ENSG Géomatique

ÉCOLE NATIONALE DES SCIENCES GÉOGRAPHIQUES

Airware

École Nationale des Sciences Géographiques

Airware

Stage de fin d'études

Cycle des Ingénieurs diplômés de l'ENSG 3ème année

Développement d'un outil de manipulation optimisée de rasters volumineux

Amaury Zarzelli

Septembre 2018

☒ Non confidentiel ☐ Confidentiel IGN ☐ Confidentiel Industrie ☐ Jusqu'au ...

École Nationale des Sciences Géographiques
6-8 Avenue Blaise Pascal - Cité Descartes - 77420 Champs-sur-Marne
Téléphone 01 64 15 31 01 Télécopie 01 64 15 31 07

Jury

Président de jury :

Didier Richard, IGN, chef du service Valilab

Commanditaire :

Nicolas Goguey, Airware

Encadrement de stage :

Nicolas Goguey, Airware, Maître de stage
Oussama Ennafii, MATIS, IGN, Professeur référent

Rapporteur principal :

Jean-Marc Le Gallic, IGN, chef du pôle technique Géoportail

Responsable pédagogique du cycle Ingénieur :

Serge Botton, IGN/ENSG/DE/DPTS

Tuteur du stage :

Anna Cristofol, IGN/ENSG/DE/DSHI

Stage de fin d'études du 23/04/2018 au 21/09/2018

Diffusion web : ☒ Internet ☒ Intranet Polytechnicum ☒ Intranet ENSG

Situation du document :

Rapport de stage de fin d'études présenté en fin de 3ème année du cycle des Ingénieurs

Nombres de pages : 35 pages

Système hôte : LaTeX

Modifications :

EDITION	REVISION	DATE	PAGES MODIFIEES
1	0	29/08/2018	Première version
1	1	30/08/2018	Relectures
2	2	05/09/2018	Modifications mineures
3	3	10/09/2018	Version finale avant rendu

Remerciements

Je remercie vivement mon maître stage Nicolas Goguey pour sa bienveillance et sa disponibilité tout au long de mon stage, rendant ce dernier agréable et très enrichissant.

Je tiens aussi à remercier toute l'équipe d'Airware pour son accueil très chaleureux et pour l'excellente ambiance qu'ils insufflent dans le bureau.

Je souhaite également remercier toute l'équipe pédagogique de l'ENSG qui, à travers les divers domaines enseignés par l'école, m'a permis d'obtenir le bagage nécessaire au bon déroulement de ce stage, et avec laquelle j'ai passé 3 excellentes années.

Enfin, je remercie ma famille et mes amis pour leur soutien quotidien.

Résumé

La société Airware commercialise une plate-forme de suivi, d'analyse et de gestion, appliqués à deux verticaux : carrières et assurances, et en utilisant comme outils le drone et les images aériennes qu'il acquiert. Airware assure l'ensemble du processus, de l'acquisition des données jusqu'à la publication des résultats d'analyse sur leur plate-forme.

Les rasters produits par l'acquisition des données par drone ainsi que les algorithmes de machine learning les traitant sont lourds. Pour manipuler ces rasters, l'un des projets de l'équipe *Data Science* est la bibliothèque Python open-source buzzard (basée sur GDAL) qui permet la lecture et l'écriture de fichiers géographiques.

L'objectif de mon stage était d'aider au développement de la version suivante de buzzard permettant une manière bien plus optimisée que les précédentes de manipuler les rasters volumineux, en utilisant notamment de la programmation parallèle.

J'ai dans un premier temps développé une preuve de concept, burito, puis, dans un second temps, j'ai assisté mon maître de stage dans le développement de la version 0.5 de buzzard qui s'est déroulé en deux parties : une réécriture des fonctionnalités précédentes, qui a été déployée ; puis l'implémentation des spécifications de la preuve de concept, qui est toujours en cours et qui sera déployée prochainement.

Mots clefs : python, raster, programmation parallèle, buzzard, GDAL

Abstract

Airware is a company which markets a drone analytics solution –from acquisition to software– to enterprises that need to collect, manage and analyze aerial data. The major customers of the company are quarries and insurances.

One of the projects of the Data Science team is an open-source Python library name buzzard, which reads and writes geographical files using GDAL. Both rasters produced by drone acquisitions and machine learning algorithms used to process them are heavy, and buzzard is developed to manage them.

The main goal of my internship was to help develop the next version of buzzard, which will introduce a new and optimized way to process heavy rasters using parallel computing.

I firstly developed a proof of concept, burito. I then helped my supervisor in developing the 0.5 version of buzzard during two steps: a rewriting of the existing functionalities, which was deployed, and the implementation of the specifications of the proof of concept, which is still undergoing. This version is planned to be released shortly after the end of my internship.

Key words: python, raster, parallel computing, buzzard, GDAL

Table des matières

Table des figures

Glossaire et sigles utiles

ENSG École Nationale des Sciences Géographiques

OSGeo Open Source Geospatial Foundation, organisation dont le but est de promouvoir les logiciels open source en géomatique

GDAL Geospatial Data Abstraction Library, bibliothèque open source permettant la manipulation de données géographiques

RAM Random Access Memory, mémoire vive

GPU Graphics Processing Unit, carte graphique

CPU Central Processing Unit, processeur

DCNN Deep Convolutional Neural Network, réseau neuronal convolutif profond, modèle de machine learning permettant notamment de classifier les images

DSM Digital Surface Model, modèle numérique de surface, MNS

AWS Amazon Web Services, solution de cloud computing proposée par Amazon, la plus populaire à l'heure de l'écriture de ce rapport

buzzard bibliothèque open source Python de traitements de données SIG développée par Airware [1]

burito BUzzard Raster Inception TOol, nom de la preuve de concept que j'ai développée durant ce stage

Introduction

Dans le cadre de ma formation d'ingénieur de l'École Nationale des Sciences Géographiques, j'ai effectué comme travail de fin d'études un stage en entreprise du 23 avril au 21 septembre 2018 dans la start-up Airware, qui commercialise une plate-forme de suivi, d'analyse et de gestion à partir d'images aériennes acquises par drone. Airware assure l'ensemble du processus, de l'acquisition des données jusqu'à la publication des résultats d'analyse sur leur plate-forme. Ses deux types de clients principaux sont les carrières et les assurances.

L'acquisition de données par drone produit des fichiers rasters, traités par photogrammétrie, très volumineux. Ces mêmes fichiers sont utilisés pour l'analyse des données, notamment automatique par le biais du machine learning.

Intégré au sein de l'équipe Data Science, l'objectif initial de mon stage était de développer des outils permettant l'amélioration des processus de l'équipe, tant par l'efficacité que par la qualité des résultats. *A posteriori*, le travail que j'ai effectué concernait principalement l'efficacité d'exécution des tâches. Ce travail est actuellement en train d'être intégré à buzzard, une bibliothèque open source de traitements de données SIG développée et maintenue par Airware.

Chapitre 1 Présentation du besoin

1.1 Contexte du stage

1.1.1 La société Airware

Airware est une start-up fondée en 2011, possédant des bureaux à Paris et San Fransisco. Les bureaux français sont ceux de l'entreprise Redbird, qui a été achetée par Airware en 2016. La société compte aujourd'hui environ une centaine de collaborateurs, et a levé plus de 90 millions de dollars lors de plusieurs tours de table ces dernières années. Airware offre une solution clé en main de planification de vols de drones, ainsi que des services dans les domaines de l'assurance (inspection de toits par exemple), de la construction et de l'industrie extractive, notamment pour les carrières (calculs de volumes sur des stocks, analyse des écoulements...). L'une des orientations stratégiques actuelles de l'entreprise concerne le développement d'algorithmes efficaces de machine learning pour traiter rapidement les données des clients.

Dans l'entreprise travaillent, en ce qui concerne le volet technique, des pilotes de drones, des topographes, des spécialistes du traitement d'images (reconstruction d'images, photogrammétrie...), des géomaticiens, des développeurs web et *cloud* (l'entreprise étant très orientée vers le cloud computing , notamment vers Amazon Web Services), et enfin l'équipe de *data scientists* que j'ai rejointe, qui comporte à l'heure actuelle 4 personnes à Paris.

L'équipe *Data Science*

Une définition possible de la *data science* est le fait de créer de la donnée à partir de données. Ainsi, la principale mission de l'équipe est de proposer des algorithmes permettant de traiter les données photogrammétriques (orthoimages et modèles numériques de surface) pour en tirer des informations sémantiques. Ces algorithmes sont ensuite mis en production sur la plateforme Airware par l'équipe de développeurs *cloud*.

Une forme de traitement est la segmentation sémantique des images, qui est réalisée à l'aide d'un réseau neuronal convolutif profond (DCNN), traitement nécessitant de manipuler de très nombreux rasters, dont l'emprise au sol est souvent de plusieurs hectares avec une résolution de quelques centimètres. Ces rasters étant très volumineux, leur traitement est long et lourd.

Les données en entrée des algorithmes de l'équipe sont les modèles numériques de surface (MNS, ou DSM, *Digital Surface Model*) et les orthoimages, ce qui est assez rare dans la littérature. Par conséquent, l'équipe doit régulièrement réaliser des expérimentations sur les fichiers, ce qui est coûteux, d'autant plus que les calculs sont réalisés sur AWS. De fait, l'équipe a besoin d'un outil efficace pour la lecture et l'écriture des fichiers géographiques.

1.1.2 La bibliothèque buzzard

Mon stage s'est inscrit dans le développement de la bibliothèque open source de manipulation de fichiers spatiaux buzzard, produite et maintenue par Airware, et principalement par mon maître de stage Nicolas Goguey depuis mi-2017. Le but principal de buzzard est d'offrir un *wrapper* de haut niveau d'abstraction pour la librairie GDAL dans le langage de programmation Python, notamment dans l'optique d'être utilisé dans des processus de *data science*, en

offrant notamment des fonctionnalités de tuilage indispensables pour le traitement de rasters volumineux.

Les fonctionnalités de buzzard sont donc principalement la lecture et l'écriture de fichiers géographiques. Lorsque le développement de buzzard a débuté, et encore à l'heure actuelle, les seules solutions open-source viables pour réaliser ces tâches sont les bibliothèques de l'OSGeo, à savoir GDAL, OGR, PROJ.4 et GEOS. Or, aucune bibliothèque en Python de haut niveau d'abstraction combinant les quatre et satisfaisant les besoins d'Airware n'existait avant buzzard. Le développement de cette dernière est donc un investissement important pour la société.

1.2 Objectifs du stage

Initialement, mon stage avait d'autres objectifs en plus de mon projet principal, notamment le développement d'une implémentation de l'algorithme *CRF as RNN* pour *Keras* et *tensorflow GPU*, que je ne détaillerai pas ici car ce ne fut pas abordé du tout lors de ces 5 mois.

L'objectif de mon stage au sein de l'équipe *Data Science* de Airware a été de développer un outil pour rendre efficace la création, le calcul et la sauvegarde de rasters volumineux, pour à terme être intégré à la bibliothèque buzzard. En effet, avant mon arrivée, les fichiers manipulés par l'équipe, qui sont des rasters de plusieurs gigaoctets, étaient traités un par un dans la chaîne de traitements, ce qui ralentissait sensiblement les phases de calculs. De plus, le volume de ces fichiers provoquait une trop forte utilisation de la mémoire vive, car ils devaient être entièrement mis en mémoire pour être traités.

Ce fonctionnement n'est pas viable, car il limite la taille maximale des rasters à traiter, ce qui réduit le nombre de clients possibles. De plus, avec l'amélioration de la résolution des appareils d'acquisition et l'augmentation de l'autonomie des drones, la taille des fichiers augmente indépendamment de l'aire totale.

Pour pallier ces problèmes, deux axes sont envisagés. Le premier est la mise en cache des résultats des traitements. Cela permettrait d'optimiser les nombreuses phases d'expérimentation de l'équipe *Data Science*, mais aussi d'accélérer les traitements pour les clients. Le second est la lecture en parallèle des fichiers. Cela est certes possible, mais pénible du fait des limitations à la fois issues du langage Python et de la bibliothèque GDAL. Ainsi, la logique nécessaire au parallélisme est volumineuse en termes de code informatique, mais cela n'est pas un problème dès lors qu'une abstraction est créée pour encapsuler cette complexité.

Certaines pistes avaient été explorées avant mon arrivée par mon maître de stage pour implémenter une solution, mais aucune d'entre elles ne réalisait les spécifications du projet.

1.3 Méthodologie employée

1.3.1 Organisation du travail

L'équipe dans laquelle j'ai été intégré applique la méthode Scrum. Nous faisions des *stand-ups* trois fois par semaine, et les membres travaillent de manière agile : au jour le jour, en fonction des besoins, un développeur peut se joindre à un autre pour réfléchir et/ou faire du *pair programming*. Ainsi, au cours du développement de mon projet, j'ai été assisté à de nombreuses reprises par mes collègues, dont mon maître de stage, et, inversement, je les ai aidés sur certaines problématiques. Ce suivi régulier m'a permis de m'assurer de travailler dans une bonne direction.

J'ai utilisé l'outil Git, associé à la solution en ligne GitHub pour versionner mon code et le partager à l'équipe de développement. De plus, j'ai participé à l'aide de ces outils à la revue de code de plusieurs travaux de mes collègues.

Enfin, conformément aux méthodes agiles, j'ai développé le projet par itérations successives, chaque étape étant fonctionnelle et ajoutant des fonctionnalités par rapport à la précédente.

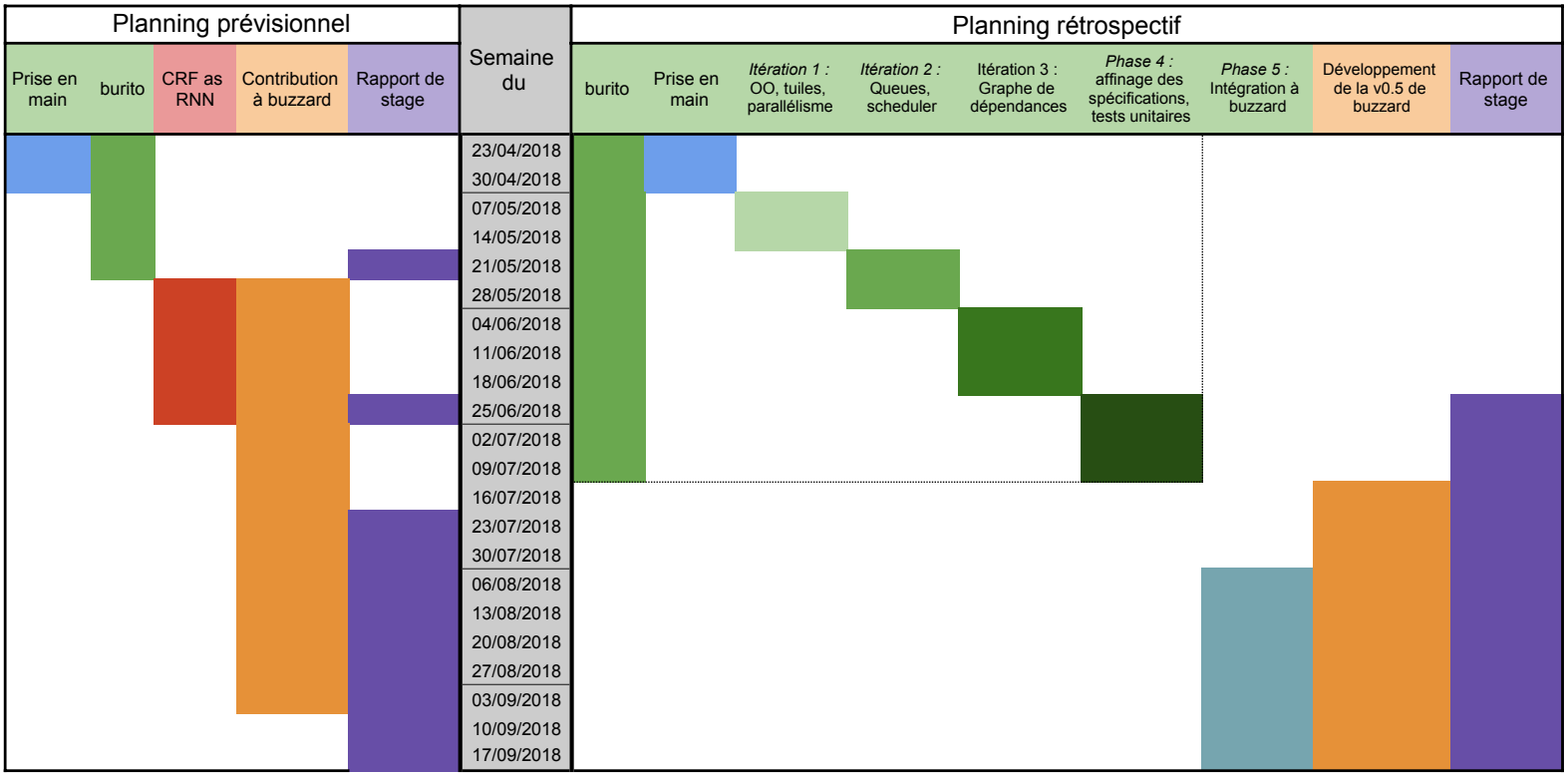

FIGURE 1.1 – Plannings prévisionnel et rétrospectif

1.3.2 Plannings prévisionnel et rétrospectif

Comme constaté sur la comparaison entre plannings prévisionnel et rétrospectif figure 1.1 (page 13), le projet que j'ai développé durant mon stage était initialement prévu comme une tâche relativement courte. Cependant, en précisant les spécifications avec mon maître de stage et en constatant la complexité du travail à réaliser, nous avons décidé de faire de cette tâche un projet de stage complet.

Son intégration à buzzard, étape très importante car permettant à la bibliothèque d'évoluer, a aussi été une part importante de mon travail car j'y ai assisté mon maître stage à temps plein.

Chapitre 2 Spécifications du projet

La première phase du projet a été de définir une première version des spécifications de l'outil, spécifications qui ont été affinées au fur et à mesure de l'avancement du travail. Je détaillerai ici les spécifications finales du projet ainsi que leurs justifications.

2.1 Optimiser les temps de traitement

Le premier grand axe de spécification est la rapidité de calcul. En effet, avant mon arrivée, les nombreux rasters traités en amont et pendant les algorithmes de machine learning l'étaient à chaque itération, et un par un. Ce processus était très long car il nécessitait de nombreux calculs, et ce à chaque fois que les développeurs voulaient tester leurs modèles et paramètres. Afin de réduire le temps d'exécution du processus, je me suis concentré sur les points suivants :

2.1.1 Ne pas calculer deux fois un pixel long à calculer

Une manière de voir l'optimisation de la tâche est de ne calculer qu'une seule fois les pixels coûteux (par exemple, ceux issus de la prédiction par DCNN nécessitant le GPU), que ce soit entre deux exécutions ou au sein de l'exécution du programme.

- Pour ne pas calculer le même pixel entre 2 exécutions du programme, on réalise une mise en cache sur le disque des rasters calculés pendant l'exécution du programme. Ainsi, lors d'un lancement ultérieur, plutôt que de recalculer les pixels coûteux, on se contentera de lire un fichier, ce qui est beaucoup plus rapide pour certains types de rasters. Cela dit, certains pixels sont très peu coûteux à calculer (par exemple, les pentes) et l'opération de lecture sur le disque peut être plus lente que le calcul. De fait, on fera en sorte que la mise en cache soit optionnelle pour pouvoir gérer les deux cas de figure.
- Pour ne pas calculer le même pixel au sein de l'exécution du programme, on réalise la même opération de mise en cache sur le disque. En effet, la somme du temps d'écriture et du temps de lecture est souvent inférieure au calcul de pixels coûteux. De plus, travaillant sur un noyau Linux, les fichiers récemment accédés sont mis en cache sur la RAM grâce au principe de *Page Cache* [8], rendant l'accès aux fichiers récents bien plus rapide. Enfin, buzzard utilise la bibliothèque GDAL pour la gestion de fichiers, et cette dernière implémente aussi un système de mise en cache des fichiers. Si l'on tire partie de ces deux systèmes de cache, cela assure que la lecture/écriture est plus rapide que le calcul. Ainsi le design du programme sera influencé par ces deux mécanismes.

2.1.2 Optimiser l'utilisation du matériel

Un autre axe d'optimisation est celui de l'utilisation d'un maximum des ressources de calcul à disposition : multiprocesseur, carte graphique, utilisation du disque. Pour ce faire, on peut subdiviser les tâches en plusieurs sous-tâches parallèles, et les ordonner entre elles.

- Afin d'optimiser l'utilisation de tous les processeurs à disposition, on peut subdiviser la tâche principale en sous-tâches parallèles. Cela se traduit sur un raster par un tuilage des rasters d'origine, permettant à chaque processeur d'exécuter le calcul sur des tuiles

séparées. La mise en cache de ces tuiles (voir partie précédente) permet de plus une utilisation maximale des lectures/écritures sur le disque.

- Afin d'optimiser encore plus l'utilisation du matériel, et notamment de la carte graphique, il convient de donner un ordre précis aux sous-tâches. En effet, certaines tâches (notamment la prédiction) nécessitent l'utilisation du GPU. Afin d'optimiser l'utilisation de ce dernier, il faut que les données dont il a besoin soient prêtes le plus tôt possible, ce qui ne peut être déterminé que par la définition d'un ordre d'exécution des tâches. De plus, le tuilage défini au-dessus permet aussi de rendre les tâches prêtes le plus vite possible car elles sont alors plus rapides à réaliser du fait de la taille limitée des tuiles.

2.1.3 Limiter le surcoût processeur

Pour éviter d'avoir une exécution trop lente du programme, il convient de limiter le surcoût processeur (*overhead*), c'est-à-dire le temps passé à l'exécution de tâches annexes au programme, qui font la liaison entre les tâches principales. En d'autres termes, la proportion du temps de traitement sur le temps total d'exécution du programme doit être extrêmement majoritaire. Par exemple, la lecture d'une tuile ne devra pas être significativement plus longue que la même opération faite par GDAL seul.

2.2 Économiser le besoin en mémoire

Une autre spécification du projet est de limiter l'utilisation de la mémoire vive. Les fichiers volumineux traités dans les processus de machine learning peuvent causer une surcharge de la RAM, qu'il faut éviter.

2.2.1 Subdiviser les calculs

Afin de ne pas mettre en mémoire la totalité des rasters traités, on peut réaliser une subdivision du travail en sous-tâches. Cette subdivision étant aussi nécessaire pour la rapidité du traitement (voir partie précédente), on constate que le tuilage permet de réaliser les deux objectifs principaux du programme.

2.2.2 Utiliser une mise en cache sur le disque

La mise en cache sur le disque présentée dans la partie précédente permet non seulement de ne pas calculer deux fois le même pixel, mais aussi de limiter l'utilisation de la mémoire vive, car la mise en cache est faite sur le disque et non dans la RAM. Le *Page Cache* du noyau Linux utilise un algorithme *Least Recently Used* (LRU) pour la gestion du cache en RAM. Ainsi, une ressource mise en cache sur le disque (et en RAM en même temps) sera retirée de la RAM si elle n'est pas fréquemment utilisée lorsque la RAM devient pleine, pour que cette dernière soit allouée aux ressources les plus récentes.

2.2.3 Limiter la back pressure

Le lancement de calculs en chaîne sur des rasters peut provoquer de la *back pressure*, c'est-à-dire une occupation de la RAM par de nombreux résultats rapidement terminés nécessités par des calculs plus longs à finir. Ainsi, il faudra là encore réaliser un ordonnancement et mettre en pause certaines tâches de manière à empêcher ce phénomène.

2.3 Spécifications de l'API

Dans buzzard, la définition d'un raster est en fait la définition d'un ensemble de protocoles à respecter. Dans un but d'intégration à la bibliothèque, le programme devra implémenter ces protocoles, permettant ainsi de manipuler sans distinction des rasters "classiques" (fichiers) et des "nouveaux" rasters en utilisant le polymorphisme par héritage.

Les méthodes à implémenter pour pouvoir satisfaire les spécifications de buzzard sont les suivantes :

- des propriétés permettant de définir le raster, son emprise au sol, ses bandes, etc. ;
- **get_data()** : méthode prenant en argument une empreinte au sol et renvoyant le tableau Numpy de données correspondant dans le fichier ;
- **iter_data()** : méthode prenant en argument un ensemble d'empreintes au sol. Il s'agit d'un itérateur qui retourne les tableaux Numpy de données correspondant dans le fichier dans l'ordre défini en entrée ;
- **queue_data()** : méthode prenant en argument un ensemble d'empreintes au sol et renvoyant une queue asynchrone de la bibliothèque standard de Python qui contient les tableaux Numpy de données correspondant dans le fichier dans l'ordre défini en entrée. Cette queue est remplie de manière asynchrone par le raster.

Ces points d'entrée permettent à un raster qui est dérivé d'autres rasters de faire des requêtes à ces derniers. Il suffira de renseigner les liaisons entre rasters pour qu'ils s'imbriquent les uns aux autres.

2.4 État de l'art en amont du stage

Afin de satisfaire les spécifications précédentes, en amont de mon stage, mon encadrant avait déjà réalisé une phase d'exploration de solutions possibles pour la réalisation du projet, mais aucune ne convenait totalement à la problématique.

Tout d'abord, il a essayé d'appliquer des modèles connus : la programmation réactive et le modèle d'acteurs.

La première piste explorée a été celle de la programmation réactive, dont le paradigme correspond à la problématique et dont la mise en place, modulaire, est adaptée à certaines spécifications. Cependant, l'implémentation en Python de la programmation réactive n'est pas à la hauteur de ses implémentations en Scala ou en JavaScript [14]. De plus, la gestion de la *back pressure* en utilisant ce paradigme est très difficile.

Le modèle d'acteur présente lui aussi l'avantage d'être modulaire, et la mise en place des acteurs atomiques est aisée. Cependant, les implémentations Python de ce modèle ne sont pas à jour et ne sont visiblement plus maintenues : en effet, les dernières mises à jours datent d'environ deux ans [11] [15]. De plus, ce modèle présente un part trop importante de surcoût processeur (*overhead*), et devient irréaliste si trop d'acteurs sont présents. Or, le projet est développé pour ouvrir potentiellement des centaines de fichiers en même temps.

Ensuite, faute de trouver un modèle correspondant à la problématique et disponible en Python, mon maître de stage a exploré les bibliothèques disponibles pouvant potentiellement répondre aux spécifications soulevées par le projet.

La première bibliothèque qui a semblé pertinente a été Dask, et plus précisément Dask Delayed [2] et Dask Distributed [3]. Le premier a été jugé trop simpliste pour la problématique. Le second présente de nombreuses fonctionnalités intéressantes pour le projet, mais présente trop de surcoût processeur. En outre, des comportements imprévisibles ont étés observés en utilisant cette bibliothèque, comme des ralentissements et des blocages inexpliqués. Ce manque de fiabilité a provoqué l'écartement de cette solution.

La seconde bibliothèque explorée a été la librairie standard de Python asyncio [5], qui a notamment l'avantage d'avoir une syntaxe très lisible. Néanmoins, cette solution n'est pas bien supportée en dessous de la version 3.5 de Python. Or, la bibliothèque buzzard a vocation à être utilisée par un public large, et donc à ne pas forcer la version de Python des utilisateurs. De plus, cette bibliothèque se concentre sur les opérations d'entrée/sortie, qui ne sont pas les opérations majoritaires réalisées par le programme.

Suite à cette phase d'exploration, mon maître de stage est arrivé à la conclusion que les briques constitutives de la solution devraient être des queues asynchrones, permettant la communication entre fils d'exécution. C'est sur cette conclusion que s'est basé le développement de la preuve de concept.

Chapitre 3 Burito : une preuve de concept

Une fois les premières spécifications définies, j'ai pu commencer le développement d'une preuve de concept (*proof of concept*, POC), que j'ai nommée burito, pour **bu**zzard **r**aster **i**nception **to**ol. J'ai implémenté les spécifications une à une dans un processus itératif (*cf.* planning rétrospectif, figure 1.1 page 13), et je détaille dans cette partie la solution définitive du programme, qui a inspiré la version 0.5 de buzzard.

Le développement de la solution présentée a été réalisé en quasi-totalité par moi-même, mais les fréquentes phases de réflexion étaient réalisées avec plusieurs membres de l'équipe Data Science.

Dans ce chapitre, je décris brièvement le fonctionnement général de la preuve de concept, puis je détaille les différentes classes qui entrent en jeu, et enfin je décris l'élément central de mon programme, le graphe des dépendances, avant de conclure sur cette preuve de concept.

3.1 Fonctionnement général

Un raster de burito est un objet complexe, qui possède, entre autres, des attributs constants, des références vers d'autres rasters (dits "primitives"), des fonctions permettant de récupérer les données sur les primitives et de lancer les calculs avec ces données en entrée, des fonctions permettant la fusion de données, des objets assurant le parallélisme, un graphe permettant l'ordonnancement des opérations (voir partie 3.3), etc.

Cet objet complexe peut être accédé par d'autres rasters (s'il est une primitive) et/ou par le fil d'exécution principal *via* les points d'entrées définis dans la partie précédente.

Pour gérer et ordonner tous ces accès, attributs et fonctions, j'ai écrit une fonction dite de *scheduler* (planificateur) qui ordonne tous les appels aux fonctions, accès au graphe, etc. Cette fonction tourne dans une boucle infinie dans un fil d'exécution propre.

Par ailleurs, le raster possède aussi des références vers des *pools* de *threads*, qui sont des regroupements de fils d'exécution permettant l'exécution de plusieurs tâches en parallèle. Les deux pools principales sont la *pool* d'entrée/sortie (*io pool*) chargée des accès en lecture et écriture sur le disque, et la *pool* de calculs (*computation pool*) chargée d'exécuter les calculs sur les tuiles du cache.

3.1.1 Exemple simple

Le digramme de séquence présenté figure 3.1 (page 21) décrit l'exécution d'une requête simple faite à un raster burito qui implémente une mise en cache et qui n'a pas de primitives.

Les instructions sont les suivantes. On crée d'abord un objet *pool* qui regroupe un seul *thread* (pour simplifier l'exemple). On créé ensuite un objet *Raster* en lui associant la pool précédente en tant que *pool* d'entrée/sortie. Une autre *pool* est créée à l'instanciation du raster pour pouvoir réaliser les calculs. Ensuite, on fait une requête au raster pour obtenir les données sur les tuiles du caches nommées a et b dans l'ordre a-b-b-a.

À l'arrivée de la requête, le *scheduler* va calculer le graphe des dépendances (voir partie 3.3) puis le parcourir. Ensuite, il va assigner à la *computation pool* le calcul des tuiles. À la réception

des résultats de ces calculs, le *scheduler* va planifier l'écriture sur le disque de ces données, puis la lecture. Pour les itérations suivantes sur la même tuile, on ne devra réaliser qu'une opération de lecture. Les tableaux de données correspondant aux emprises de la requête sont renvoyées dans le même ordre, et ce dès qu'ils sont prêts.

3.2 Interactions des classes

Comme on peut le constater sur la figure 3.2 (page 22), les classes sont séparées en 2 catégories : les classes privées, et la classe de façade, *Raster*. Cette dernière permet d'exposer les méthodes et attributs publics. La séparation entre la façade et le raster a été faite pour éviter que le fil d'exécution responsable du *scheduler* (attribut *scheduler_worker* de la classe *Raster*), qui a une référence vers la fonction *scheduler()*, ne fasse référence à la même classe, créant ainsi un cycle qui n'est collecté qu'à la fin de l'exécution du programme. Dans l'état présenté dans le diagramme de classes, lorsque toutes les références à une instance de Raster sont perdues, on envoie un message au *scheduler_worker* qui stoppe la boucle infinie définie dans la fonction *scheduler()*. Cela ne serait pas possible si les deux éléments étaient présents dans la même classe car le *worker* faisant alors référence à une fonction membre, il serait impossible de perdre toutes les références à une instance.

L'organisation des classes différencie aussi les rasters qui implémentent la mise en cache de ceux qui ne le font pas (distinction nécessaire, voir partie 2.1.1) par le biais d'héritage. Ainsi, en fonction des paramètres fournis par l'utilisateur au constructeur de la classe *Raster*, on instanciera l'une ou l'autre des classes en attribut *backend*, qui agira de la même manière pour la façade par polymorphisme.

3.3 Graphe des dépendances

Afin de pouvoir définir à partir d'une requête faite à un raster tous les calculs nécessaires, les dépendances à satisfaire, et pouvoir les ordonner, et ce en évitant les redondances, j'ai utilisé un graphe orienté. Chaque raster (et donc chaque *scheduler*) a un graphe attitré, qui est mis à jour à chaque requête faite au dit raster. Le parcours et la modification du graphe permettent au *scheduler* de planifier les opérations : à partir des attributs d'un nœud, il est possible de déterminer l'action à réaliser, et une fois celle-ci terminée, le nœud est retiré du graphe.

Un exemple de graphe de dépendances dans un cas avec mise en cache est présenté en figure 3.3 (page 23). Dans cet exemple, les nœuds sont représentés par des cercles, et deux cercles d'une même couleur (excepté blanc) représentent la même emprise au sol.

Terminologie employée

- **produce** : les nœuds qualifiés ainsi sont ceux qui correspondent à des requêtes faites au raster *via* les points d'entrée *get/iter/queue_data()* ;
- **read** : ces nœuds correspondent aux tuiles du cache qu'il faut lire pour pouvoir retourner les données dans le nœud *produce*. Pour un nœud *produce* donné, les nœuds *read* auxquels il sera lié correspondront aux tuiles du cache qui intersectent son emprise au sol ;
- **write** : ces nœuds correspondent aux tuiles du cache qui ne sont pas encore écrites sur le disque ;
- **compute** : ces nœuds correspondent aux emprises au sol qu'il faut calculer pour pouvoir écrire les données sur le cache ;
- **collect** : ces nœuds correspondent aux requêtes qu'il faut faire aux primitives d'un raster pour pouvoir réaliser les calculs.

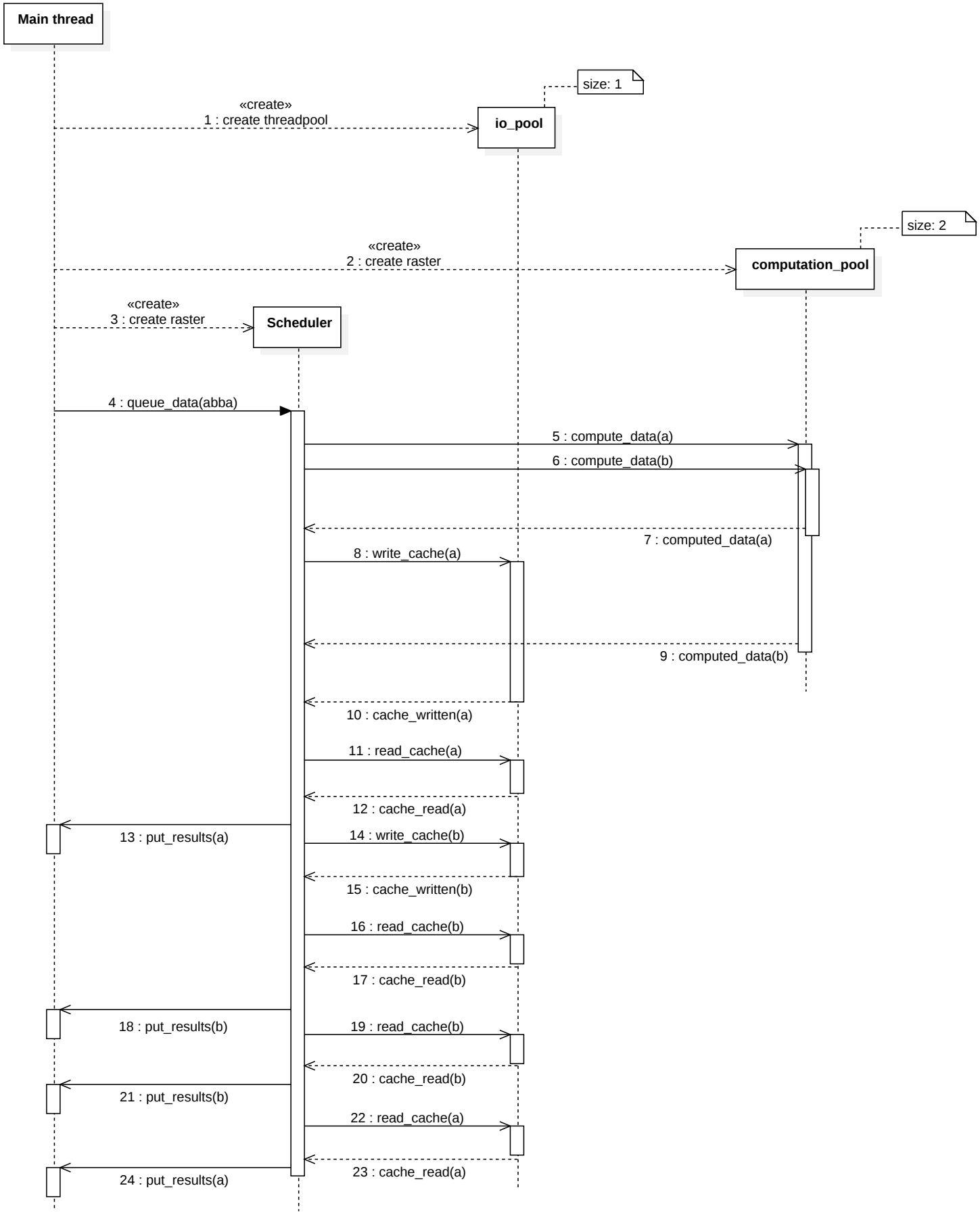

FIGURE 3.1 – **Exemple de diagramme de séquence.** On créé dans le *thread* principal une *threadpool*, que l'on associe à un raster à la création de ce dernier. On fait une requête sur le raster sur les tuiles a et b dans l'ordre a-b-b-a.

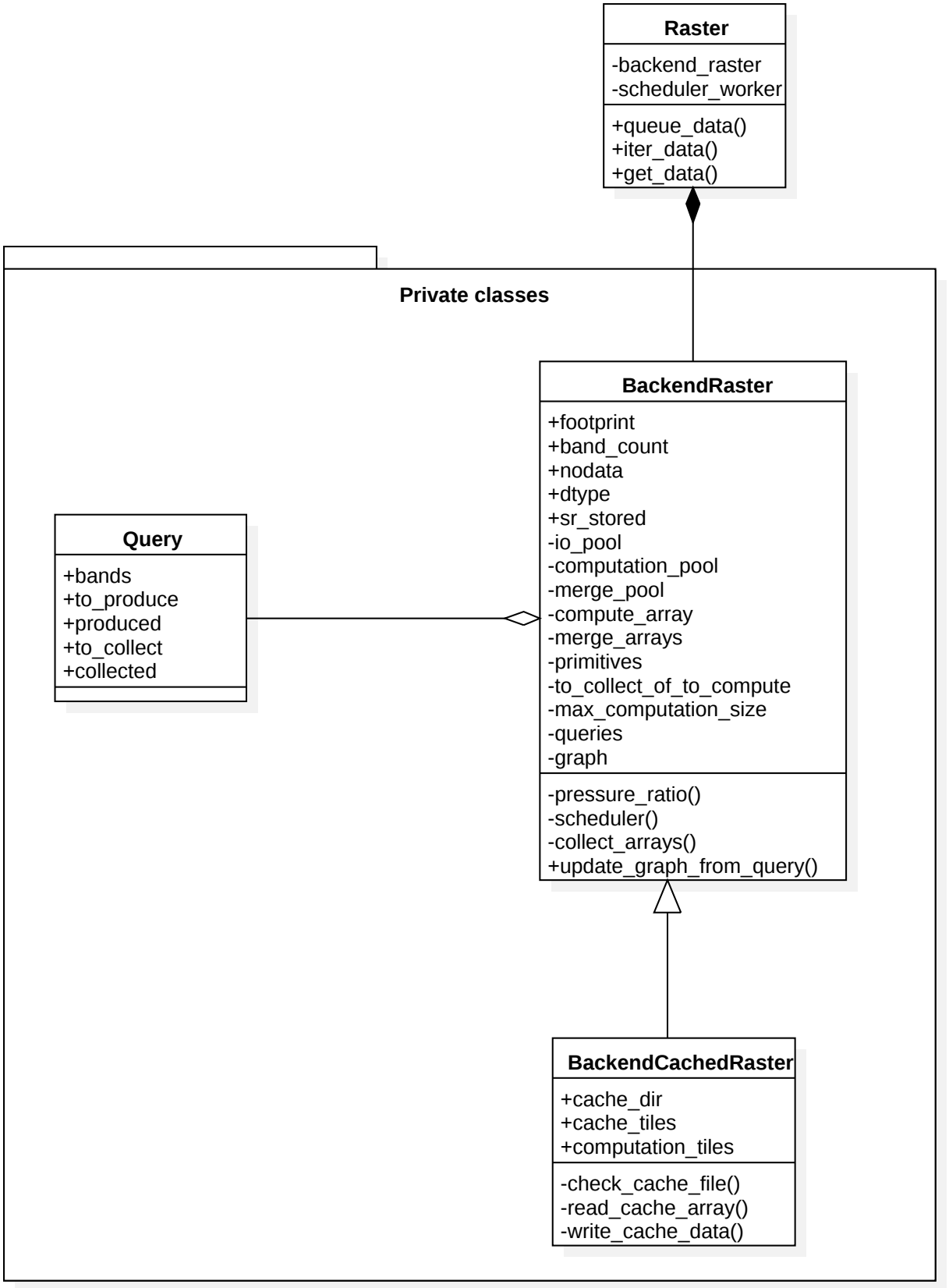

FIGURE 3.2 – **Diagramme de classes de burito.** Les points d'entrée dans la classe publique permettent la séparation du *thread* planificateur et de l'implémentation de sa fonction, permettant la *garbage collection*

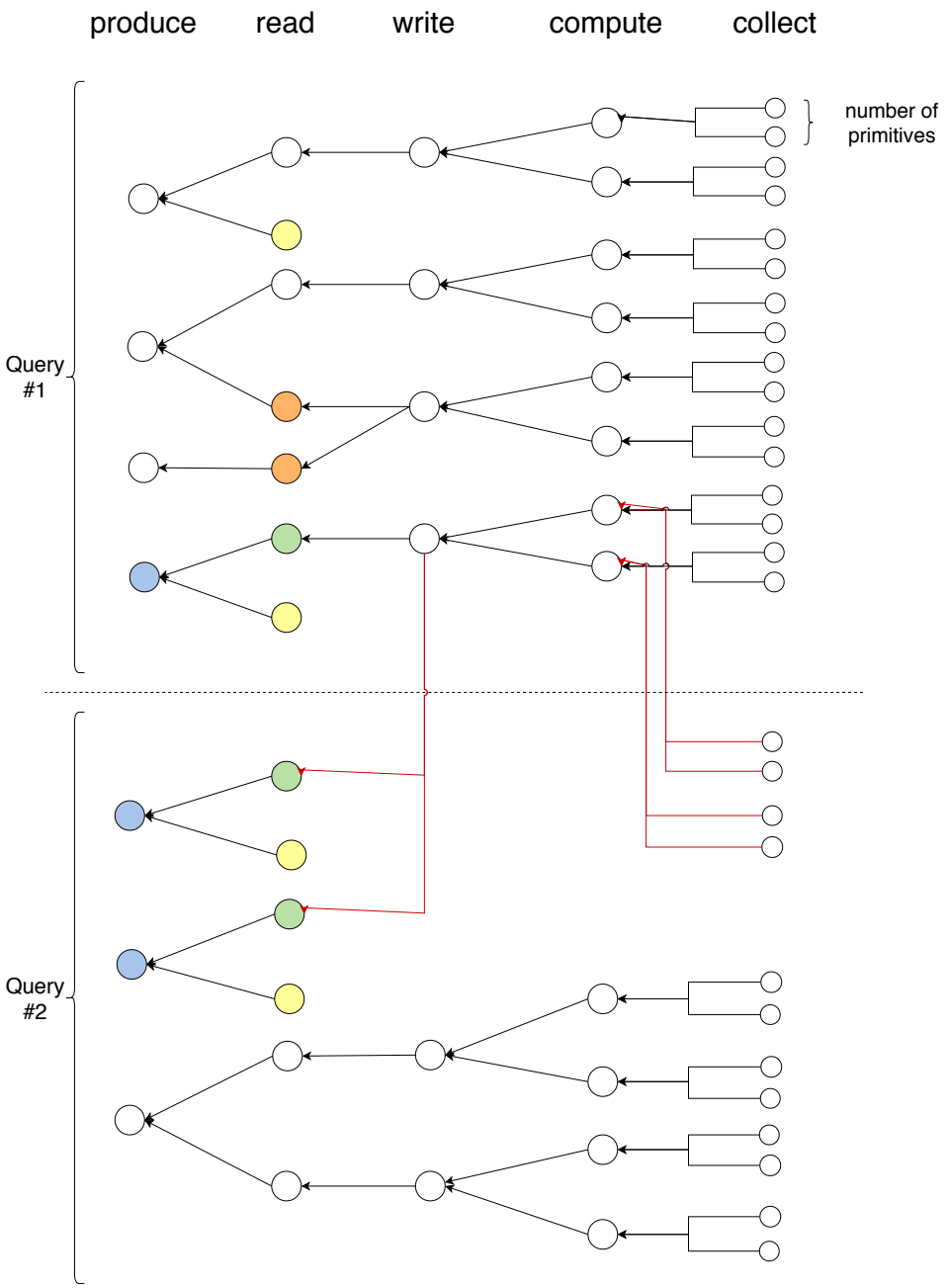

FIGURE 3.3 – **Exemple de graphe de dépendances.** Les nœuds de la même couleur (excepté blanc) correspondant à des mêmes tuiles de cache. Ainsi, la même tuile à lire est présente plusieurs fois, elle ne sera écrite qu'une seule fois. Les dépendances indiquées en rouge signifie que l'on prévoit autant de requêtes pour les mêmes tuiles des rasters primitifs qu'il y a de requêtes. Cela permet d'éviter le blocage du programme.

On constate que la structure du graphe est similaire à celle d'un arbre, mais en est distincte car, par exemple, plusieurs *produce* peuvent dépendre de *read* sur la même emprise au sol, qui ne dépendent logiquement que d'un seul *write* (nœuds oranges sur l'exemple).

Les dépendances que j'ai indiquées en rouge sur la figure montrent que, parmi les requêtes faites à un même raster, même si plusieurs requêtes dépendent d'un même *write*, les nœuds *collect* correspondant seront dupliqués pour chaque requête qui en dépend. Ce choix est fait pour éviter le blocage d'une requête. En effet, les résultats d'une requête sont renvoyés dans le même ordre que les entrées, et ce dans une queue à taille limitée. Si, dans l'exemple, la queue de chaque requête est de taille 2 et que les données n'en sont pas retirées tant que les 2 requêtes n'ont pas de résultat, et si les *collect* n'étaient pas dupliqués, la requête 2 ne recevrait jamais de données et l'exécution du programme serait bloquée. La duplication permet de résoudre ce problème.

3.4 Retours sur la réalisation

Le prototype final que j'ai développé, étant fonctionnel, a prouvé que le développement de burito est possible. Par ailleurs, il a permis la découverte de problématiques qui n'étaient pas prévues lors de la définition des spécifications (par exemple les problèmes de *garbage collection*) et la définition précise des spécifications de la solution qui sera déployée dans la prochaine version de buzzard.

Chapitre 4 Déploiement de la solution

4.1 Intégration à buzzard

L'objectif final du développement de burito étant son intégration à buzzard, la dernière étape de mon projet a été ladite intégration. Son développement a été fait en *pair programming*, avec comme développeur principal mon maître de stage.

L'intégration a été réalisée en deux étapes, les tests unitaires étant écrits à la fin de chaque étape.

4.1.1 Développement de la version 0.5.0b0 de buzzard

La version 0.5.0b0 est une réécriture des classes de la version précédente (0.4.4) de buzzard, sans changement majeur sur les fonctionnalités et sur l'API publique. Le développement de cette version a fait l'objet d'une *Pull Request* sur le dépôt GitHub de buzzard, visible à l'adresse https://github.com/airware/buzzard/pull/36.

Séparation des spécifications des classes

Afin de préparer l'intégration de burito dans buzzard, il a fallu repenser l'organisation des classes de la bibliothèque, et leurs interactions. Dans l'état précédent de buzzard (version 0.4.4), quelques classes suffisaient au fonctionnement de la librairie. Dans le cadre de la version 0.5, mon maître de stage et moi-même avons décidé de définir un ensemble de classes abstraites définissant de manière très fine tous les objets pouvant être traités par la bibliothèque, l'ouvrant ainsi aux extensions.

Comme visible sur la figure 4.1 (page 26), initialement buzzard ne supportait que les drivers GDAL, et les comportements de chaque type de *proxy* (objets manipulables par buzzard) étaient tous définis au sein d'une seule classe.

L'apport majeur des classes définies pour la version 0.5 comme dans la figure 4.2 (page 26) est la différenciation entre *Proxy* et ses classes filles (colonne centrale dans le diagramme de classes). Ainsi, la classe *Stored* introduit une notion d'écriture, la classe *Emissary* une notion de *driver* et la classe *PooledEmissary* une notion de *file descriptor*. Cela permet de définir des classes concrètes qui n'implémentent que certaines de ces spécifications, par exemple, la classe *NumpyRaster* qui a été ajoutée dans cette version, qui n'implémente pas de notion de driver.

Séparation entre façade et implémentation

En prévoyant l'implémentation du *scheduler* de burito dans buzzard, nous nous sommes rendu compte que les références de la *DataSource* vers le scheduler, de celui-ci vers les *proxies* et de ces derniers vers la *DataSource* créaient un cycle inter-*thread*, qui n'était pas détruit lors de la *garbage collection*, provoquant d'importantes fuites mémoire. Pour pallier ce problème, nous avons créé deux niveaux "topologiques" de classes : des classes dites de "façade" qui exposent des méthodes publiques, et des classes qui implémentent les spécifications de ces dernières (figure 4.3, page 27). Les références se font uniquement dans le sens façade vers implémentation. Ainsi, quand la référence aux classes de façade est perdue dans le programme principal, celles vers le

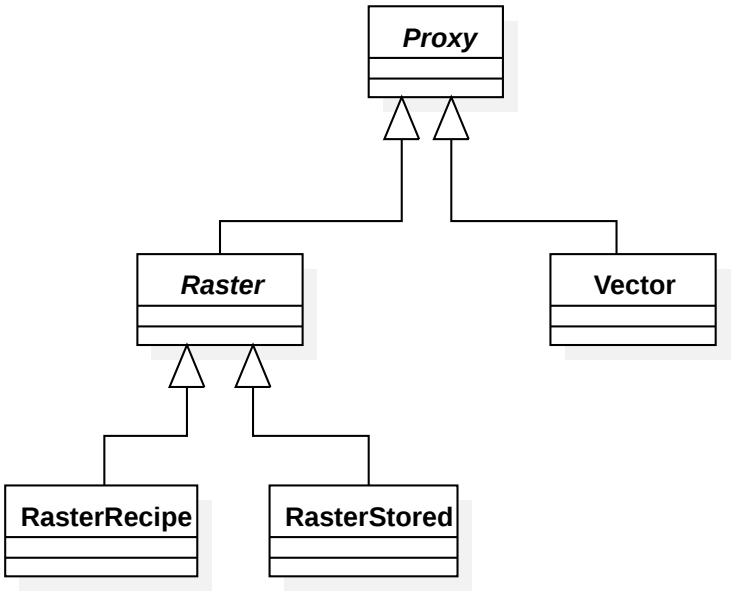

FIGURE 4.1 – Classes *Proxy* de buzzard v0.4.4

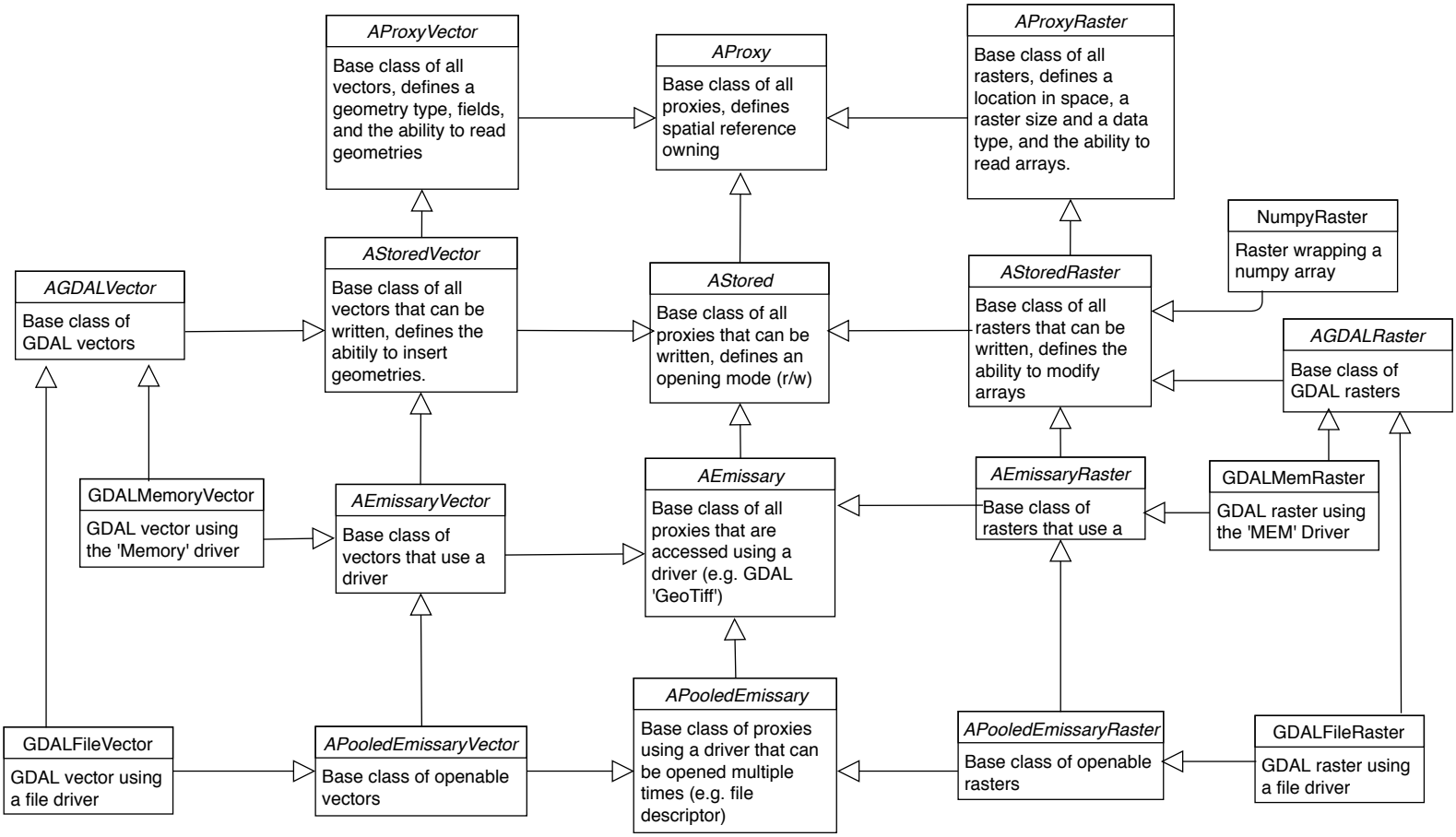

FIGURE 4.2 – Classes *Proxy* de buzzard v0.5.0b0

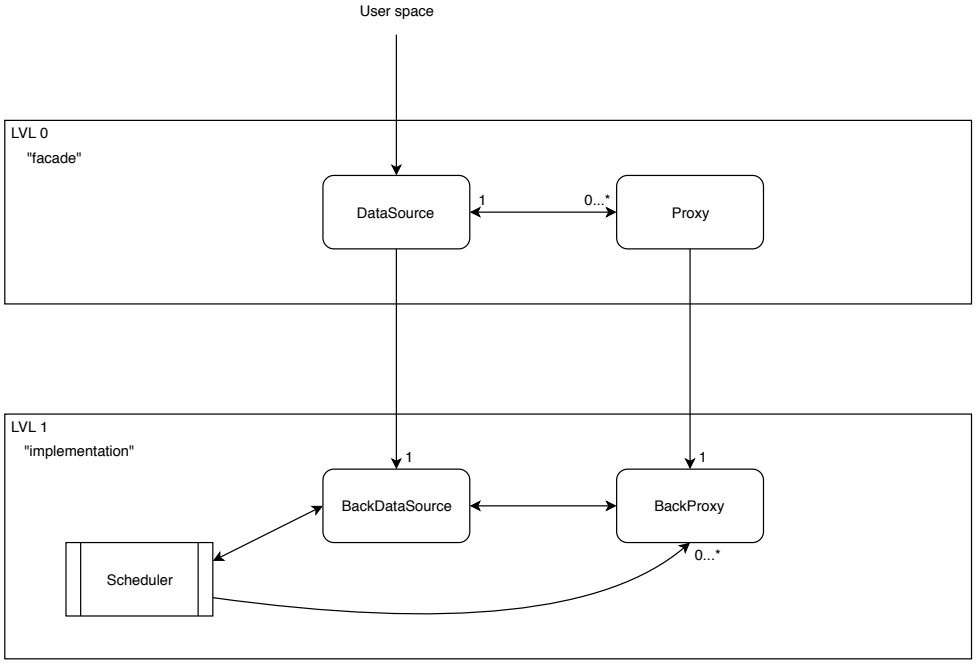

FIGURE 4.3 – Topologie des classes de la version 0.5.0b0

niveau topologique d'implémentation le sont aussi. Cela dissocie le cycle des autres références, et il peut donc être collecté. Ce système est similaire à celui de la séparation entre *Raster* et *BackendRaster* de burito (partie 3.2).

La conséquence de cette topologie est la multiplication par deux du nombre de classes : chaque classe du diagramme de la figure 4.2 (page 26) existe en version façade et implémentation.

4.1.2 Développement de la version 0.5.0 de buzzard

Le développement de cette version est l'intégration à proprement parler de burito à buzzard, qui s'est traduite par la création de nouvelles classes et la modification de la classe *DataSource*.

Pour cette version, l'implémentation de burito a été repensée de zéro. Le point de départ de la réflexion sur l'interaction entre les composantes logicielles de l'implémentation est le modèle d'acteur [10], bien que l'implémentation finale s'en éloigne sur de nombreux points et ne respecte donc pas le modèle.

Les principes sur lesquels se base cette implémentation de burito sont les suivants :

- **Classes Acteur**
  Chaque fonction atomique du programme est définie dans une classe Acteur. Des instances de ces classes sont instanciées une fois par raster, et d'autre une fois par *pool*. Les interactions entre les classes Acteurs sont visibles sur la figure 4.4, page 28.

- **Classe Message**
  Les instances des classes Acteur sont repérées à l'aide d'adresses uniques, et communiquent entre elles par l'intermédiaire de messages, instances d'une classe Message, qui contiennent l'adresse de réception, une en-tête définissant le type de message, et des paramètres permettant l'exécution des tâches créées par la réception du message.

- **Queues asynchrones**
  À l'instar de la version précédente de burito, et comme l'avait conclu mon maître de stage avant mon arrivée, l'ensemble du programme repose sur des queues asynchrones qui permettent l'exécution de plusieurs tâches de nature différente en parallèle tout en rendant disponibles le plus rapidement possible les résultats lorsqu'ils sont prêts.

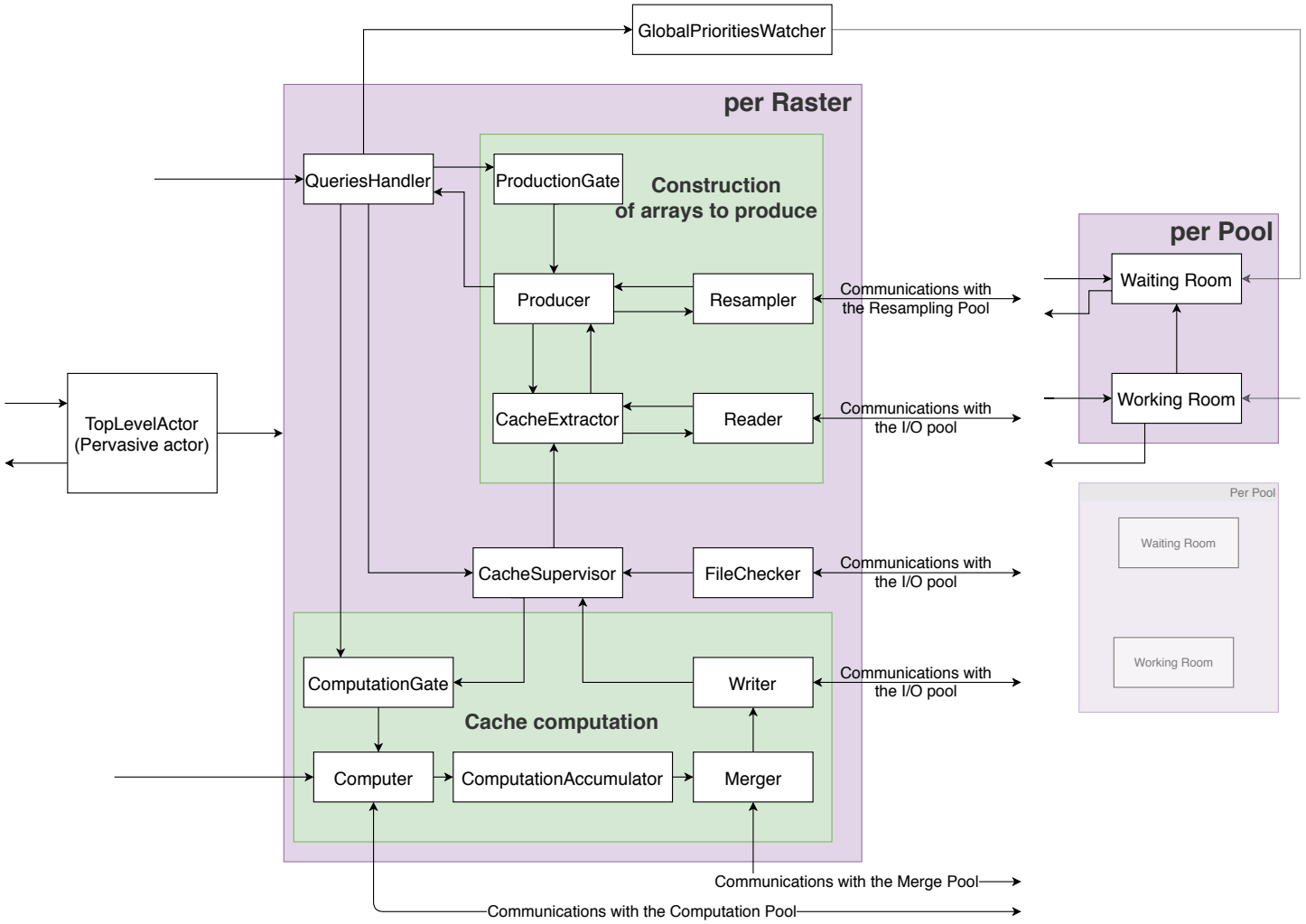

FIGURE 4.4 – **Interaction entre les acteurs de la seconde implémentation de burito.** Deux classes ne sont instanciées qu'une seule fois, certaines classes sont instanciées une fois par raster, et d'autres une fois par *pool*. Les instances communiquent entre elles *via* un système de passage de messages, et sont repérées à l'aide d'adresses.

- **Pools**
  L'outil principal permettant l'exécution en parallèle des tâches est la *pool*, c'est-à-dire un regroupement de fils d'exécution, à laquelle on attribue un type de tâche particulier. Au sein d'un raster, il y a 3 références vers des *pools* (plusieurs références peuvent se faire vers la même pool), pour réaliser 6 types d'opérations, du calcul des données à la lecture de fichiers.
- **Scheduler**
  De la même manière que la version précédente, le programme nécessite un ordonnanceur (ou *scheduler*) synchrone. Or, dans cette version, ce dernier est unique, et non associé à un seul raster. Son rôle est de transporter les messages d'acteur en acteur en cascade.

Par rapport à l'implémentation initiale du burito, ce programme a des avantages significatifs. Tout d'abord, il permet de s'extraire du graphe des dépendances de la preuve de concept, qui, bien qu'utile pour la conception, provoquait des calculs coûteux de parcours de graphe, et rendait la résolution des bugs très difficile. Ensuite, contrairement à ma version originelle, seul un nombre réduit de *threads* est créé (notamment un seul *scheduler*), alors qu'auparavant un *thread* était créé par raster. Enfin, la subdivision du programme en de nombreuses classes atomiques le rend modulaire, et facilement adaptable (par exemple pour des rasters qui n'implémentent pas la mise en cache).

Date d'intégration à buzzard

Étant donné l'état d'avancement de la version 0.5.0 de buzzard à l'heure de l'écriture de ce rapport, le projet n'aura pas été intégré à buzzard lors de mon départ de la société. Il devrait cependant l'être assez prochainement.

4.2 Développement d'un outil de visualisation

Un problème que nous avons rencontré en testant burito est la difficulté à rendre compte du comportement global du programme avec uniquement des affichages console basiques, du fait du nombre important d'interactions entre les objets et de la nature parallèle de l'exécution.

Pour pouvoir mener à bien les tests et le débug de burito, il a fallu développer un outil de visualisation. Nous avons avec mon maître de stage défini des spécifications par le biais d'un schéma JSON, dont des instances sont envoyées par un *WebSocket* lié au programme à un client de visualisation, qui a été développé par un autre membre de l'équipe en utilisant D3.js et React sur le dépôt GitHub suivant : `https://github.com/HerveNivon/hagedash`. Un exemple de visualisation est présenté sur la figure 4.5, page 30.

Cet outil permet de visualiser à intervalles réguliers l'état global de l'exécution du programme *via* des diagrammes circulaires rendant compte de l'utilisation des *threads*, et un diagramme de Sankey affichant les dépendances et les flux entre les rasters.

Le tableau de bord sera intégré à buzzard pour la sortie définitive de la version 0.5.

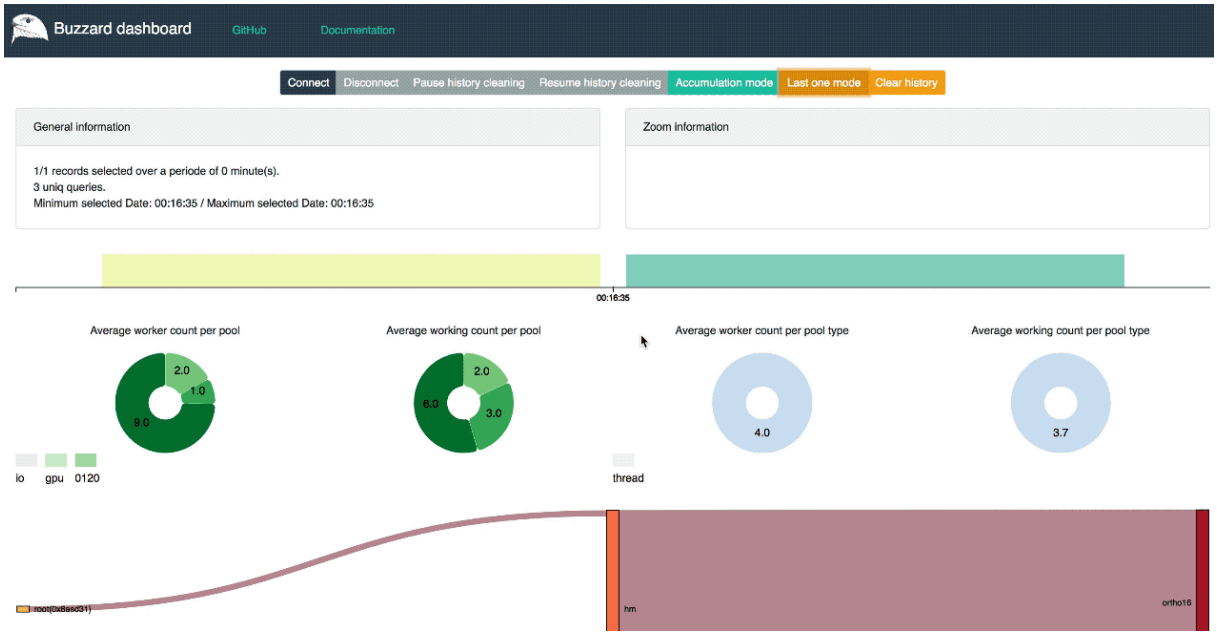

FIGURE 4.5 – *Dashboard* de visualisation de l'état de l'exécution de burito

Chapitre 5
Bilan et perspectives

5.1 Travail réalisé

Durant les cinq mois que j'ai passés au sein de l'équipe *Data Science* d'Airware, j'ai pu réaliser deux tâches majeures.

En premier lieu, j'ai conçu une preuve de concept en réalisant et affinant les spécifications d'un projet permettant la manipulation des raster volumineux utilisés par l'équipe. Ce développement, qui a duré dix semaines, a permis de détecter les principales difficultés de la réalisation du programme tout en prouvant la faisabilité du projet. En tant que preuve de concept, la version finale du projet est très lacunaire et ne peut être intégrée directement à buzzard, la bibliothèque qu'utilise Airware pour traiter les données géographiques.

C'est pour cette raison que la suite et la fin de mon stage ont été consacrées à l'implémentation des spécifications du projet au sein de buzzard. Cela s'est traduit par le développement de la prochaine version de la librairie, la version 0.5, qui implémente de nouvelles fonctionnalités, mais qui revisite surtout l'implémentation des fonctionnalités auparavant existantes.

Cette seconde phase a été réalisée en *pair programming* : mon maître de stage a écrit l'ensemble du code, afin d'assurer sa qualité (il avait le rôle de *conducteur*), et je l'ai assisté dans sa tâche (j'avais le rôle d'*observateur*), notamment durant les phases de réflexion.

5.2 Difficultés rencontrées

Tout au long de mon stage, j'ai rencontré de nombreuses difficultés, causées notamment par la complexité de la tâche et mon manque d'expérience en programmation.

La première difficulté du projet a été de définir précisément les spécifications du projet. En effet, nous savions initialement quelle tâche le projet final devait remplir, mais pas la manière précise de la réaliser. De fait certains points du cahier des charges étaient flous, et ne se sont éclaircis qu'après avoir expérimenté des solutions, et donc au cours du développement de la preuve de concept.

Par ailleurs, la majeure difficulté de ce développement a été d'implémenter toutes les spécifications à la fois. En effet, certaines d'entre elles sont parfois antagonistes (par exemple, la rapidité de calcul et l'économie de mémoire). Il a donc fallu régler les curseurs de satisfaction de chacun des critères du projet.

Certains problèmes insoupçonnés, notamment le problème de *garbage collection* lié à l'ordonnanceur, ont posé des challenges quant au design de l'application. Les designs résultants ont posé à leur tour de nouveaux problèmes : c'est le cas du graphe des dépendances qui, tout en satisfaisant de nombreuses spécifications, a rendu le programme de la preuve de concept extrêmement difficile à débugger. C'est pour cette raison que nous avons eu l'idée d'un tableau de bord pour visualiser l'exécution du programme, car de simples logs textuels ne suffisaient pas. Ce tableau n'a finalement pas servi pour la preuve de concept mais sera utilisé lors du développement de la version 0.5 de buzzard.

Enfin, de nombreuses difficultés que j'ai rencontrées personnellement durant le stage étaient dues à mon manque d'expérience en programmation. En effet, à mon arrivée dans l'entreprise,

j'étais un novice en programmation concurrente, et ne connaissais pas du tout ses implémentations en Python. De plus, mon manque de familiarité avec ce langage a rendu lente mon application des bonnes pratiques de programmation, nécessaires à fournir un code de qualité de production. Dans la seconde partie de mon stage, ce problème a été moins accentué du fait de l'écriture en *pair programming* du code. Cependant, je n'avais jamais été observateur pendant une aussi longue période, et j'ai éprouvé des difficultés à remplir ce rôle.

5.3 Perspectives

Au moment de l'écriture de ce rapport, le développement de la version 0.5 de buzzard est toujours en cours, et aucune version stable n'a encore été déployée. Ainsi, la suite logique du projet auquel j'ai participé durant mon stage est son intégration à buzzard *via* le développement de la version 0.5.

Ce développement nécessite encore plusieurs étapes. La première est la généralisation du modèle présenté dans la partie précédente aux rasters qui n'implémentent pas la mise en cache. Cela devrait être relativement peu complexe du fait de la nature modulaire de la seconde implémentation de burito. Une autre généralisation du processus de burito pourrait être appliquée à tous les rasters issus de fichiers. En effet, leur traitement actuel est potentiellement sous-optimal par endroits.

Une autre fonctionnalité introduite dans la version 0.5 de buzzard sera le tableau de bord décrit dans la partie 4.2. Cet outil permettra la visualisation de l'exécution d'un ensemble de calculs sur des rasters interdépendants, et permettra par exemple de déceler des problèmes dans cette exécution.

Enfin, le développement de la bibliothèque buzzard se fait en continu, comme en témoigne la liste de choses "à faire" de la libraire (disponible à l'adresse `https://github.com/airware/buzzard/wiki/TODO`). Toutes les tâches décrites dans cette liste, qui est agrandie tous les mois, sont loin d'être réalisées, et la nature open-source de buzzard permet à n'importe qui de contribuer à leur avancement.

Conclusion

Airware, et plus précisément l'équipe *Data Science* avait, avant mon arrivée, besoin d'un outil permettant une manipulation optimisée des rasters en Python. La preuve de concept burito est un premier jet d'outil répondant à ce besoin, et la version 0.5 de la bibliothèque open source de manipulation de données géographiques buzzard comportera une implémentation des spécifications de cette preuve de concept.

Cet outil rendra beaucoup plus efficaces les travaux de l'équipe *Data Science*, et rendra plus rapides certains traitements utilisés par les clients de l'entreprise.

Ce stage a été ma première expérience en tant que géomaticien dans une entreprise. J'ai découvert une approche différente de la *data science* par rapport à mon stage précédent qui s'était déroulé en milieu universitaire. J'ai pu y appliquer les compétences acquises durant ma formation, notamment les techniques de gestion de projet. J'ai aussi acquis des compétences en programmation parallèle, en Python, et en programmation en général.

Ce stage a également été pour moi l'occasion de me rendre compte de mes lacunes, notamment au niveau de mes compétences en programmation, dues à mon manque d'expérience dans ce domaine. Néanmoins, malgré ces lacunes, j'ai pu me rendre utile dans l'équipe notamment pendant les périodes de réflexion, et l'encadrement de mon maître de stage m'a permis de progresser sur certains points, rendant mon travail utilisable au sein de l'entreprise.

Bibliographie

[1] *Code source de buzzard*. URL : https://github.com/airware/buzzard (visité le 03/09/2018).

[2] *Documentation de Dask Delayed*. URL : http://dask.pydata.org/en/latest/delayed.html (visité le 29/08/2018).

[3] *Documentation de Dask Distributed*. URL : https://distributed.readthedocs.io/en/latest/ (visité le 29/08/2018).

[4] *Documentation de la bibliothèque GDAL*. URL : https://trac.osgeo.org/gdal/wiki (visité le 16/08/2018).

[5] *Documentation de la librairie standard asyncio*. URL : https://docs.python.org/3/library/asyncio.html (visité le 29/08/2018).

[6] *Documentation de NetworkX*. Bibliothèque de manipulation de graphes en Python. URL : https://networkx.github.io/ (visité le 16/08/2018).

[7] Brendan Fortuner. *Intro to Threads and Processes in Python*. Introduction pour les débutants sur les threads et processes en Python. URL : https://medium.com/@bfortuner/python-multithreading-vs-multiprocessing-73072ce5600b (visité le 16/08/2018).

[8] *Linux Page Cache Basics*. URL : https://www.thomas-krenn.com/en/wiki/Linux_Page_Cache_Basics (visité le 16/08/2018).

[9] *Page Wikipedia en anglais sur l'architecture orientée événement*. URL : https://en.wikipedia.org/wiki/Event-driven_architecture (visité le 16/08/2018).

[10] *Page Wikipedia en anglais sur le modèle d'acteur*. URL : https://en.wikipedia.org/wiki/Actor_model (visité le 16/08/2018).

[11] *Pykka*. Implémentation en Python du modèle d'acteurs. URL : https://github.com/jodal/pykka (visité le 29/08/2018).

[12] *Référence du Global Interpreter Lock (GIL) Python*. URL : https://wiki.python.org/moin/GlobalInterpreterLock (visité le 16/08/2018).

[13] *Référence du langage Python*. URL : https://docs.python.org/3/reference/ (visité le 16/08/2018).

[14] *The Reactive Extensions for Python (RxPY)*. Implémentation Python du reactive programming. URL : https://github.com/ReactiveX/RxPY (visité le 29/08/2018).

[15] *Thespia*. Implémentation en Python du modèle d'acteurs. URL : https://github.com/godaddy/Thespian (visité le 29/08/2018).