*while*).

2.3 Bibliothèques de radio logicielle

Il existe des logiciels dédiés à l'implémentation de radio logicielle et au traitement du signal. Leur objectif est de fournir un framework contenant des outils de haut niveaux pour la conception et la réalisation de chaîne logicielle.

La bibliothèque open-source la plus connue et la plus largement utilisée est **GNU Radio** [1]. Il s'agit d'une bibliothèque open source implémentée en C ++ qui peut être soit utilisée sur le matériel générique dont dispose un·e utilisateur·ice, soit utilisée dans un environnement de simulation, s'il n'y a pas de matériel par exemple. La bibliothèque fournit également une interface graphique qui facilite son utilisation pour une communauté qui est familière avec l'utilisation d'outils comme Simulink. Grâce à ses caractéristiques, cette bibliothèque peut être utilisée dans de nombreux cadres comme pour la recherche et le développement, l'enseignement ou même pour des utilisations de radioamateur. Cette bibliothèque dispose donc d'une vaste communauté contribuant à l'amélioration du projet et à une documentation fournie. GNU Radio fournit un ensemble de "blocs" correspondant à des filtres, des encodeurs, des démodulateurs, des éléments de synchronisation et tout autre élément existant dans les systèmes radio. Si par contre, un bloc n'existe pas, il est possible de le créer et de l'ajouter à la bibliothèque. Il y a également un système de connexion des blocs entre eux, c'est-à-dire comment les données sont gérées et transférées d'un bloc à l'autre. D'un point de vue ordonnancement des tâches d'une chaîne, dans la mesure du possible, un thread est attribué à chaque bloc à la compilation [13], ce qui permet d'exécuter en concurrence les différentes tâches. La force de cet ordonnancement est d'exploiter le parallélisme entre tâches et sa principale faiblesse est de faire la répartition des ressources sans distinction de la taille relative des tâches. Les tâches les plus gourmandes en calcul, typiquement un décodeur ou un encodeur, se voient attribuer autant de ressource qu'une tâche beaucoup plus rapide à être exécutée, comme un additionneur par exemple.

Il existe d'autres bibliothèques comme **StreamIt** [16] et **Array-OL** [2] dont l'usage est beaucoup moins répandu que pour GNU Radio. **StreamIt** fournit un langage et un environnement de compilation en Java pour des applications de streaming, que ce soit pour des systèmes embarqués ou des systèmes plutôt tournés hautes performances. Dans la même idée que pour GNU Radio, les fonctions de traitement radio sont implémentés sous forme de "filtre", un filtre pouvant être un modulateur, un encodeur, etc. Chaque filtre possède des canaux d'entrées et de sortie qui se comportent comme des buffers FIFO (first in, first out). Une autre caractéristique de ce langage est de proposer différentes configurations de stream : en *pipeline* (séquence de stream), en *split-join* (les streams peuvent être répartie sur plusieurs buffers ou y être concaténé) ou en FeedbackLoop, c'est-à-dire en boucle. Tous les graphes de flux de données peuvent être décrits par une composition de ses différentes structures de streams. L'ordonnancement est réalisé à la compilation de manière automatique et de façon à équilibrer les charges en faisant du parallélisme de pipeline pour des architectures multicœurs.

Array-OL est un langage spécifique au traitement intensif de signal conçu notamment pour des structures complexes de données comme des vidéos : celles-ci comportent un objet bi-dimensionnel (une image) et une dimension temporelle. Ce type d'objet implique les difficultés suivantes : un accès aux données complexes et qui diffère selon la dimension, pas de modèle de flots de données spécifique existant, un ordonnancement peu évident pour de telles applications avec notamment des contraintes temporelles d'utilisation fortes. Ce langage exploite le parallélisme de données dans une même tâche, mais aussi le parallélisme de tâche en utilisant notamment le principe de pipeline. L'ordonnancement est également fait de manière statique

et automatique, à la compilation.

AFF3CT est une boîte à outil de radio logicielle développée en C++ qui fournit à la fois une bibliothèque de codage canal et un simulateur. La partie bibliothèque permet l'exploitation en situation réelle de chaîne de communication adaptable aux besoins de l'utilisateur·ice et la partie de simulation intégrée à la bibliothèque permet de prototyper de nouvelles chaînes et de tester leurs caractéristiques. Il est par exemple possible de caractériser la résilience d'un code correcteur d'erreur vis-à-vis d'un canal donné, c'est-à-dire de savoir si le code permet au message d'être décodé correctement, malgré les erreurs introduites par le canal. Elle est actuellement utilisée dans un contexte industriel par Airbus et Thalès. AFF3CT qui utilise une bibliothèque d'instruction vectorisée MIPP [4], ce qui permet de vectoriser le traitement d'une trame, auquel cas, il s'agit de vectorisation intra-trame. Il est également possible d'avoir de la vectorisation inter-trame, c'est-à-dire d'effectuer le traitement sur deux trames en même temps par exemple. Cette opération a pour effet d'augmenter le débit, mais aussi d'augmenter la latence. Une autre façon d'optimiser le calcul est de faire du parallélisme inter tâche, certaine chaîne sont en effet parallélisées à la main dans AFF3CT en utilisant les techniques de pipeline et de duplication de tâches qui seront expliquées plus en détail dans la partie 3.2.

3 Parallélisation et ordonnancement des chaînes de communication

Cette section aborde les techniques de parallélisation et d'ordonnancement de chaîne de communication et le modèle que j'ai pu développer pour appréhender le problème. Pour ce faire, dans la partie 3.1 est défini le modèle générique d'une chaîne de communication puis la partie 3.2 présente les techniques de parallélisation qui seront utilisées pour l'ordonnancement, l'ensemble des règles spécifiques à l'ordonnancement seront définies dans la partie 3.3 et enfin la partie 3.4 présente un exemple permettant d'illustrer le modèle développé.

3.1 Modèle d'une chaîne

Chaque bloc, qui correspond un traitement spécifique (encodage, filtre, etc), se traduit par une tâche de calcul. La chaîne de traitement est donc représentée par une liste T contenant N tâches t_i, $1 \leq i \leq N$. L'ensemble des tâches se succède, la tâche t_i reçoit une trame de t_{i-1} la traite et l'envoie à t_{i+1}. La tâche t_1 correspond à la première tâche de la chaîne, à son début. La tâche t_N correspond à la dernière tâche, c'est-à-dire à la fin de la chaîne.

Chacune des tâches possède un poids, c'est-à-dire un temps de calcul qui correspond au temps de traitement séquentiel d'une trame par la tâche. Ce temps de traitement est défini par rapport à la machine et à son architecture (CPU) sur laquelle sont effectués les traitements. Le temps d'exécution séquentiel d'une tâche t_i est appelé p_i.

3.2 Techniques de parallélisation d'une chaîne

Pour paralléliser, deux techniques sont considérées : le pipeline et la duplication de tâche.

Le pipeline peut s'expliquer par la réalisation à la chaîne de mini-fraisiers, en considérant qu'il y a les étapes de montage suivantes : 1. Mettre un cercle de génoise au fond d'un moule, 2. Disposer les fraises, 3. Ajouter la crème, 4. Recouvrir avec un 2e cercle de génoise. Il ne peut y avoir qu'un seul gâteau par poste de montage, une fois qu'un cercle de génoise a été disposé au fond d'un moule, ce gâteau-là est amené au poste de disposition des fraises 2. et laisse la place libre à un nouveau gâteau en 1. Ainsi, lorsque le premier gâteau est enfin monté à l'étape 4., il y a trois autres gâteaux en train d'être réalisés dans la chaîne. Si l'on considère que la réalisation de ces étapes demande le même temps T, alors lorsque la chaîne est établie, il y a un gâteau fait tous les T. S'il n'y avait qu'une seule personne pour faire ce gâteau, elle devrait faire chaque étape à la suite et il lui faudrait $4 \times T$ pour faire le montage d'un seul gâteau. Les figures 7a et 7b

montrent ces deux façons de faire pour le traitement d'instructions avec en abscisse les différents cycles d'horloges et en ordonnée les instructions à réaliser.

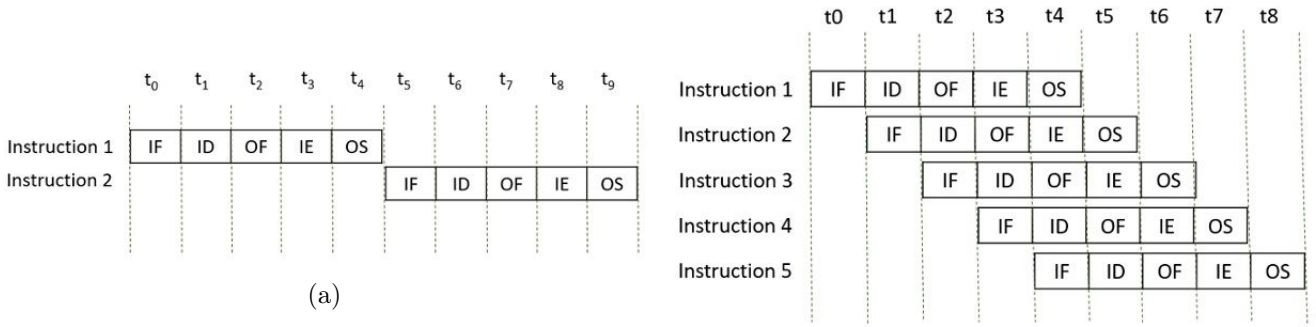


FIGURE 7 – Exécution séquentielle d'instruction (a) et pipeline d'instruction (b), *source : binaryterms.com.*

La figure 7a montre que pour traiter les instructions 1 et 2, il faut en tout 8 cycles d'horloge. Alors que pour le pipeline, figure 7b, après 8 cycles d'horloge, cinq instructions ont été réalisées. Chaque morceau d'instruction, appelé étage, est traité par une ressource différente pendant un cycle d'horloge. Pour le pipeline présenté à la figure 7b, il faut cinq ressources pour la réaliser. Dans la suite de ce rapport, l'utilisation du pipeline est faite pour du streaming, c'est-à-dire un flot continu de données (trames). Alors, en régime permanent, l'amorce du pipeline devient négligeable. De plus, il est possible d'avoir des étages dont le temps de traitement sont différents. Dans ce cas, c'est l'étage avec le temps de traitement le plus long M qui rythmera le pipeline. Même si un étage a un temps d'exécution $T < M$, alors la ressource sera inactive pendant $M - T$ le temps que le cycle se termine. Il existe des **algorithmes optimaux de pipeline** en équilibrage de charge [14] applicables dans certaines configurations de chaîne. La partie 4.2 présente un de ces algorithmes.

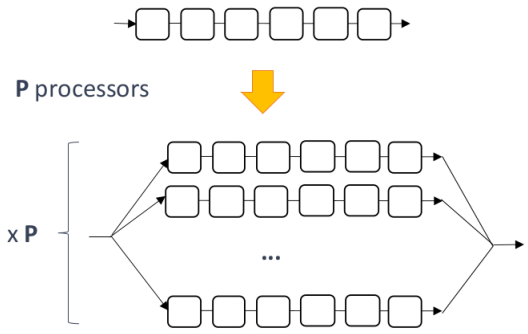


FIGURE 8 – Parallélisation d'une chaîne sur P processeurs.

La duplication de tâche et/ou d'étage est une autre façon de paralléliser. Une tâche (étage) est un traitement (série de) à effectuer sur des données. Il est possible d'appliquer le même traitement à plusieurs données simultanément. Cela signifie que plusieurs ressources effectuent le même traitement sur un ensemble de trames. La figure 8 présente la parallélisation par duplication sur P ressources d'une séquence de tâches (ou chaîne). Le fait de **paralléliser en dupliquant est optimal** si la nature des tâches le permettent. La partie 3.3 présente les différentes natures existantes de tâches et présente dans quel cas ce type de parallélisation est optimal.

3.3 Règles du problème

L'objectif est de trouver une façon d'organiser l'exécution sur les ressources disponibles, c'est-à-dire de trouver un ordonnancement, en ayant le débit le plus élevé et la latence la plus faible d'exécution de trames. Ci-dessous sont définies plusieurs règles permettant de définir le modèle :

1. Les ressources considérées sont **homogènes et uniformes**, alors le temps de calcul d'une tâche est indépendant de la (des) ressource(s) qui lui est (sont) allouée(s).
2. Chaque tâche, selon sa nature, peut être soit exécutable **séquentiellement**, comme c'est le cas pour des blocs de synchronisation de trame, soit l'exécution est faite **en parallèle**. Dans le cas d'une chaîne avec des tâches uniquement séquentielles, c'est la technique de parallélisation par **pipeline** qui est optimale (voir 3.2). Dans le cas d'une chaîne contenant uniquement des tâches parallélisables, c'est la parallélisation par **duplication** qui est optimale (voir 3.2).
3. Une liste de tâches peut être découpée en plusieurs sous-chaînes contiguës. Une sous-chaîne $s = [t_i,\ t_{i+1},\ ...,\ t_j]$ est appelée **étage**. L'ensemble des tâches d'un étage sont exécutées pour une trame avant de passer à la trame suivante. Les tâches d'un étage sont exécutées par les mêmes ressources et ne sont pas partagées avec d'autres étages.
4. Les opérations considérées pour réaliser l'ordonnancement sont : la **pipeline** d'une sous-chaîne (étage) ou de la chaîne entière et la **duplication** d'une sous-chaîne (étage) ou de la chaîne entière.
5. Un étage contenant au moins une tâche séquentielle est un étage séquentiel, aucune tâche ne peut être parallélisée dans un étage séquentiel. Autrement dit, **un étage est séquentiel tant qu'il contient au moins une tâche séquentielle**.
6. Le temps d'exécution séquentiel d'un étage $s = [t_i,\ t_{i+1},\ ...,\ t_j]$ correspond au temps qu'il faut pour exécuter une à une les tâches de l'étage sans duplication :

$$\mathscr{T}_{seq}(s) = \sum_{k=i}^{j} p_k$$

7. Les **surcoût** de communication, de synchronisation d'étage (lors d'un pipeline), de synchronisation d'un étage multi-threadé (mono-threadé) vers un étage mono-threadé (multi-threadé) **ne sont pas pris en compte**.
8. Le temps d'exécution d'un étage s parallélisé sur R ressources est le temps d'exécution séquentielle de l'étage divisé par le nombre de ressources :

$$\mathscr{T}_{par}(s) = \frac{\mathscr{T}_{seq}(s)}{R}$$

9. Soit $s = [t_i,\ t_{i+1},\ ...,\ t_j]$ un étage d'une chaîne, son temps d'exécution apparent est noté $\mathscr{T}(s)$, il est, soit séquentiel, soit parallèle :

$$\mathscr{T}(s) = \mathscr{T}_{par}(s) \text{ ou } \mathscr{T}_{seq}(s)$$

Pour réaliser les objectifs et/ou mesurer leur avancement, il faut définir plusieurs variables :

- N : nombre de tâches dans la chaîne ;
- P : nombre total de ressources disponibles ;
- R : nombre de ressources encore disponibles (non allouées) ;
- nE : nombre d'étages ;
- M : temps d'exécution maximal d'entre tous les étages ;
- $D = 1/M$: débit d'exécution des tâches ;

- $L = (nE - 1) \times M + \mathscr{T}_{nE}$: temps de traitement d'une trame parcourant toute la chaîne, le temps d'exécution du dernier étage est compté comme tel, en effet, la trame peut être directement envoyée à la sortie, elle n'a pas besoin d'attendre la fin de l'exécution de tous les autres étages ;
- E : efficacité, rapport entre le temps d'exécution séquentielle de la chaîne $\mathscr{T}_{seq,tot}$ et la latence L :

$$E = \frac{\mathscr{T}_{seq,tot}}{L}$$

Dans la suite du rapport, la mesure de la qualité d'un ordonnancement est faite selon les trois métriques D, L et E : D doit être le plus élevé possible, L le plus faible possible à débit équivalent et E le plus proche de 1 à débit et latence équivalente. Il y a un ordre de priorité entre ces trois métriques, c'est le débit qu'il faut maximiser en priorité, puis la latence et enfin E. Cette mesure de l'ordonnancement est noté DLE et il vaut $DLE = (D,\ 1/L,\ E)$ qui est la variable à maximiser.

3.4 Exemple

Afin de mieux comprendre comment fonctionne le modèle, un exemple est détaillé dans cette sous-partie. Soit une chaîne de cinq tâches dont chaque temps d'exécution est considéré comme entier afin de simplifier le problème, aussi la taille de tâche la plus petite possible atteignable sera 1. Les différentes tâches sont décrites comme suit :

— t_1 : parallélisable, $p_1 = 5$;
— t_2 : parallélisable, $p_2 = 3$;
— t_3 : séquentielle, $p_3 = 1$;
— t_4 : parallélisable, $p_4 = 1$;
— t_5 : parallélisable, $p_5 = 2$.

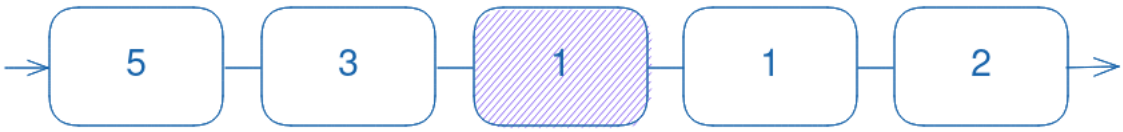
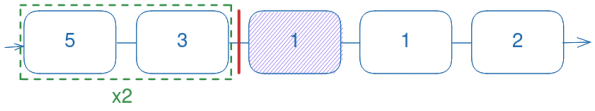


FIGURE 9 – Exemple de chaîne mêlant tâches séquentielles (hachurée) et parallélisables (fond blanc).

La figure 9 présente cette chaîne avec ses caractéristiques, les tâches en blancs sont les tâches parallélisables et les tâches hachurées sont les tâches séquentielles. La contrainte principale est d'avoir le débit D le plus élevé possible, ce qui signifie d'avoir le temps maximal d'étage M le plus faible possible en dupliquant et pipelinant. La seconde contrainte est de ne pas avoir une latence de traitement de trame L trop grande, ce qui signifie qu'il faut limiter le nombre d'étages de pipeline, ce qui peut s'opposer à ce qui doit être fait pour augmenter le débit D. L'exemple est étudié pour des nombres différents de ressources $P = 1, 3, 4, 12$, l'ensemble des métriques pour les configurations étudiées sont rassemblées dans le tableau 2.

P=1 :
La chaîne est exécutée tâche par tâche de manière séquentielle. Ici $M = 5 + 3 + 1 + 1 + 2 = 12$ donc $D = 1/12$, aussi $L = M \times 1 = 12$ et $E = 1$.

P=3 :
La tâche séquentielle du milieu oblige à vouloir isoler les deux premières tâches dans un même étage. Cet étage étant plus conséquent que les autres, est dupliqué, c'est-à-dire exécuté sur deux ressources.

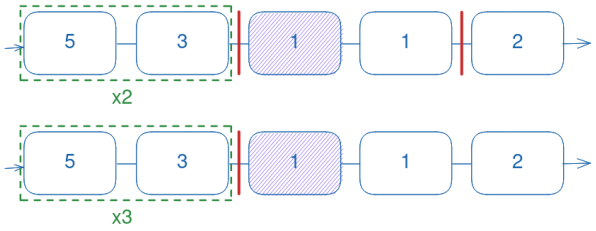


Dans ce cas, l'étage le plus gros parmi les deux possibles s'exécute en 4, donc $M = 4$ donc $D = 1/4$ et

$L = M \times 2 = 8$ et $E = 1$. Par rapport au cas avec $P = 1$, le débit et la latence sont meilleures.

P=4 :

A priori, le fait de rajouter une ressource pour l'exécution de la chaîne amène à penser que les métriques seront meilleures. Étant donné que latence et le débit sont reliés au temps maximal M, l'idée est de faire diminuer le poids de l'étage maximal. Cependant, n'est pas évident ici de savoir quelle opération effectuer puisque pour $P = 3$ les deux étages existants sont de même poids.



Les deux cas sont testés :

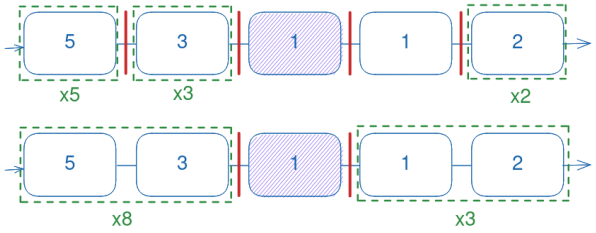
- 1er cas : il y a un étage supplémentaire, le débit $D = 1/4$ est toujours le même, mais la latence et l'efficacité changent $L = M \times 3 = 12$ et $E = (4 + 2 + 2)/L = 2/3 = 0,66$;
- 2e cas : l'idée est de diminuer ici l'étage parallélisable en le dupliquant à nouveau et en gardant le même nombre d'étages que pour $P = 3$. Le premier étage a un poids de $(5 + 3)/3 = 2,22$ et le deuxième étage a un poids de 4. Dans ce cas $M = 4$ et le débit est donc toujours le même $D = 1/4$ mais la latence change $L = M \times 2 = 8$ et $E = (2,22 + 4)/8 = 0,78$.

Dans les deux cas, le débit est le même qu'avec une ressource de moins, quant à la latence, elle, est soit identique, soit pire. Cet exemple, dans la configuration avec $P = 4$, montre d'une part qu'il n'est pas évident de savoir quelle opération appliquer pour pouvoir augmenter le débit, et d'autre part, il est possible de se retrouver avec des métriques moins bonnes qu'avec moins de ressources.

Meilleure configuration ?

La meilleure configuration possible est celle où chaque étage a un poids inférieur ou égal au poids de l'étage séquentiel, ici de 1. Il existe plusieurs configurations pour y arriver.

Une première idée serait d'isoler chaque tâche et de dupliquer les tâches parallèles jusqu'à avoir un poids égal à 1. Cela donne la première configuration obtenue pour $P = 12$ ressources. Alors les métriques donnent $M = 1$ et $D = 1$ et comme il y a cinq étages $L = M \times 5 = 5$ et $E = 1$.



Une deuxième idée est de d'isoler les étages séquentiels et parallélisables puis de dupliquer ces étages le plus possible. Cette deuxième configuration permet de limiter la latence puisque le nombre d'étages est plus faible avec le même débit : $D = 1$, $L = 3$ et $E = 1$.

Il est difficile d'évaluer, à la main, quelle est la meilleure configuration possible. Il est possible de rater une façon de faire et de se retrouver loin de l'optimal.

Ainsi, une chaîne est assimilée à une liste de tâches possédant leurs propres caractéristiques. Étant donné les techniques de parallélisations envisagées (pipeline et duplication), les règles définies permettent d'établir le comportement de ces tâches vis-à-vis de ces techniques-là. Ce modèle permet maintenant d'envisager le développement d'heuristiques d'ordonnancement.

Nombre de ressources P	Temps maximal d'étage M	Débit D	Latence L	Efficacité E
1	12	0,08	12	1
3	4	0,25	8	1
4	4 4	0,25 0,25	12 8	0,66 0,78
12	1 1	1 1	5 3	1 1

TABLEAU 2 – Métriques de la chaîne exemple selon différent nombre de ressources.

4 Algorithmes de parallélisation et d'ordonnancement des chaînes de communication

Il serait imaginable de faire un algorithme décrivant toutes les solutions possibles et de ne garder que la meilleure, mais cette solution est fortement combinatoire, c'est pour cette raison que le choix se porte sur le développement d'heuristiques. Cette partie présente donc différentes heuristiques que j'ai explorées et qui cherchent à s'approcher d'une solution optimale. La partie 4.1 présente une heuristique se basant sur l'aléatoire, la partie 4.2 présente une manière optimale de faire le pipeline. Enfin, la partie 4.3 expose une heuristique que j'ai développée associant les deux techniques de parallélismes décrites plus haut : pipeline et duplication.

4.1 Algorithme aléatoire

Dans un premier temps, j'ai réalisé un algorithme simple comme point de référence. Dans cette sous-partie est tout d'abord expliquée l'idée de l'algorithme, puis son implémentation et enfin son application à des exemples.

L'idée de l'algorithme est de générer un certain nombre de configurations de pipeline, avec un nombre d'étages P connus à l'avance, de manière aléatoire et d'en prendre la meilleure en fonction de son évaluation DLE. Le pseudo code ci-dessous présente les différentes étapes du code :

```
pip = pipeline_aleatoire(P, liste_taches)
meilleur_pip = pip
DLE = mesure_DLE(pip)
for i in range(nb_iterations):
    pip = pipeline_aleatoire(P, liste_taches)
    if mesure_DLE(pip) > DLE:
        meilleure_pip = pip
        DLE = mesure_DLE(pip)
```

L'objet *pip* est une liste d'étages de la chaîne. La fonction *pipeline_aleatoire* prend en entrée le nombre d'étages voulus et la liste de tâche de la chaîne et tire au hasard $P-1$ barrières comprise entre 1 et $N-1$ qui font office de séparations entre les différents étages de pipeline. La fonction *mesure_DLE* quant à elle prend en entrée le pipeline et permet d'évaluer la variable DLE du pipeline considérée. Le nombre d'itérations, ici $nb_iterations$, est pris arbitrairement.

Pour l'implémentation en Python, plusieurs objets ont été définis :

- **task** : objet représentant une tâche avec comme variables interne son nom (*name*), son poids (*time*) et un booléen indiquant s'il est parallélisable ou non (*is_dup*, est vrai si tâche parallélisable) ;
- **stage** : objet représentant un étage de pipeline, contient des tâches, peut être dupliqué s'il ne contient que des tâches parallèles et possède comme variables internes : une liste de tâches (*list_tasks*), son temps d'exécution (*time*), son temps séquentiel d'exécution sans duplication (*seq_time*), son statut de duplication (*is_dup*), le nombre de tâches non duplicables (*non_dup_tasks*) ;

- **ordonnancement** : objet représentant une façon dont la chaîne est découpée en étages et le nombre de ressources utilisées. Cet objet contient donc des étages et possède une fonction d'évaluation de l'ordonnancement renvoyant DLE, valeur à maximiser.

Il y a plusieurs inconvénients à utiliser cet ordonnanceur, notamment qu'il n'y a pas de certitude de tomber sur l'optimal et qu'il n'exploite qu'un seul type de parallélisme (pipeline). Cependant, il a permis de définir et d'implémenter des objets qui seront réutilisés pour le développement d'autres algorithmes.

4.2 Algorithme de pipeline optimal pour équilibrage de charge

Il existe dans la littérature un algorithme de pipeline optimal de chaîne dîtes 1D [14], dans la suite du rapport cet algorithme est désigné par l'appellation *Pinar1D*. Cet algorithme prend en entrée une chaîne $T = [t_1,\ t_2,\ ...,\ t_N]$ de N éléments de poids divers $w = [p_1,\ p_2,\ ...,\ p_N]$ et le nombre de ressources P. Il donne en sortie un découpage de la chaîne en au plus P étages différents tout en minimisant leur poids maximal, noté B. Par rapport au modèle défini en 2.5.1, ce B correspond à M le temps d'exécution maximal d'entre tous les étages.

La découpe en étages de pipeline repose sur le fait de trouver la valeur optimale de B. Soit W un tableau contenant les poids cumulatifs de chaque tâche avec :

$$W_i = \sum_{k=1}^{i} p_k, \quad i \leq N$$

La valeur idéale de B, notée B^* est $B^* = \frac{W_N}{P}$. Or cette valeur peut ne pas être atteignable si : $\min_k(p_k) > B^*$, c'est-à-dire si le poids minimal des tâches est supérieur à B^*. La première partie de l'algorithme consiste donc à trouver une valeur de B, pour lequel il existe un découpage possible en au plus P étages. Cette valeur, si elle existe, est appelée B_{opt}. Une fois ce B_{opt} trouvé, il est possible de faire le découpage. Pour cela, la chaîne est parcourue de gauche (début) à droite (fin) et l'algorithme fait des paquets de tâches dont le poids total fait au plus B_{opt}.

Pour comprendre l'ensemble de l'algorithme, il faut d'abord décrire la fonction EXACT_BISECT, qui permet de trouver B_{opt} :

```
# EXACT_BISECT
B_star = W[N]/P
B1 = B_star
UB = B_star + max(w)
utile = True

while utile is True:
    L = [0 for i in range(P)]
    Bt = int((B2 + B1)/2)
    if RPROBE(Bt, P, W, L, N) is True:
        if B2 = Bt:
            utile = False
        else:
            B2 = Bt
    else:
        if B1 = Bt:
            utile = False
        else:
            B1 = Bt
B_opt = B2
```

La fonction RPROBE renvoie *True* si le découpage est possible et *False* sinon en fonction de B, du nombre de ressources P et du nombre de tâches N. Cette fonction fait une recherche binaire sur la chaîne pour

faire des paquets avec le maximum de tâches et dont le poids total ne dépasse pas B. Une fois ce travail effectué, s'il y a P ou moins de paquets (étages), alors le découpage est possible pour B, sinon, il y a plus de P paquets et le découpage n'est pas possible.
La fonction EXACT_BISECT prend en entrée le tableau de la somme cumulative des poids W, la liste des poids $w = [p_1,\ p_2,\ ..,\ p_N]$, le nombre de tâches de la chaîne N et le nombre de ressources P. Elle donne en sortie le B optimal, noté B_{opt}, en utilisant aussi le principe de recherche binaire en testant plusieurs valeurs de B. B ne peut être plus faible que B^* et il est borné par ce B^* plus le poids de la tâche maximal, ce qui donne :

$$B^* \leq B \leq B^* + \max_k(p_k)$$

Pour un B trouvé, la fonction RPROBE est utilisée pour déterminer s'il y a un découpage possible. Si c'est le cas, l'opération est recommencée pour un B plus faible. S'il n'y a pas de découpage possible pour le B testé, alors B_{opt} est le B de l'itération d'avant.

La structure de l'algorithme, dont les fonctions sont explicitées plus bas, est donné par le pseudo code ci-dessous :

```
# PINAR 1D
B_opt = EXACT_BISECT(W, w, N, P)
barrieres = PART(B_opt, P, W, N)
pipeline = []
b_min = 0
i = 1
b_max = barrieres[i]
while b_max != b_min faire:
    for k in range(b_min, b_max):
        ajouter a tmp_etage la k-ieme tache
    ajouter tmp_etage a pipeline
    b_min = b_max
    i+=1
    b_max = barrieres[i]
```

La fonction PART renvoie les barrières, c'est-à-dire à quels endroits il faut découper pour un B donné et prend en entrées : B, la liste de la somme cumulative des poids W, le nombre de ressources P et le nombre de tâches N.

À titre d'illustration, cet algorithme est appliqué à l'exemple de la chaîne donnée en figure 10. Il n'est

FIGURE 10 – Exemple de chaîne.

pas intéressant de considérer des cas où il y a plus de ressources que de tâches, le cas limite étant d'avoir autant de tâches que de ressources, auquel cas, chaque tâche se voit attribuer une ressource. Le cas où il n'y a qu'une seule ressource implique de ne pas pouvoir faire de parallélisme puisque la chaîne sera exécutée séquentiellement. Les cas les plus complexes sont donc pour $1 < P < N$ Dans l'exemple, il y a cinq tâches, l'algorithme est envisagé pour $P = 3$ et $P = 4$. Puisque l'opération de duplication n'est pas envisagée, la nature des tâches n'est pas importante (séquentielle ou parallèle).
Pour la chaîne considérée, le tableau cumulatif des poids W est : $W = [5, 6, 9, 10, 12]$. La limite supérieure de B est notée UB et sa limite inférieure est notée LB.

- $P = 3$: dans ce cas $B^* = 4$ et il est possible d'encadrer B par $4 \leq B \leq 9$ alors $UB = 9$ et $LB = 4$. L'obtention de B_{opt} s'obtient avec la fonction EXACT_BISECT en plusieurs itérations :

— Itération 1 : pour $B = |(4+9)/2| = 6$, il est possible de trouver un découpage d'au plus trois paquets dont la taille maximale vaut 6 (voir figure 11). Dans ce cas, la limite maximale est abaissée à 6 : $UB = 6$.

— Itération 2 : pour $B = |(4+6)/2| = 5$, il est possible de trouver un découpage d'au plus trois paquets dont la taille maximale vaut 5 (voir figure 12). Dans ce cas, la limite maximale est abaissée à 5 : $UB = 5$.

— Itération 3 : pour $B = |(4+5)/2| = 4$, dans ce cas $B = LB$ donc l'algorithme s'arrête (idem si $B = UB$).

L'optimal trouvé dans ce cas est $B_{opt} = 5$, ce qui donne le découpage présenté à la figure 12.

- $P = 4$: dans ce cas $B^* = 3$ et il est possible d'encadrer B par $3 \leq B \leq 8$ alors $UB = 8$ et $LB = 3$. L'obtention de B_{opt} s'obtient avec la fonction EXACT_BISECT en plusieurs itérations :

  — Itération 1 : pour $B = |(3+8)/2| = 5$, il est possible de trouver un découpage d'au plus trois paquets dont la taille maximale vaut 5 (voir figure 12). Dans ce cas, la limite maximale est abaissée à 5 : $UB = 5$.

  — Itération 2 : pour $B = |(3+5)/2| = 4$, il n'est pas possible de trouver un découpage pour ce B là, l'algorithme s'arrête.

  L'optimal trouvé dans ce cas est $B_{opt} = 5$, ce qui donne le découpage présenté à la figure 12.

FIGURE 11 – Découpage de la chaîne avec Pinar1D pour B=6 et P=3.

FIGURE 12 – Découpage de la chaîne avec Pinar1D pour B=5 et P>=3.

Dans les deux cas, pour $P = 3$ et pour $P = 4$, le découpage trouvé est le même et ce serait le même aussi pour une configuration avec plus de ressources puisque la tâche de poids 5 est le goulot d'étranglement.

4.3 Ordonnanceur mêlant duplication et découpage optimal de chaîne

Cette partie constitue l'essentiel de ma contribution. Elle présente l'heuristique développée permettant d'améliorer les métriques DLE. Par rapport à Pinar1D, cette heuristique possède l'avantage de pouvoir faire diminuer le temps d'exécution des étages parallèles tout en exploitant le parallélisme de pipeline. Pour pouvoir expliquer son fonctionnement, il faut d'abord définir certaines opérations sur les chaînes en 4.3.1 et les sous-parties suivantes 4.3.2, 4.3.3, 4.3.4 et 4.3.5 décrivent les différentes étapes de cet algorithme.

4.3.1 Définition d'opérations élémentaires dans une chaîne

Les opérations supplémentaires définies dans cette sous-partie sont la **duplication**, le **vol de tâche** et la **fusion** d'étage.

La **duplication** d'étage consiste à répartir le traitement des trames de cet étage sur une ressource supplémentaire. Ce qui a pour effet de diminuer encore le temps d'exécution de cet étage. Soit un étage s à

qui sont déjà allouées R ressources ($R \geq 1$), après duplication, le temps d'exécution apparent est diminué, car exécuté sur $R+1$ et il vaut :

$$\mathscr{T}(s) = \frac{\mathscr{T}_{seq}(s)}{R+1}$$

Cette opération améliore le débit et la latence si elle est appliquée à l'étage dont le temps d'exécution est maximal (M).

L'opération **pipeline** utilise l'algorithme Pinar1D (voir 4.2) sur un étage (sous-chaîne). L'objectif est de faire diminuer le temps d'exécution maximal des étages M afin d'améliorer le débit de traitement de la chaîne.

Le vol de tâche est une opération décrite par la figure 13. Pour une chaîne donnée de cinq tâches, découpée ici en trois étages distincts, l'idée est d'équilibrer les étages entre eux pour mieux repartir les tâches entre les ressources. Par exemple, si le 2e étage contenant t_3 veut voler une tâche. Il peut soit voler t_2 à gauche, soit voler t_4 à droite.

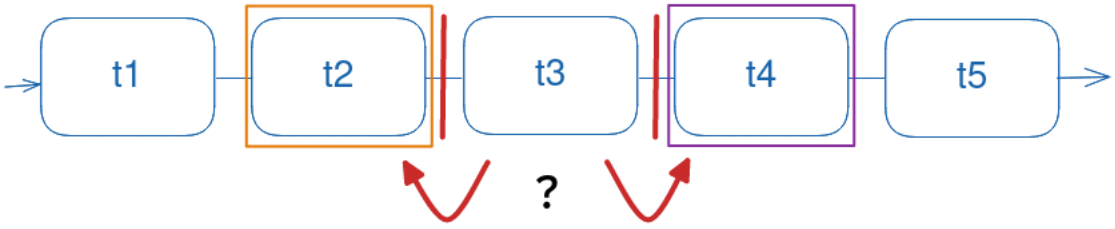


FIGURE 13 – Opération de vol de tâche.

Enfin, l'opération de **fusion** consiste à mettre dans un seul étage les tâches de deux étages consécutifs. Par exemple, sur la figure 14, le 2e et 3e étages sont fusionnés pour ne former qu'un seul et même étage. Si un des deux étages est séquentiel, alors le nouvel étage obtenu par fusion est nécessairement séquentiel (voir règle 5 du 3.3).

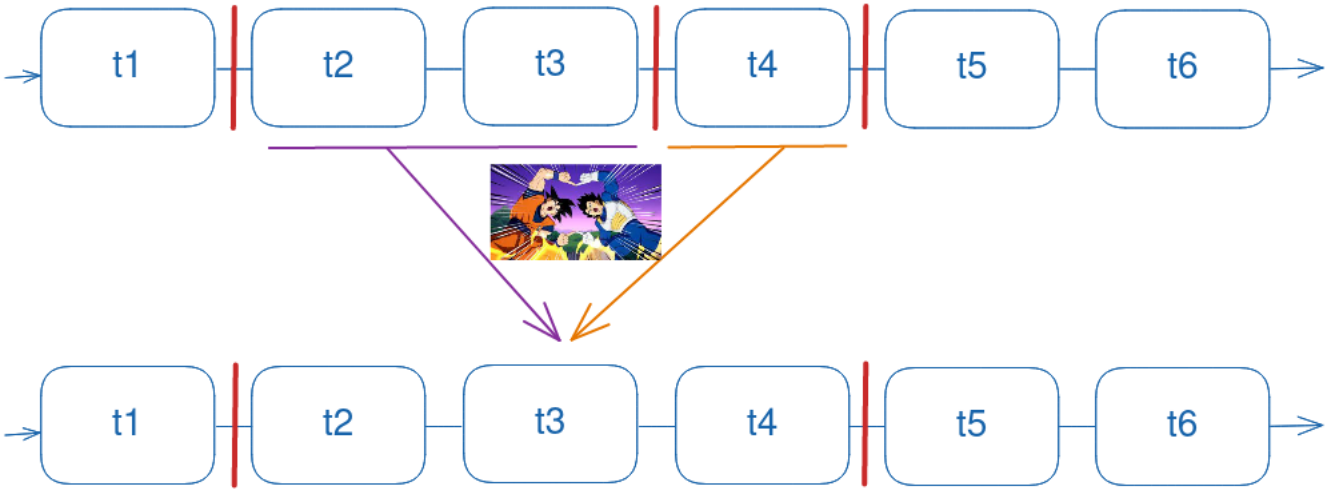


FIGURE 14 – Opération de fusion d'étages.

Le pseudo-code ci-dessous décrit la structure de l'algorithme. Celui-ci prend en entrée une chaîne, sous la forme d'une liste T de tâches et un nombre de processeurs P. En sortie, l'heuristique renvoie une liste d'étages contenant l'ensemble des tâches de T. Chacun de ces étages contient l'information s'il est dupliqué

ou non et si c'est le cas, de combien de ressources il dispose pour cela. Le nombre de ressources disponibles, c'est-à-dire non utilisées pour le traitement de la chaîne, est noté R, cette valeur peut évoluer à chaque étape de l'algorithme.

```
R, S = init(T)
if R < 0:
    R = equilibrage(S, P)
elif R > 0:
    R = distribution(S, P)
reconfiguration(S, P)
latence_reduction(S, P)
```

Les fonctions *init*, *equilibrage*, *distribution*, *reconfiguration* et *latence_reduction* sont expliquées dans les sous-parties suivantes. Pour expliquer ces différentes étapes, la chaîne exemple de la figure 15 sera gardée le long des sous-parties, les tâches parallèles sont en blanc et les séquentielles hachurées en bleu.

FIGURE 15 – Exemple de chaîne d'entrée de l'heuristique.

4.3.2 Initialisation

L'initialisation, fonction ***init***, consiste à séparer dans la chaîne les tâches parallèles des tâches séquentielles. Cette fonction prend donc en entrée la liste de tâches T et renvoie en sortie une liste d'étages S et une valeur de R. Les étages contenus dans S sont constitués de tâches de même nature : soit uniquement des tâches parallèles, soit uniquement des tâches séquentielles. Ainsi la liste d'étages obtenue est une alternance d'étages séquentiel et parallèle, comme sur la figure 16.

FIGURE 16 – Initialisation : séparation des tâches parallèles et séquentielles.

4.3.3 Pré-traitements

Une fois l'étape d'initialisation faite, il y a trois cas possibles :

- $R < 0$: il y a plus d'étages que de ressources, dans ce cas l'étape d'*équilibrage* est appliquée ;
- $R = 0$: il y a autant d'étages que de ressources, il n'y a pas de pré-traitement, l'étape suivante est celle de *reconfiguration* ;
- $R > 0$: certaines ressources ne sont pas allouées, il est donc possible de faire une première *distribution* des ressources dans le but d'améliorer les métriques.

Cette sous-partie traite des cas où $R < 0$ et $R > 0$.

Après le découpage fait à l'initialisation, il est possible d'avoir des configurations où il y a plus d'étages que de ressources, c'est-à-dire que $R < 0$. Dans ce cas, il faut prévoir une étape de rééquilibrage avec la fonction ***equilibrage***. Dans cette étape, il faut réaliser autant de fusions (cf. 4.3.1) que de ressources manquantes. Pour choisir chaque fusion, toutes les possibilités sont testées, la configuration minimisant M, le temps maximal d'étages de S est retenue. Le code ci-dessous présente la façon dont est fait le choix de fusion d'étages à chaque *gain* d'une ressource.

```
# EQUILIBRAGE
while R < 0:
    M = max_time(S)
    N = len(S)
    index_fusion = (0, 1)
    ## Choix de la fusion
    for k in range(N-1):
        S_tmp = fusion(S, S[k], S[k+1])
        M_tmp = max_time(S_tmp)
        if M_tmp > M:
            M = M_tmp
            index_fusion = (k, k+1)
    S = fusion(S, S[index_fusion[0]], S[index_fusion[1]])
    R += 1
```

Cet algorithme termine bien, en effet à chaque itération dans la boucle while, il y a fusion de deux étages, ce qui a pour effet de libérer une ressource, c'est-à-dire qu'à chaque itération R augmente de 1. La condition $R = 0$ est donc atteinte.

Si par contre $R > 0$, alors il est possible d'effectuer une première ***distribution*** des ressources. Comme le but est de minimiser M, si l'étage le plus long (temps apparent $\mathscr{T}$ le plus grand) est parallèle, alors cet étage sera dupliqué une fois, si l'étage le plus long est séquentiel, alors l'algorithme Pinar1D est appliqué, via la fonction *pipeline*, sur l'étage en question. Ces opérations sont appliquées jusqu'à ce qu'il n'y ait plus de ressources disponibles ($R = 0$) ou que le nombre de ressources disponible R reste constant, ce qui peut arriver en essayant de pipeliner un étage qui est déjà découpé de façon optimale selon Pinar1D.

```
# DISTRIBUTION
prev_R = 0
while R > 0 and R != prev_R:
    N = len(S)
    T_max = T_0
    index = 0
    # Trouver l'etage le plus long
    for k in range(N):
        T = T_k #temps apparent du kieme etage
        if T > T_max:
            T_max = T
            index = k

    # Duplication si l'etage le plus long est parallele
    if est_parallele(S[index]) = True:
        duplique(S, index)
        prev_R = R
        R = R - 1

    # Pipeline si l'etage le plus long est sequentiel
    else:
        proc = pipeline(S, index, 2)
        prev_R = R
        R = R - (proc-1)
```

Il est alors possible de se retrouver avec la configuration de la chaîne présentée en figure 17. Le premier étage a été découpé en deux étages distincts et l'étage parallèle a été dupliqué : deux ressources sont maintenant allouées à cet étage comme signalé en vert sur la figure. Cette étape termine bien, car à chaque passage dans la boucle while, soit R diminue de 1 et donc en décroissant strictement, il finira par être nul, soit R reste constant, ce qui a pour effet de sortir de la boucle également.

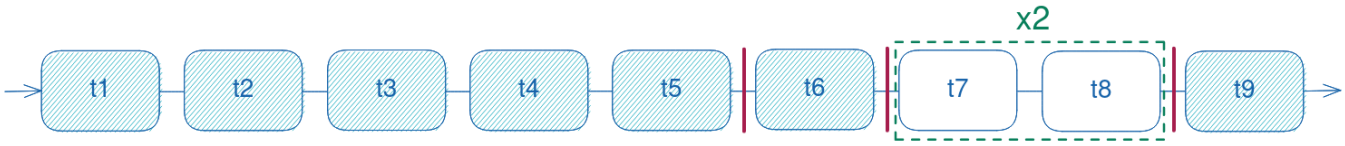


FIGURE 17 – Première distribution sur la chaîne exemple.

4.3.4 Étape de reconfiguration

L'étape de reconfiguration est appliquée lorsque toutes les ressources ont été allouées aux étages de S et donc que $R = 0$. Le but de cette étape est d'appliquer des opérations permettant de libérer une ressource, avec de la *fusion* notamment, pour pouvoir la redistribuer ensuite et améliorer les métriques DLE. L'opération de *vol* est également tentée afin d'améliorer la latence. Le pseudo-code ci-dessous présente cette étape :

```
# RECONFIGURATION
DLE_0 = evalue_DLE(S) #metriques apres init et pre-traitements
utile = True
while utile = True:
    Trouver la meilleure fusion donnant DLE_fusion
    Trouver le meilleur vol donnant DLE_vol
    #Appliquer la meilleure operation
    DLE = max(DLE_fusion, DLE_vol)
    if DLE > DLE_0 and R>=0:
        if DLE = DLE_vol:
            Appliquer meilleur vol
        else:
            Appliquer meilleure fusion
            R = distribution(S, P)
        DLE_0 = DLE
    else:
        utile = False
```

La meilleure fusion et le meilleur vol sont déterminés par recherche exhaustive de la même façon que pour l'étape d'équilibrage. Comme l'opération de *fusion* permet de libérer une ressource, cette ressource est immédiatement réinvestie pour pouvoir minimiser M en appliquant la fonction *distribution*.

4.3.5 Étape de réduction de latence

L'étape de réduction de latence intervient après l'étape de reconfiguration. Il est possible d'avoir une chaîne avec un débit acceptable, mais avec un grand nombre d'étages, ce qui a pour effet d'avoir une latence L importante puisque L est proportionnelle au nombre d'étages (voir 3.3). Cette étape a pour but de réarranger les étages qui ne sont pas dominants et dont le temps apparent est inférieur à M, pour diminuer leur nombre. La figure 18 présente la chaîne exemple après l'étape de reconfiguration qui comporte alors cinq étages. Le soleil indique l'étage dominant.

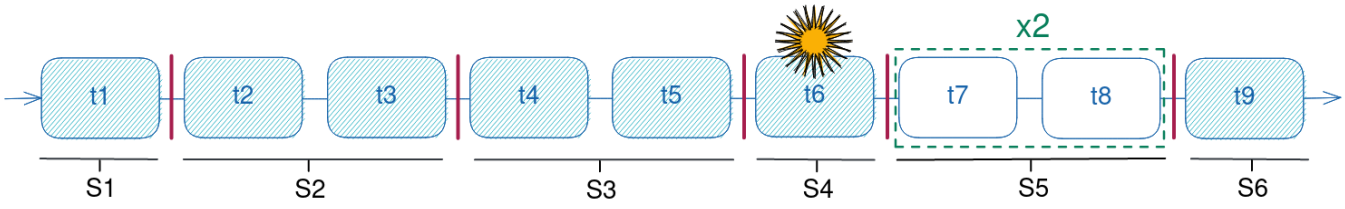


FIGURE 18 – Chaîne après reconfiguration et avant l'étape de réduction de latence.

La chaîne est parcourue de gauche à droite, pour deux étages successifs, l'étage potentiel résultant de leur fusion est comparé à l'étage dominant, s'il possède un temps apparent plus faible que l'étage dominant alors la fusion est effectuée. Par exemple, par rapport à la figure 18, cela reviendrait à faire : si $\mathscr{T}_1 + \mathscr{T}_2 \leq \mathscr{T}_3$ alors fusionner $S1$ et $S2$, ce qui donnerait la configuration observée sur la figure 19. Puis recommencer avec le nouveau $S1$ et $S2$. Par contre, il n'est pas possible d'envisager de fusionner un étage avec $S4$, l'étage le plus conséquent. D'autre part, pour tester l'intérêt de la fusion d'un étage avec un étage qui a été dupliqué, il faut prendre en compte le temps séquentiel et non pas le temps apparent. En effet, la fusion de $S5$ avec $S6$ (figure 18) donnerait un étage séquentiel.

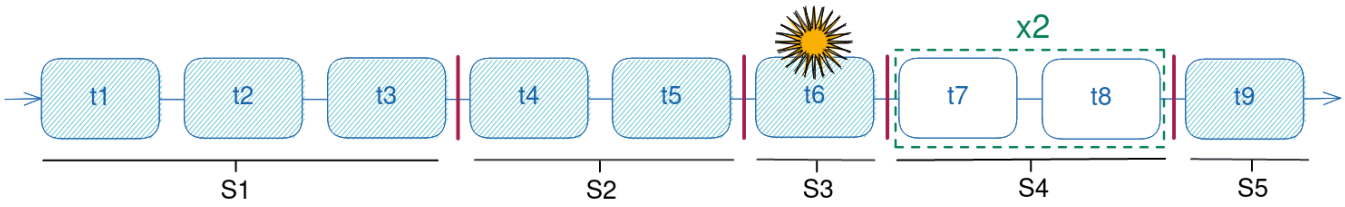


FIGURE 19 – Chaîne après la réduction de latence.

Une autre idée en perspective serait d'appliquer Pinar1D uniquement sur des sous-chaînes séquentielles qui ne contiennent pas l'étage dominant. Par exemple, pour la chaîne de la figure 18, les sous-chaînes concernées seraient $S1 + S2 + S3 = [t_1,\ t_2,\ t_3,\ t_4,\ t_5]$ et $S5 = [t_9]$ en prenant comme valeur de B le temps d'exécution maximal M. Le choix est fait de ne pas prendre en compte les étages dupliqués dans ces sous-chaînes. En effet, la duplication permet d'améliorer le débit, ce que ne permet pas l'opération de pipeline. Cette façon de faire est optimale sur la découpe de ces sous-chaînes.

4.3.6 Cas limites

Il existe deux configurations où l'ordonnancement est évident : le cas où toutes les tâches sont séquentielles, il faut alors appliquer l'algorithme Pinar1D en fonction du nombre de ressources disponibles P. À l'inverse, lorsque toutes les tâches de la chaîne sont parallélisables, il faut dupliquer la chaîne P fois (voir 3.2). L'objectif de l'heuristique est de respecter ces deux configurations-là et de gérer en plus les cas intermédiaires où la chaîne comporte des tâches séquentielles et parallèles.

Dans cette section a été présentée plusieurs heuristiques : la première utilise seulement la parallélisation de pipeline, mais n'est pas optimale. La deuxième est une heuristique utilisant la même technique de parallélisation, mais est optimale pour l'équilibrage de charges. La troisième heuristique présentée est celle que j'ai développée. Elle exploite les deux types de parallélismes de pipeline et de duplication et choisi de manière automatisée où appliquer ces techniques dans la chaîne. Afin de montrer l'intérêt d'utiliser ensemble ces deux techniques, il faut faire l'évaluation et la comparaison de ces heuristiques, notamment Pinar1D et celle que j'ai développée par rapport à leur utilisation en radio logicielle.

5 Évaluation : application à des chaînes existantes

Le but de cette section est de mettre en pratique les heuristiques d'ordonnancement et de parallélisme développées plus haut. Pour ce faire, j'ai réalisé un simulateur en Python dont le fonctionnement est expliqué dans la partie 5.1. Le but est de se rapprocher du fonctionnement qu'il y a dans AFF3CT en prenant des chaînes et des tâches issues de cette bibliothèque. La sous partie 5.1.1 détaille de quelle façon est réalisée la mesure du temps d'exécution dans AFF3CT et comment réutiliser cela dans le simulateur et la sous-partie 5.1.2 présente la chaîne qui servira de support pour l'évaluation. La partie 5.2 présente

les résultats des solutions obtenues par le simulateur par rapport à l'implémentation qui est faite dans AFF3CT mais aussi par rapport à l'algorithme Pinar1D.

5.1 Implémentation d'un simulateur

Le simulateur a été fait pour mesurer l'intérêt des techniques d'ordonnancement parallèles et automatiques développés par rapport à l'implémentation déjà existante dans AFF3CT. Le choix s'est porté sur la réalisation d'un simulateur [3] en Python pour la facilité de développement qu'offre ce langage. En plus de cela, le langage permet de faire de la programmation orientée objet, ce qui est adapté au modèle. Les objets suivants ont été crées :

- tâche (**task**) : définie par un poids p et un nom t ;
- étage (**stage**) : défini par une liste de tâches T, une variable booléenne is_dup selon si l'étage est duplicable ou non et le nombre de ressources allouées à l'étage ;
- ordonnancement (**scheduling**) : défini par une liste d'étages, le nombre total de ressources utilisées, les métriques de débit, latence et efficacité.

Pour chaque objet construit et chaque heuristique développée, il existe aussi des tests unitaires permettant de vérifier leur bon fonctionnement. De plus, il existe une extension de AFF3CT en Python, il est donc envisageable de tester la solution par la suite avec cette extension.

5.1.1 Mesure du temps d'exécution des tâches

Une fois les objets construits et les heuristiques implémentées, il faut tester sur des exemples et notamment des chaînes déjà implémentées dans le simulateur d'AFF3CT [4]. Les objets **task** permettent de construire une liste de tâche correspondant à une chaîne réelle avec les informations de temps d'exécution et le nom de la tâche, ainsi, j'ai pu recréer dans mon simulateur Python des chaînes d'AFF3CT. Or pour une chaîne réelle ou simulée, il existe une variabilité du temps d'exécution d'une même tâche en fonction de trame traitée notamment.

Afin de mieux se rendre compte de cette variabilité du temps d'exécution des tâches, j'ai décidé d'étudier une chaîne du simulateur d'AFF3CT. Les différentes tâches de cette chaîne sont les mêmes que celle de la figure 2, avec les caractéristiques choisies suivantes :

- la source génère des trames de manière aléatoire, chaque trame possède 1723 bits d'informations ;
- l'encodage choisi est de type polaire ;
- après l'étape d'encodage, la taille totale de la trame est de 2048 bits ;
- canal AWGN (bruit additif, blanc et gaussien) et le rapport signal sur bruit est de 4.2 dB ;
- la tâche sink vérifie si la trame décodée est identique à celle émise.

La transmission continue tant qu'un nombre fixé de trames erronées n'est pas détecté en fonction du rapport signal sur bruit.

Il existe dans AFF3CT, une fonctionnalité pour mesurer le temps d'exécution d'une tâche, il est notamment possible d'en récupérer la valeur dans l'affichage. Cependant, cette fonctionnalité, ne fait pas de mesure systématique à chaque exécution d'une tâche au sein d'une simulation. J'ai donc décidé de rajouter dans le code source ma propre mesure de temps à chaque fois qu'une tâche est appelée.

L'exécution d'une simulation correspond au traitement de 13554 trames et autant de mesure de temps d'exécution de chacune des tâches. Les options de simulations d'AFF3CT, pour la version 2.25.1. utilisée, permettent de choisir le nombre de threads sur lequel les simulations doivent être réalisées et par défaut

3. gitlab.inria.fr/dorhan/my-intership-with-aff3ct.git
4. aff3ct.readthedocs.io/en/latest/user/simulation/overview/overview.html

prend le nombre maximal de threads disponibles. Ici le choix a été fait de faire tourner les simulations sur un seul thread afin d'avoir une idée du temps d'exécution séquentiel uniquement. Ces mesures sont réalisées sur un ordinateur portable branché sur secteur, hors réseaux et avec seulement une fenêtre de terminal lançant la commande de simulation et rien d'autre. L'ordinateur que j'ai utilisé possède six cœurs Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz, une carte graphique NVIDIA Corporation TU117GLM [Quadro T1000 Mobile] / Quadro T1000/PCIe/SSE2, une mémoire de 15,3 Gio et un disque de 512 Go. Le compilateur utilisé est C++ GNU (g++) version 9.4.0.

Les graphiques de la figure 32 de l'annexe A présentent le temps d'exécution de chacune des tâches en fonction du nombre d'appels de la tâche ou numéro de trame traitée pour une simulation. À droite de chaque graphique se trouve un histogramme des temps d'exécutions d'une tâche, ce qui permet de se rendre compte de l'étalement et de la variabilité des temps. La première chose qui se remarque est le fait que pour toutes les tâches, il y a une diminution du temps d'exécution autour de la 1000e trame.

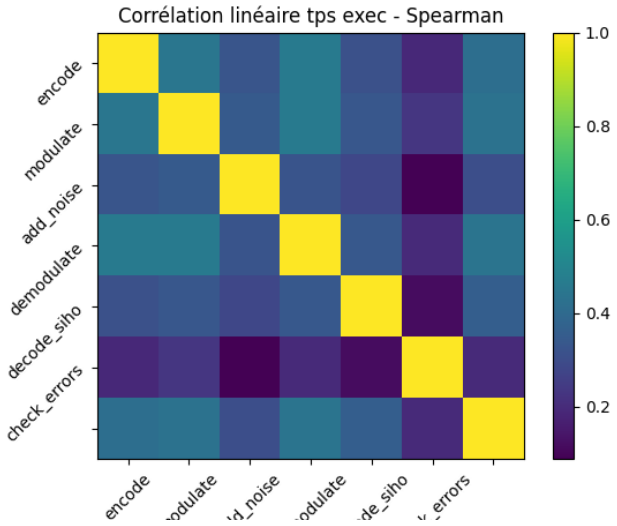


FIGURE 20 – Matrice de corrélation - Spearman - des temps d'exécutions des différentes tâches de la chaîne.

Comme la simulation est lancée "à froid", il faut un certain temps pour les données soient ramenées dans le cache. Les graphiques de la figure 33 de l'annexe B présente également les temps d'exécutions de chaque tâche en fonction ses appels successifs lorsqu'il y a eu auparavant plusieurs exécutions de la simulation. Dans ce cas, il n'y a pas cet effet de diminution général du temps d'exécution. Pour les tâches source, encodeur, canal, décodeur et moniteur, il est possible d'observer deux choses sur leur histogramme : la distribution des temps d'exécution n'a pas l'air de suivre de lois évidentes, notamment normales, et surtout, il y a un faible étalement de la valeur des temps de d'exécution pour ces tâches avec un pic important pour une frange de valeur dans la majorité des cas, le temps d'exécution de ces tâches se trouve dans l'intervalle de valeur de ce pic. Il est possible de se servir de ces temps par la suite pour l'évaluation. Pour les tâches de modulation et de démodulation, la distribution des temps d'exécution est plus étalée et ne semble pas suivre de loi normale. Un choix possible est d'écarter les valeurs les plus extrêmes puis de moyenner sur les valeurs restantes.

La figure 20 présente la matrice de corrélation de Spearman des temps d'exécution d'une tâche par rapport aux autres tâches. Dans l'ensemble, le coefficient de corrélation est inférieur à $0,5$, il n'y a donc pas de corrélation entre les temps d'exécution d'une tâche par rapport à l'autre. La tâche qui a les coefficients de corrélation le plus bas est la tâche du moniteur (*check_ error*), cela s'explique que cette tâche est décorrélée du fonctionnement de la chaîne puisqu'elle compare simplement deux trames entre elles.

L'étude qui a été réalisé sur les temps d'exécutions des tâches de la chaîne a permis de réaliser tout d'abord que pour mesurer le temps d'exécution d'une tâche, il faut faire tourner plusieurs fois les simulations pour éviter les effets de cache. Alors dans ce cas, pour l'ensemble des tâches, le temps d'exécution est concentré principalement dans un intervalle donné, ce qui permet d'établir une valeur représentative du temps d'exécution de la tâche. D'autre part, il n'y a pas de corrélation entre les temps d'exécution des tâches. Plusieurs pistes pourraient être explorées pour compléter cette étude et notamment :

- tracer la variation moyenne des temps d'exécution sur plusieurs simulations pour savoir s'il y a également un phénomène de cache à cette échelle ;
- déterminer s'il y a une corrélation entre le temps d'exécution d'une trame par une tâche par rapport à la trame précédente ;
- réaliser les mêmes mesures en fonctionnement réel.

Cette liste est non exhaustive et pourrait permettre de mieux connaître les biais et incertitudes manipulées. En effet, pour réaliser l'évaluation des heuristiques, il est nécessaire d'avoir des valeurs de temps d'exécution de tâche dans le simulateur implémenté. Cela permettrait de se rendre compte à quel point une heuristique serait efficace si elle était implémentée dans AFF3CT.

5.1.2 Chaîne de communication issue d'AFF3CT

Le standard de communication DVB-S2, pour *Digital Video Broadcasting - second generation*, est utilisé notamment pour la transmission satellitaire de contenu de vidéo [7] et est implémentée dans la bibliothèque AFF3CT [5]. Ce standard propose quatre types de modulations adaptatives parmi la QPSK, 8PSK, 16APSK et 32APSK. Le codage canal est également adaptatif et utilise des codes LDPC. Grâce à la diversité de codage canal et de modulation que propose ce standard, il peut être utilisé notamment pour du service de broadcast (*BS, Broadcast Services*), la transmission TV (*TV, Digital multiprogramme Television*) ou de la TV haute définition (*HDTV High Definition Television*).

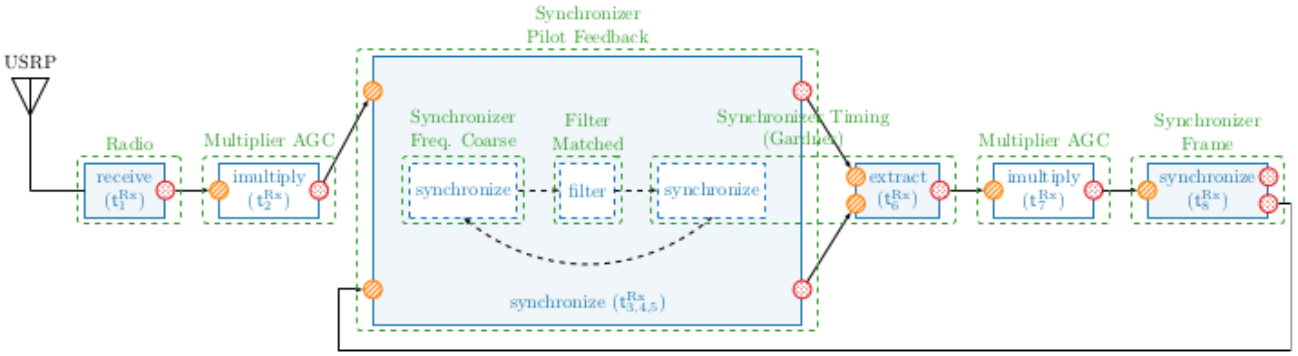
J'ai choisi d'utiliser cette chaîne pour réaliser l'évaluation des heuristiques. Je m'intéresse uniquement à la chaîne de réception où se trouve le décodeur, c'est-à-dire la tâche la plus gourmande en calcul.

Lorsque la chaîne est mise en fonctionnement, elle ne peut pas immédiatement traiter les trames qu'elle reçoit. Il faut des phases de synchronisation pour par exemple savoir où se situe le début de chaque trame avant que la transmission puisse commencer. Pour ce standard, elles sont au nombre de cinq [3] :

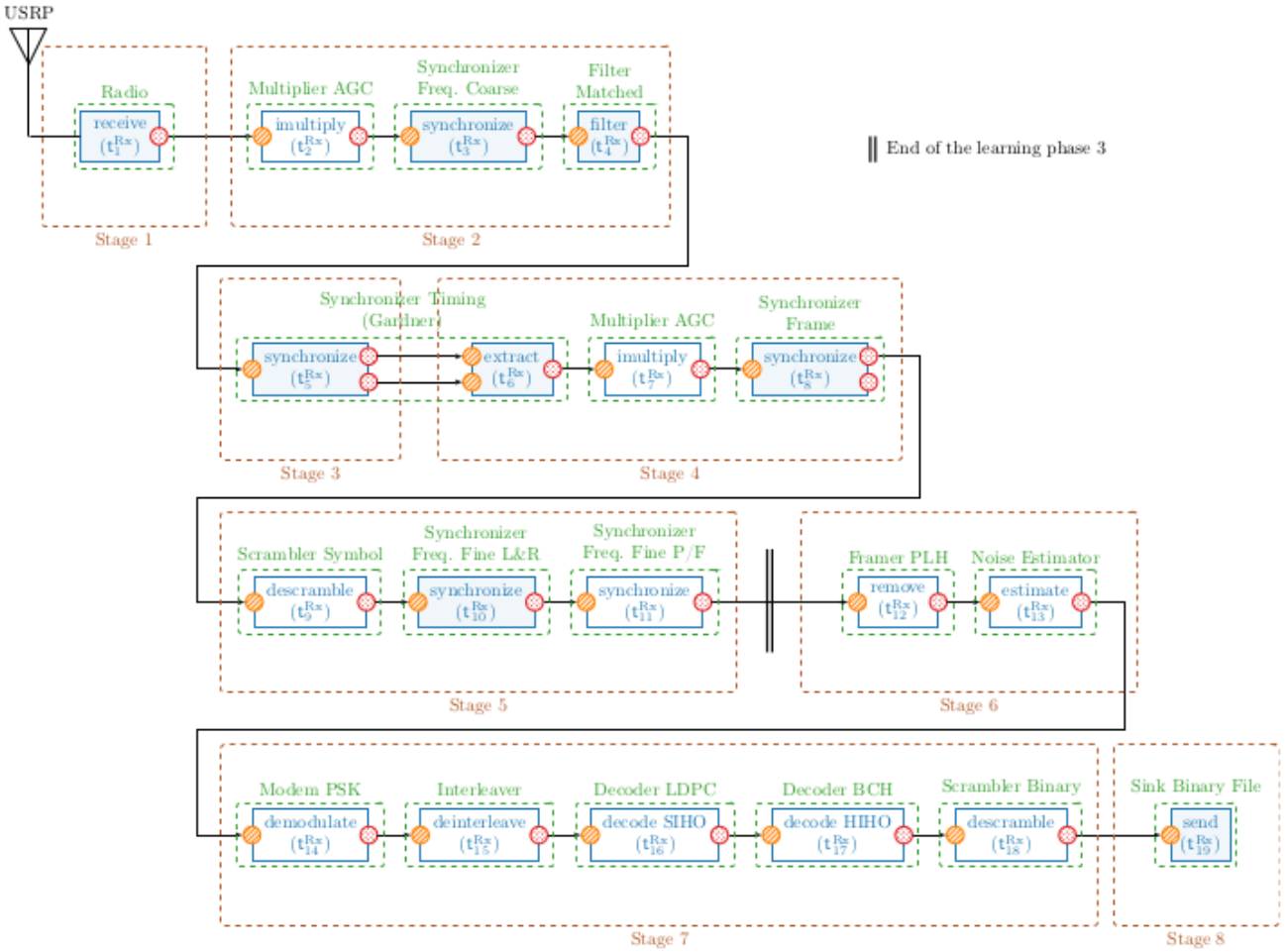
1. phase d'attente : phase durant laquelle le récepteur attend que l'émetteur envoie la première trame (voir figure 21a) ;
2. phase d'apprentissage 1 : se déclenche lorsqu'une trame est détectée et est exécutée pendant 150 trames (voir figure 21a) ;
3. phase d'apprentissage 2 : se déclenche à la fin de la phase d'apprentissage 1 et se déroule pendant 150 trames également (voir figure 21a), à la fin de cette phase, la chaîne est réagencée dans la configuration de la figure 21b ;
4. phase d'apprentissage 3 : correspond à l'exécution des tâches de *radio* jusqu'à *synchronizer sreq. Fine P/F* de la figure 21a, se déclenche après la phase 2 et est exécutée pendant 200 trames ;
5. phase de transmission : correspond à l'exécution de la chaîne entière (figure 21b), à ce moment la communication est établi et la transmission d'information commence réellement.

Les phase d'apprentissages 1 et 2 ne seront pas étudiées puisqu'elles contiennent des boucles et les heuristiques développées ne prennent pas en charge ce type de configuration. Par contre, l'évaluation portera sur l'ordonnancement de la phase d'apprentissage 3 et de la phase de transmission. Ces deux phases se distinguent par le fait que la tâche dominante n'est pas de même nature : séquentielle pour la phase d'apprentissage 3 et parallèle pour la phase de transmission. Pour l'heuristique que j'ai développé en 4.3, je

5. github.com/aff3ct/dvbs2.git



(a) Waiting phase and learning phase 1 & 2.



(b) Learning phase 3 & transmission phase.

FIGURE 21 – Implémentation logicielle de DVB-S2 [3].

m'attends à ce que la phase d'apprentissage 3 bénéficie majoritairement du parallélisme de pipeline et que la phase de transmission bénéficie plutôt de la technique de duplication et de pipeline.

5.2 Évaluation expérimentale

Cette partie se concentre sur l'évaluation de l'heuristique développée en 4.3 et de l'algorithme Pinar1D qui a été décrit en 4.2. Ces deux heuristiques sont comparées également à l'implémentation qui est faite

dans AFF3CT pour la chaine de réception du standard DVB-S2 pour les deux phases suivantes : la phase d'apprentissage 3 et la phase de transmission. Les critères de comparaisons seront le nombre de processeurs utilisé, le débit, la latence et l'efficacité. Comme l'implémentation d'AFF3CT utilise au plus 8 processeurs, cette valeur constituera le maximum possible de processeurs pour les deux autres heuristiques. Les valeurs de temps d'exécution des tâches prise sont celles issues de l'évaluation AFF3CT [3]. Une autre partie de l'évaluation consiste à observer l'évolution des métriques, débit et latence en fonction du nombre de ressources allouées. Cette étude peut s'avérer intéressantes lorsqu'il s'agit de faire un choix du nombre de ressources à allouer pour exécuter une chaîne avec des contraintes de performances.

5.2.1 DVB-S2 chaîne de réception en phase d'apprentissage 3

Lors de cette phase, la tâche la plus conséquente est séquentielle, il s'agit de la tâche *extract* ou aussi appelée *synchronisation timing 2*, sur la figure 22 elle est signalisée par la couronne.

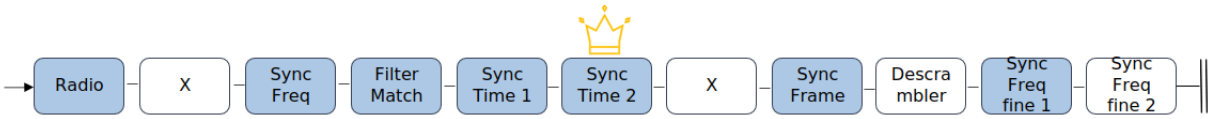


FIGURE 22 – Chaîne DVB-S2 lors de la phase d'apprentissage 3

Comparaison avec AFF3CT

Les différents découpages sont ceux obtenus à la figure 23 avec celui d'AFF3CT, de l'heuristique de Pinar1D et de mon heuristique. Les barrières de couleurs (vertes pour AFF3CT, bleu pour Pinar1d et violettes pour mon heuristique) permettent de séparer les différents étages. Il n'y a pas de duplication pour mon heuristique, ce à quoi, il était possible de s'attendre, car la tâche la plus conséquente est séquentielle et le reste tout le long du déroulement de mon heuristique. Le découpage optimal est donc celui donné par Pinar1D puisqu'il pipeline de façon optimale. Le but de mon heuristique est de se rapprocher le plus possible de la configuration obtenue par *Pinard1d*. La différence entre les deux provient de l'étape de réduction de latence (voir sous-partie 4.3.5) qui n'est pas optimale. Dans les trois cas, la tâche la plus conséquente est placée dans un étage de pipeline seule, c'est le temps d'exécution de cette tâche qui va rythmer le débit du pipeline.

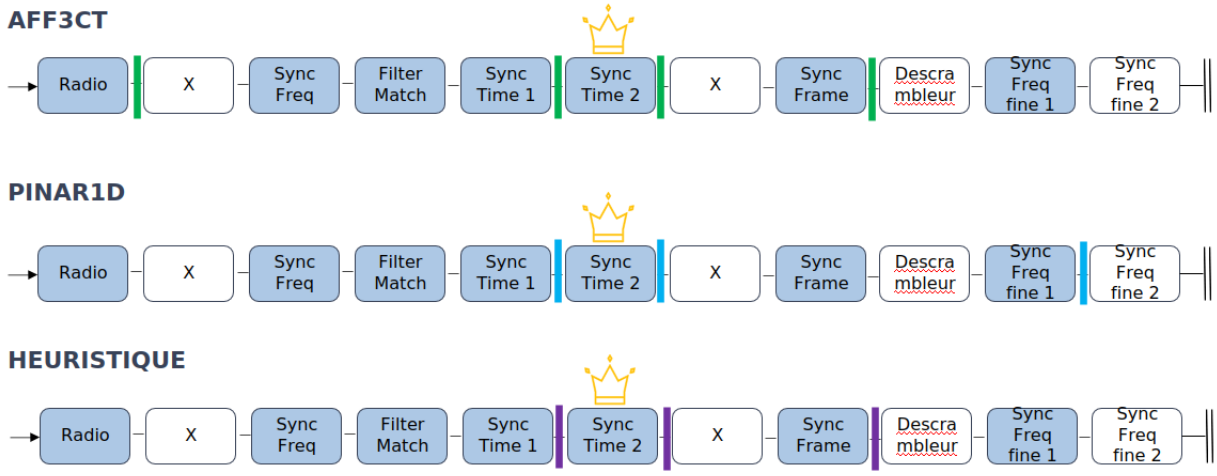


FIGURE 23 – Découpage de la chaîne de réception DVB-S2 en phase d'apprentissage 3 pour AFF3CT (haut), Pinar1d (milieu) et mon heuristique (bas).

Le tableau 3 présente l'évaluation des trois différentes stratégies d'ordonnancement, AFF3CT, Pinar1d et mon heuristique, selon les critères suivants : temps maximal d'étage, débit, latence, efficacité et nombre de ressources utilisées. Les cases colorées en vert indiquent la meilleure configuration par rapport à la

Ordonnancement	tps max (us)	débit (trame/s)	latence (us)	efficacité	nb de ressources
AFF3CT	4 109	243	18 775	0,66	5
Pinar 1D	4 109	243	14 350	0,86	4
Heuristique	4 109	243	14 666	0,84	4

TABLEAU 3 – Évaluation des métriques pour les différents ordonnancements en phase d'apprentissage 3.

métrique concernée et les cases en jaune signalent une configuration sous-optimale, mais très proche par rapport à la métrique concernée également. Les trois configurations obtiennent des performances de débit identiques. Elles se distinguent cependant sur la latence, Pinar1D et mon heuristique ont une latence plus faible et donc meilleure que celle d'AFF3CT. Le découpage d'AFF3CT utilise une ressource de plus et donc possède un étage de plus par rapport aux deux autres configurations, ce qui peut expliquer sa plus grande latence. Étant donné que la tâche la plus conséquente est séquentielle, la valeur de M est minorée par la valeur du temps d'exécution de cette tâche. L'objectif cet algorithme est donc de mettre cette tâche dans un seul étage, puis de faire au mieux avec la latence, sachant que l'optimal est donné par Pinar1D. Mon heuristique est très proche du résultat obtenu par Pinar1D ce qui est satisfaisant pour les métriques observées.

Nombre de ressources pour des performances attendues
Les figures 24 et 25 présentent le débit et la latence en fonction du nombre de ressources allouées à la chaîne DVB-S2 en phase d'apprentissage 3 pour les heuristiques Pinar1d et mon heuristique. Les graphiques ne vont pas au-delà de 11 ressources, car pour 12 ressources ou plus, aucune des métriques, débit D ou latence L, ne progresse encore. Il n'est donc pas utile de vouloir mettre plus de ressources que cela.

En ce qui concerne le débit, mon heuristique et de Pinar1D ont un débit identique en fonction du nombre de ressources. Ce qui signifie que du point de vue du débit, l'heuristique est optimale. Sur la figure 24, le débit atteint un plateau à 242 trames/s pour quatre ressources.

Pour la latence, mon heuristique est très proche du comportement de Pinar1D (voir figure 25). Entre 4 et 10 ressources, mon heuristique possède une latence plus élevée de l'ordre de $2,2\%$ au maximum. Pour Pinar1D, la valeur de la latence n'évolue plus au-delà de 4 ressources et pour mon heuristique, elle est quasiment constante entre 4 et 9 ressources. Il faut aller jusqu'à 10 ressources pour se retrouver avec la même valeur de latence que pour Pinard1D.

Globalement, il n'est pas toujours intéressant de vouloir rajouter plus de ressource pour exécuter la chaîne, à partir d'un certain nombre d'entre elles, les métriques n'évoluent peu ou plus du tout. La figure 26 présente le débit en fonction de la latence. L'idéal étant d'avoir le débit le plus élevé et la latence la plus faible. Ainsi, en associant cette courbe avec le nombre de ressources nécessaires pour avoir ces performances, il est possible de choisir le nombre de ressources à attribuer à la chaine.

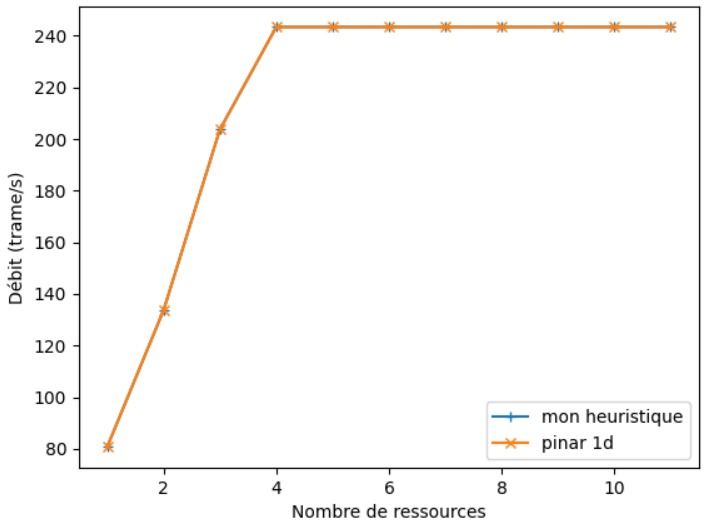


FIGURE 24 – Débit en fonction du nombre de ressources allouées pour la chaîne DVB-S2 en phase d'apprentissage 3.

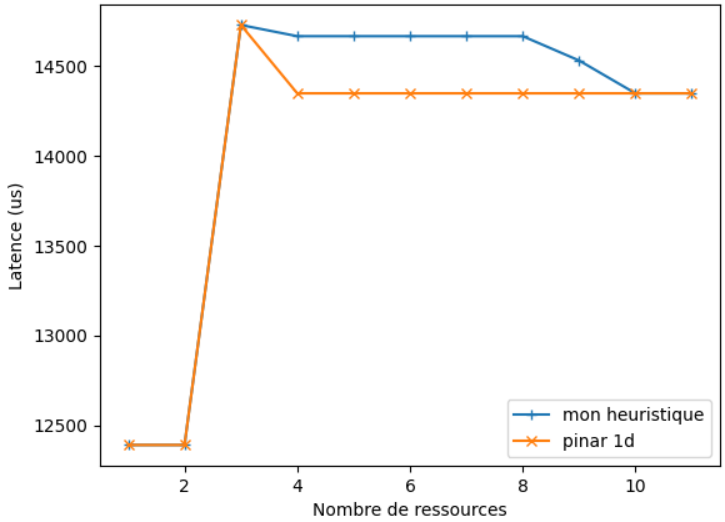


FIGURE 25 – Latence en fonction du nombre de ressources allouées pour la chaîne DVB-S2 en phase d'apprentissage 3.

5.2.2 DVB-S2 chaîne de réception en phase de transmission

Lors de cette phase, la tâche la plus conséquente est parallèle, il s'agit de la tâche *Decoder BCH*, sur la figure 27 elle est signalisée par la couronne. Comme expliqué dans la partie 2.1, la tâche la plus conséquente en calcul est la tâche de décodage.

Comparaison avec AFF3CT

Les différents découpages sont ceux obtenus à la figure 28 avec celui d'AFF3CT, de l'heuristique de Pinar1D et de mon heuristique. Les différents étages du pipeline sont indiqués par les barrières de couleurs (vertes pour AFF3CT, bleu pour Pinar1D et violettes pour mon heuristique) alors que les étages dupliqués

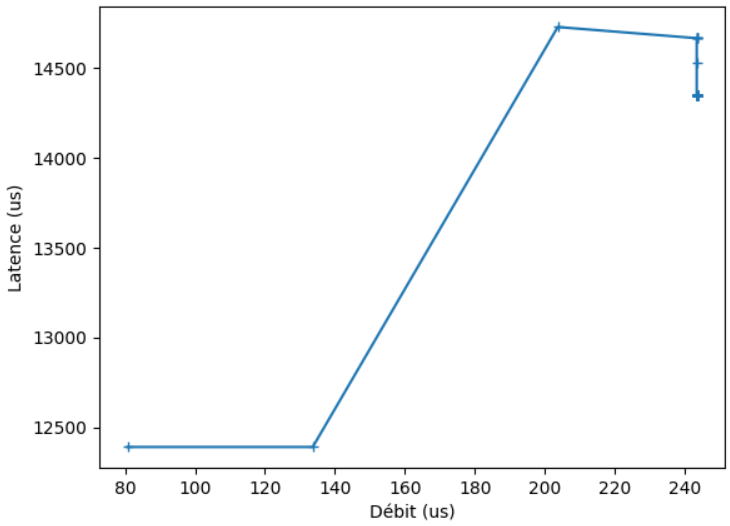


FIGURE 26 – Débit en fonction de la latence pour la chaîne DVB-S2 en phase d'apprentissage 3.

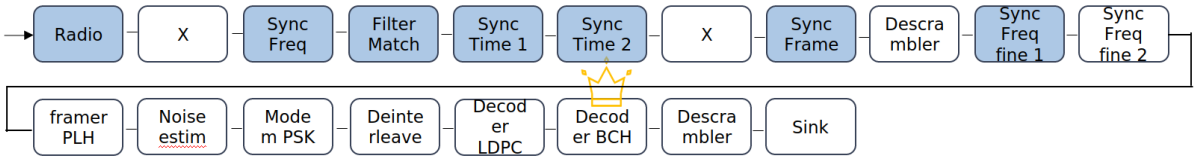


FIGURE 27 – Chaîne DVB-S2 lors de la phase de transmission

sont signalés par un cadre vert en pointillé, le nombre de ressources allouées à l'étage est également inscrit. AFF3CT pipeline la chaîne en huit étages alors que Pinar1d ne fait que trois étages en isolant la tâche *decoder BCH* puisqu'il s'agit du goulot d'étranglement. Mon heuristique utilise quant à elle les deux techniques de parallélisation possibles en pipelinant sur cinq étages et un dupliquant 4 fois l'étage contenant la tâche la plus conséquente.

Le tableau 4 présente l'évaluation des différentes configurations d'ordonnancement selon les critères de débit, latence, efficacité et nombre de ressources utilisées et reprends les mêmes codes couleurs que pour le tableau 3. Mon heuristique obtient un débit presque neuf fois supérieur à celui d'AFF3CT et sept fois supérieur à celui de Pinar1D. De la même façon, il y a un gain net sur la latence puiqu'elle est plus de dix fois plus faible par rapport à AFF3CT et presque trois fois plus faible par rapport à Pinard1D. Le gain obtenu grâce à la parallélisation par duplication est très net ici pour un même nombre de ressources que pour AFF3CT.

Ordonnancement	tps max (us)	débit (trame/s)	latence (us)	efficacité	nb de ressources
AFF3CT	42 587	23	298 232	0,19	8
Pinar 1D	32 905	30	68 434	0,89	3
Heuristique	4 109	203	22 289	0,89	8

TABLEAU 4 – Évaluation des métriques pour les différents ordonnancements en phase de transmission.

Nombre de ressources pour des performances attendues

Les figures 29 et 30, le débit et la latence en fonction du nombre de ressources allouées à la chaîne DVB-S2

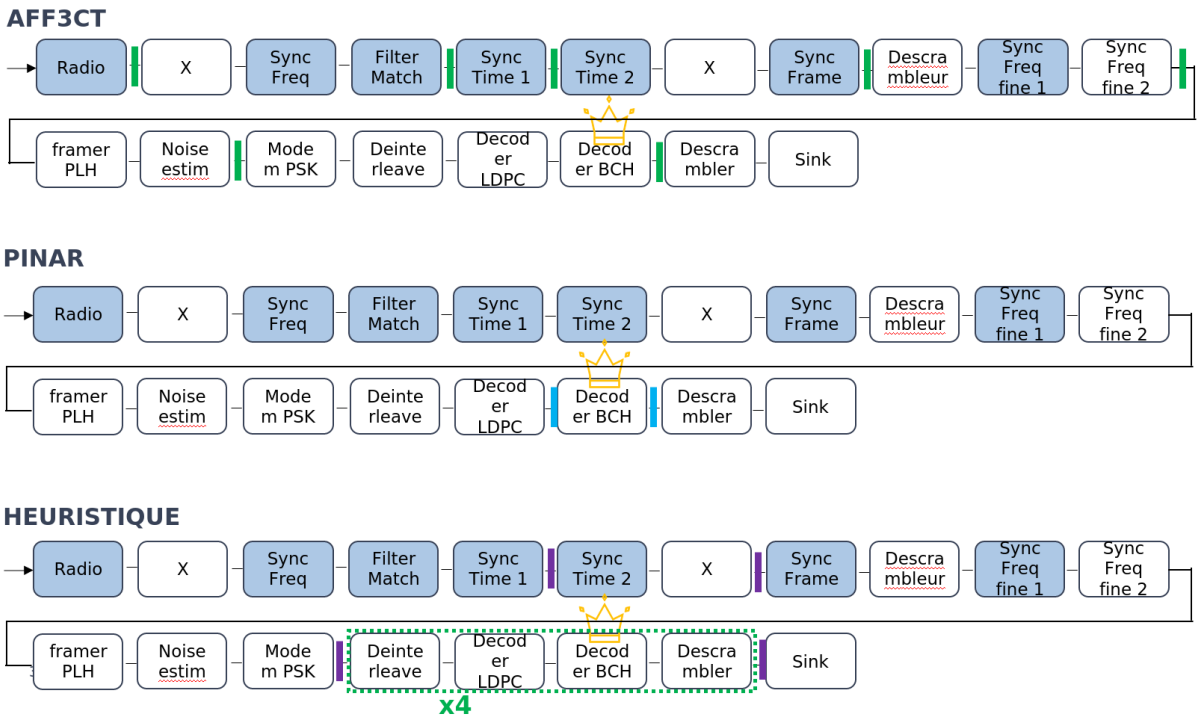


FIGURE 28 – Découpage de la chaîne de réception DVB-S2 en phase de transmission pour AFF3CT (haut), Pinar1d (milieu) et mon heuristique (bas).

en phase de transmission pour les heuristiques Pinar1d et mon heuristique. Les graphiques ne vont pas au-delà de 22 ressources, car pour 23 ressources ou plus, aucune des métriques, débit D ou latence L ne progresse encore. Il n'est donc pas utile de vouloir mettre plus de ressources que cela.

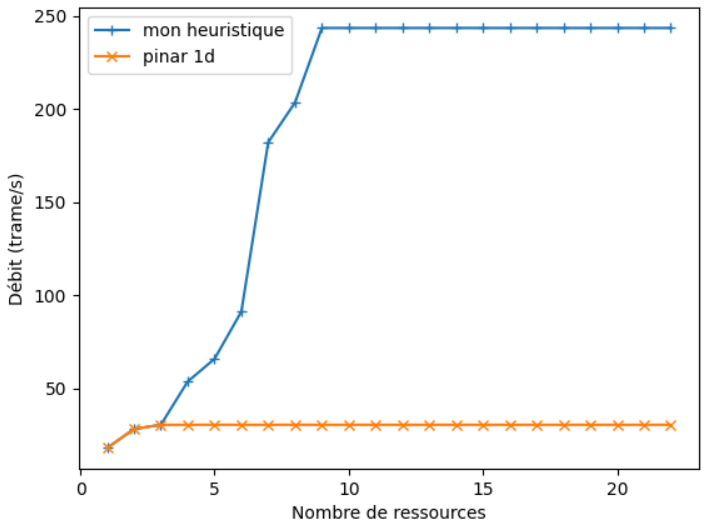


FIGURE 29 – Débit en fonction du nombre de ressources allouées pour la chaîne DVB-S2 en phase de transmission.

Le débit obtenu par mon heuristique est presque sept fois plus élevé par rapport à Pinar1D et ne progresse plus à partir de 9 ressources (voir figure 29). Ce qui correspond au moment où l'étage parallèle le plus

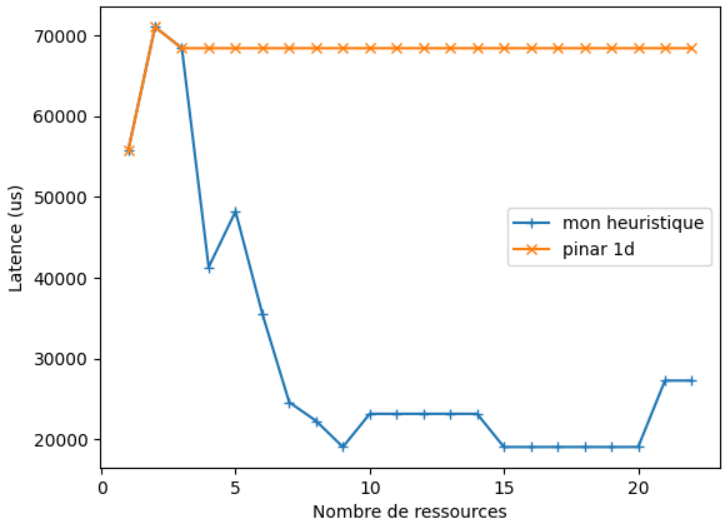


FIGURE 30 – Latence en fonction du nombre de ressources allouées pour la chaîne DVB-S2 en phase de transmission.

conséquent (celui contenant le décodeur) a été dupliqué suffisamment de fois pour que le nouvel étage le plus conséquent soit un étage séquentiel.

La latence atteinte par mon heuristique est jusqu'à trente fois plus faible que celle de Pinar1D pour 9, 15, 16, 17, 18, 19 ou 20 ressources (voir figure 30). Le fait qu'elle n'est pas une décroissance monotone peut s'expliquer par l'étape de réduction de latence non optimale d'une part, et d'autre part par le fait que rajouter une ressource peut avoir comme effet de rajouter un étage, ce qui peut dégrader la latence.

La figure 31 présente le débit en fonction de la latence. La configuration la plus intéressante reste lorsque le débit est le plus élevé et la latence la plus basse qui est atteinte pour 9 ressources. Cette courbe en l'associant avec le nombre de ressources de chaque configuration permet de faire un choix du nombre de ressources à choisir pour des contraintes de débit-latence données.

Cette partie évaluation a permis de valider l'intérêt d'avoir la duplication comme technique de parallélisation en plus de la technique de pipeline et notamment pour des configurations où la tâche la plus conséquente est parallèle. Dans le cas où la tâche la plus conséquente est séquentielle, l'optimal est donné par l'algorithme Pinar1D et l'heuristique développée s'en rapproche et permet d'avoir des valeurs de latence très proche pour la chaîne évaluée. En plus de cela, il est possible de vouloir connaître le nombre de ressources à allouer à l'exécution d'une chaîne pour des performances de débit et de latence voulue : il est possible de remonter à cette information grâce aux courbes montrées plus haut. Il faut également remarquer qu'il n'est pas toujours nécessaire de rajouter toujours plus de ressources dans le but d'améliorer les métriques, ce peut même être contre productif et dégrader notamment la latence.

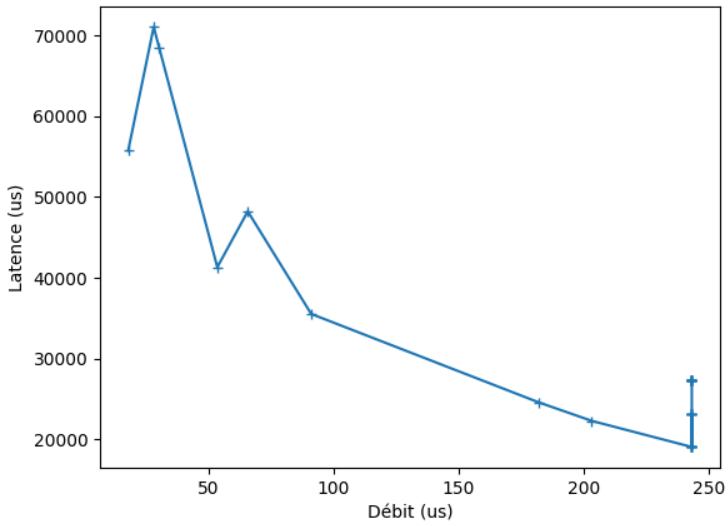


FIGURE 31 – Débit en fonction de la latence pour la chaîne DVB-S2 en phase de transmission.

6 Conclusion

Le but de ce stage était de développer une heuristique capable de faire réaliser un ordonnancement automatique et parallèle de chaînes de radio logicielles et notamment à celles du logiciel AFF3CT. La seule technique de parallélisation utilisée auparavant dans AFF3CT ou même dans les autres bibliothèques radio logicielle est le parallélisme de pipeline qui est fait de manière manuelle par l'utilisateur·ice. L'idée principale de ce travail est de rajouter, en plus de cette technique, du parallélisme de duplication de tâche et d'étage de pipeline, et cela, en fonction des caractéristiques initiales d'une chaîne, afin d'automatiser le processus de parallélisation. Pour cela, j'ai développé un **modèle du problème** permettant d'établir des règles claires pour la création d'une heuristique. Puis, à partir de ce modèle, j'ai développé une **heuristique** dont le but est d'avoir le meilleur débit et la plus faible latence de traitement de trames qui associe techniques de pipeline et de duplication. J'ai fait l'**implémentation de cette heuristique** dans un simulateur Python ainsi que l'implémentation d'une heuristique optimale d'équilibrage de charge utilisant uniquement le pipeline. Enfin, l'évaluation de cette heuristique sur une chaîne implémentée dans AFF3CT a permis de valider l'intérêt de rajouter la technique de duplication. Il est possible maintenant d'**obtenir un ordonnancement** pour une chaîne quelconque en connaissant ses données de dépendance de tâches, de temps d'exécutions de tâche, de nature de tâches (séquentielle ou parallèle) dans le simulateur Python pour un nombre de ressources voulues. Il est également possible de **connaître le nombre de ressources** nécessaires à l'obtention de performances voulues.

À la suite de ce stage, je vais continuer à travailler sur ce sujet en thèse. À court terme, j'envisage de remplacer l'étape de réduction de latence par une version optimale, de comparer les performances avec GNU Radio puisqu'il s'agit de la référence dans le domaine, mais aussi d'améliorer le modèle adopté en prenant en compte les coûts de synchronisation et de communication. J'envisage également de tester cet ordonnanceur sur les chaînes 5G qui sont actuellement développées par l'équipe CSN de l'IMS dans le cadre de la collaboration commune sur ce projet. Par la suite, cet ordonnanceur devra être intégré à AFF3CT. Enfin, sur du plus long terme, l'idée est de compléter et d'enrichir le modèle de chaîne en incluant des tâches conditionnelles ou des boucles, même de gérer l'ordonnancement de plusieurs chaînes s'exécutant sur la même architecture.

Bilan personnel

Suite à ce stage, je poursuis en thèse sur le même type de problématiques et dans la même équipe. Ce stage a été l'occasion de me familiariser avec l'équipe de recherche et de prendre en main le sujet qui se situe à l'interface de deux disciplines en cohérence avec ma formation.

Références

[1] Gnu radio, the free and open software radio ecosystem. https ://www.gnuradio.org/, consulté le 7 juillet 2022.

[2] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Research Report RR-6113, 2007.

[3] Adrien Cassagne. *Optimization and Parallelization Methods for the Software-Defined Radio*. PhD thesis, 12 2020.

[4] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jego. Mipp : a portable c++ simd wrapper and its use for error correction coding in 5g standard. pages 1–8, 02 2018.

[5] Pablo De Oliveira Castro Herrero. *Expression et optimisation des réorganisations de données dans du parallélisme de flots*. Theses, Université de Versailles-Saint Quentin en Yvelines, December 2010.

[6] David .G Messerschmidt Edward Ashford Lee. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36, January 1987.

[7] V1.2.1 ETSI EN 302 307. Digital video broadcasting (dvb) ; second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications (dvb-s2), 2009.

[8] Lawrence Goeller and David Tate. A technical review of software defined radios : Vision, reality, and current status. In *2014 IEEE Military Communications Conference*, pages 1466–1470, 2014.

[9] Inria. `www.inria.fr/fr/centre-inria-de-luniversite-de-bordeaux`, consulté le 24 Mai 2022.

[10] Anne F. MacLennan. Celebrating a hundred years of broadcasting – an introduction and timeline. *Journal of Radio & Audio Media*, 27(2) :191–207, 2020.

[11] Rudy Lauwereins Marc Engels, Greet Bilsen and Jean Peperstraete. Cyclo-static dataflow : Model and implementation. *1995 International Conference on Acoustics, Speech, and Signal Processing*, May 1995.

[12] Vincent Mazet. Communications numériques - université de strasbourg. vincmazet.github.io, Consulté le 6 juillet 2022.

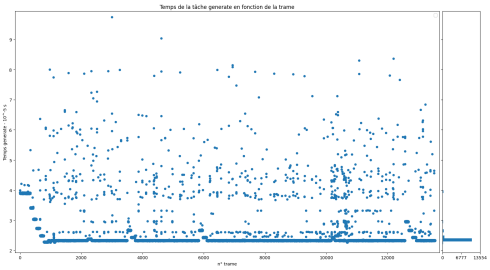
[13] Marcus Miller. Evolving the gnu radio scheduler, February 2020.

[14] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8) :974–996, 2004.

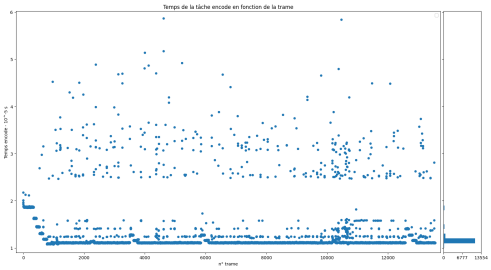
[15] AFF3CT team. Aff3ct documentation. aff3ct.readthedocs.io, 2022.

[16] Saman Amarasinghe William Thies, Michal Karczmarek. Streamit : A language for streaming applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002.

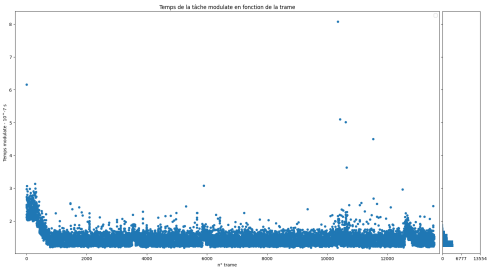
Annexe A



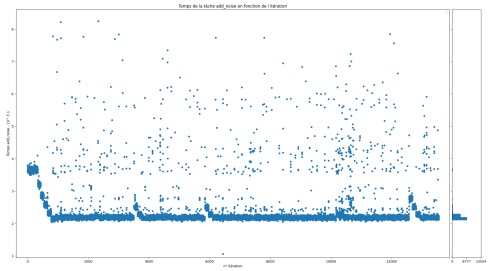
(a) Source.



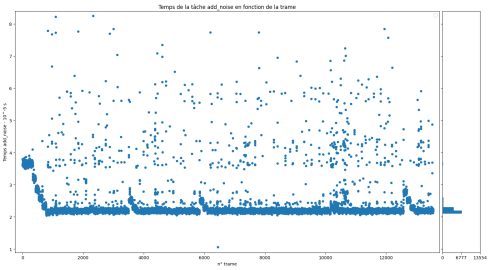
(b) Encodeur.



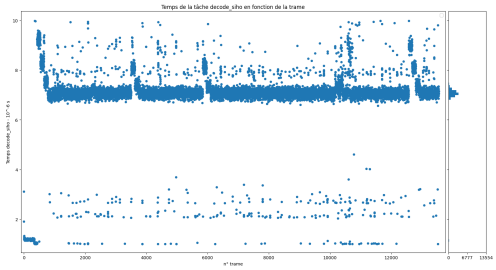
(c) Modulateur.



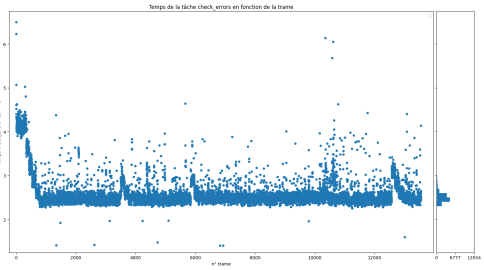
(d) Channel.



(e) Démodulateur.



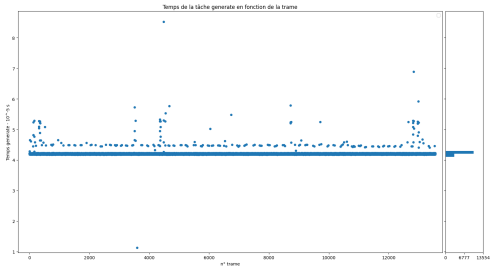
(f) Décodeur.



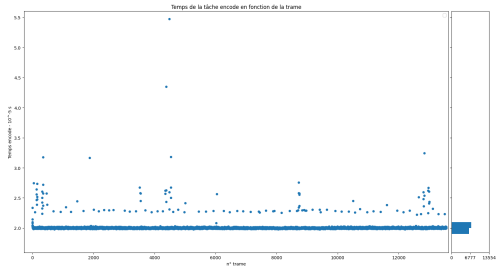
(g) Moniteur.

FIGURE 32 – Temps d'exécution des tâches de la chaîne pour une simulation en fonction du numéro de trame.

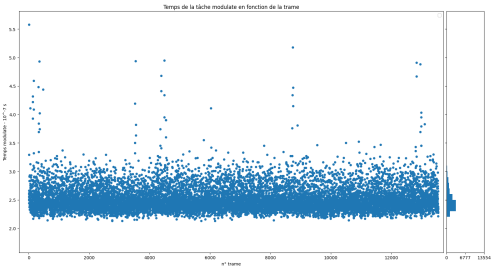
Annexe B



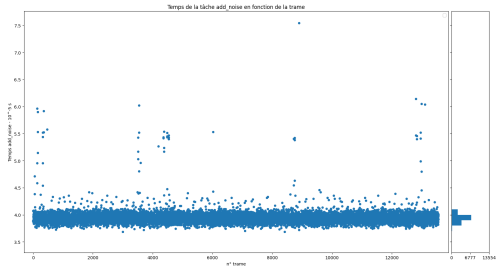
(a) Source.



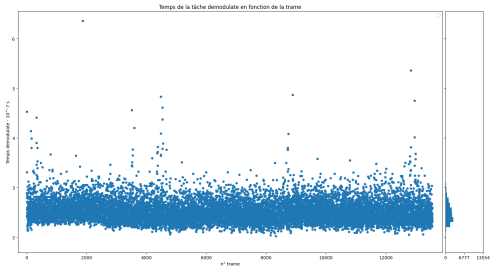
(b) Encodeur.



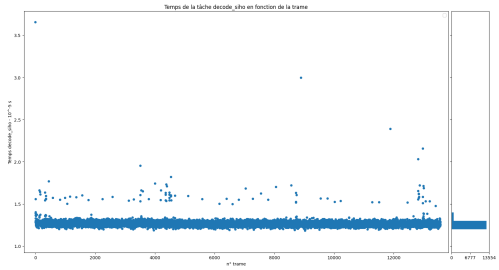
(c) Modulateur.



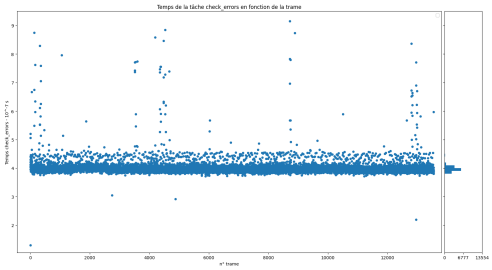
(d) Canal.



(e) Démodulateur.



(f) Décodeur.



(g) Moniteur.

FIGURE 33 – Temps d'exécution des tâches de la chaîne pour une simulation en fonction du numéro de trame, réalisé après plusieurs appels de simulation.