Nassim Maali
Arthur Pointel
Hamza Rouar
Guillaume Troesch

Université d'Orléans
M1 IMIS

Rapport Travaux d'études de recherche

Structure de DAG pour le calcul parallèle en Python

2019-2020

Sommaire

1. Etude du sujet

A) Objectif

Le travail présenté dans ce rapport a été réalisé dans le cadre du module Travaux d'études de recherche (TER) par groupe de quatre étudiants, en première année de Master IMIS à l'Université d'Orléans. L'objectif de ce travail est d'implémenter un prototype en Python d'une structure de DAG pour permettre la simulation de fluides. Un premier rapport avait déjà été rendu pour suivre la progression de notre travail. Ce rapport est le rapport final de nos travaux et recherches. Il contribue à approfondir les notions du premier rapport d'avancement.

Un DAG (Directed Acyclic Graph, ou Graphe orienté acyclique en français) est un graphe orienté qui ne comporte pas de circuit. Dans notre cas, un DAG peut être utilisé pour simuler des systèmes sanguins, des réseaux hydrauliques ou encore des trafics routiers. Chaque arc et chaque noeud d'un graphe possède des données relatives à la simulation souhaitée, et l'étude dans le temps permet de représenter l'avancement du système au niveau des jonctions. On appelle une racine du graphe un noeud qui n'a aucune arête orientée vers ce noeud, c'est-à-dire que ce noeud ne contient que des arcs partants de ce noeud. On considère les racines comme les sources des fluides étudiés. On appelle une feuille du graphe un noeud qui n'a aucune arête partant de ce noeud. Les feuilles sont considérées comme les puits des fluides. Dans la suite de notre projet, on supposera qu'aucun noeud n'est racine et feuille à la fois, et toutes les arêtes sont orientées.

Ainsi, notre objectif est de proposer un programme optimisé, rapide et efficace pour permettre à un utilisateur d'étudier l'évolution d'un système de fluide dans le temps, sans qu'il ne se soucie de la manière dont les données sont enregistrées. Il doit pour cela donner cinq éléments en entrée de notre programme : le graphe qu'il souhaite avec les données dont il a besoin sur chaque noeud et sur chaque arête, accompagné de quatre fonctions :

- Une fonction "racine" qui doit retourner à chaque temps comment une racine du graphe est alimentée
- Une fonction "feuille" qui doit retourner à chaque temps comment une feuille du graphe est vidée
- Une fonction "noeud" qui doit retourner à chaque temps la nouvelle valeur d'un noeud du graphe en fonction du graphe au temps précédent
- Une fonction "arête" qui doit retourner à chaque temps la nouvelle valeur d'une arête du graphe en fonction du graphe au temps précédent

Par exemple, si un utilisateur veut utiliser notre travail pour simuler l'écoulement des eaux dans différents bassins, il pourra donner en entrée le graphe (voir Partie 1.B. pour le format du graphe en entrée) avec comme noeuds les bassins avec la quantité maximale d'eau que peut contenir les bassins, et avec comme arêtes les tuyaux entre les bassins avec la quantité maximale d'eau que peut contenir un tuyau. La fonction racine doit indiquer la quantité d'eau ajoutée dans les bassins racines. La fonction feuille doit indiquer la quantité d'eau retirée des bassins feuilles. La fonction noeud doit indiquer comment chaque bassin se vide physiquement dans les tuyaux. Et la fonction arête doit indiquer comment chaque tuyau fait couler l'eau vers un autre bassin.

Les fonctions "noeud" et "racine" sont importantes, car elles peuvent varier selon si un tuyau est défectueux, ou si un bassin laisse échapper de l'eau par exemple. L'écoulement peut aussi varier en fonction d'autres paramètres comme la viscosité ou la densité du fluide étudié.

Notre programme a été réalisé en Python en prévision d'être converti en C/C++. Le premier point de ce rapport explique notre représentation des données du DAG sous Python, pour ensuite s'intéresser à l'évolution d'un système de DAG dans le temps par notre programme, l'optimisation de notre programme par la parallélisation et terminer sur l'évolution future de notre projet.

B) Format de DAG

Le premier objectif de ce projet a été d'établir la représentation des données d'un DAG, tout en considérant une efficacité optimale de l'exécution de notre programme, c'est-à-dire d'optimiser au maximum l'accès aux données du graphe. En informatique, il existe de multiples façons de représenter un graphe. Par exemple, un graphe peut être représenté par sa matrice d'adjacence; ou par une liste de noeuds contenant chacun la liste des arêtes sortantes, elles-même contenant le noeud vers lequel elles pointent; ou encore par le format Compressed Sparse Row (CSR).

Il existe bien d'autres méthodes pour représenter un graphe en informatique. Dans notre cas, et après l'étude de la thèse écrite par Hélène Coullon, nous avons décidé d'utiliser le format CSR pour différentes raisons. Le format CSR consiste à "stocker des matrices creuses de façon relativement légère puisqu'il permet de ne stocker que les éléments qui ne sont pas des zéros", c'est-à-dire de stocker la matrice d'adjacence en ne considérant que les liens entre les noeuds.

Pour cela, le format CSR est constitué de 2 tableaux :

- *index* : tableau de taille nombre de noeuds + 1, qui contient pour chaque noeud i la somme du nombre d'arêtes sortantes des noeuds strictement inférieurs à i, avec i allant de 0 au nombre de noeuds + 1.
  index[i] = somme du nombre d'arêtes sortantes des noeuds strictement inférieurs à i.
- *voisins* : tableau de taille le nombre d'arêtes du graphe (c'est-à-dire aussi la dernière valeur du tableau index), qui contient pour tous les noeuds pris à la suite, leurs noeuds suivants.

Ainsi, dans ce format, si on cherche les noeuds suivants d'un noeud x, on regarde les noeuds qui sont entre *voisins[index[x]]* et *voisins[index[x+1]]* dans le tableau *voisins*. Par la suite, d'autres tableaux ont été ajoutés à cette représentation CSR, pour stocker le flot et la capacité maximale de chaque noeud et de chaque arête, ainsi que la liste des noeuds racines, la liste des noeuds internes et la liste des noeuds feuilles.

Les deux tableaux *index* et *voisins* permettent l'accès à tous les noeuds et arêtes, mais cet accès n'est pas immédiat dans plusieurs cas, par exemple dès l'instant où l'on veut obtenir les arêtes pointant vers un noeud en particulier. Après une étude plus approfondie de notre travail et de la thèse de Hélène Coullon, d'autres données peuvent être enregistrées dans notre format CSR :

- *index_as* : (index des arêtes sortantes d'un noeud) correspond au premier tableau *index* présenté précédemment simplement renommé.
- *voisins_as* : (voisins des arêtes sortantes d'un noeud) tableau de taille le nombre d'arêtes du graphe qui contient pour tous les noeuds pris à la suite, leurs arêtes sortantes.
- *sommets_fils* : (voisins des noeuds suivants d'un noeud) correspond au second tableau *voisins* présenté précédemment simplement renommé.

- *index_ae* : (index des arêtes entrantes d'un noeud) fonctionne de la même manière que *index_as*, mais pour les arêtes entrantes.
  index_ae[i] = somme du nombre d'arêtes entrantes des noeuds strictement inférieur à i
- *voisins_ae* : (voisins des arêtes entrantes d'un noeud) tableau de taille le nombre d'arêtes du graphe qui contient pour tous les noeuds pris à la suite, leurs arêtes entrantes.
- *sommets_par* : (voisins des noeuds précédents d'un noeud) tableau de taille le nombre d'arêtes du graphe qui contient pour tous les noeuds pris à la suite, leurs ,noeuds entrants.

Remarque : sommets_fils et sommets_parents se réfèrent respectivement à index_as et index_ae pour connaître les noeuds suivants et les noeuds précédents.

Ainsi, les arêtes entre *voisins_as[index_as[x]]* et *voisins_as[index_as[x+1]]* correspondent aux arêtes sortantes d'un noeud x.
Les noeuds entre *sommets_fils[index_as[x]]* et *sommets_fils[index_as[x+1]]* correspondent aux noeuds suivants du noeud x.
Les arêtes entre *voisins_ae[index_ae[x]]* et *voisins_ae[index_ae[x+1]]* correspondent aux arêtes entrantes d'un noeud x.
Les noeuds entre *sommets_par[index_ae[x]]* et *sommets_par[index_ae[x+1]]* correspondent aux noeuds suivants du noeud x.

En considérant n=nb de noeuds, et m=nb d'arêtes Le format CSR est bien moins lourd en mémoire qu'une matrice d'adjacence, puisqu'on enregistre 2 tableaux de taille n+1 et m, alors que la matrice d'adjacence enregistre d'une matrice de taille n*n, tout en ayant un accès direct à un élément sans parcours. De plus, comme il s'agit de tableaux, l'accès en mémoire à plusieurs noeuds est plus rapide que dans le cas où le graphe est représenté par des noeuds enregistrés dans les arêtes. Cet accès rapide sera intéressant à prendre en compte lors de l'optimisation et de la répartition des données pour le calcul parallèle.

C) Evolution d'un DAG : 2 études

Un autre objectif de ce projet a été d'imaginer l'évolution d'un système pour écrire un premier programme séquentiel, tout en gardant en tête qu'il devra être parallélisé par la suite.

Une première étude nous a mené à demander à l'utilisateur des fonctions "racines" et "feuilles" identiques à celles présentées en Partie 1, mais également des fonctions "accumulation" et "séparation" sur chaque noeud. De cette manière, chaque noeud peut mettre à jour les arêtes de son voisinage. Cette étude était accompagnée d'une mise à jour entre t et t+1 en partant des feuilles pour remonter aux racines. Ainsi par exemple avec une fonction racine donnant 10 unités au départ, une fonction feuille vidant une feuille dès qu'elle avait du flot, une fonction accumulation disant qu'un noeud prend tout le flot en commençant par les arêtes à gauche, et une fonction séparation disant qu'un noeud donne tout le flot possible en commençant par les arêtes à gauche, on obtient la simulation suivante :

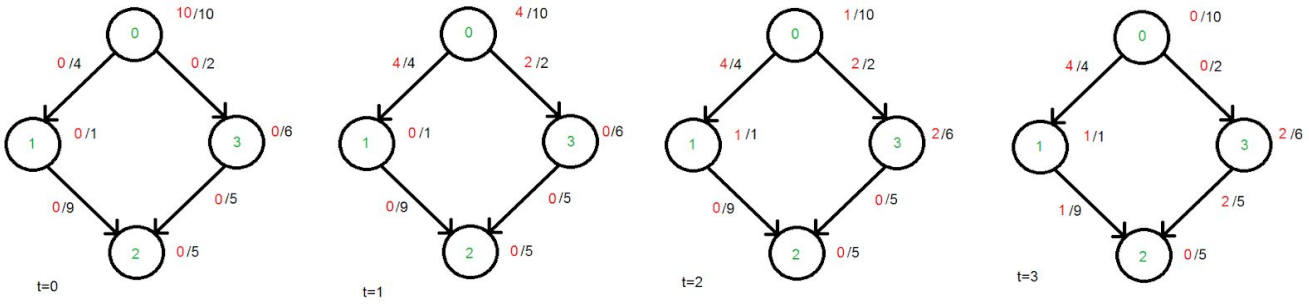


figure 1 : étude d'une évolution, exécution non parallélisable.
Les noms des noeuds sont notés en vert.
La capacité maximale des noeud et des arêtes sont notées en noir.
Le flot dans chaque noeud et chaque arête est noté en rouge.
Le noeud 0 est un racine, le noeud 2 est une feuille, les noeuds 1 et 3 sont des noeuds internes.

La simulation précédente n'est pas parallélisable à cause de l'ordre imposant de traiter les noeuds feuilles pour remonter vers les noeuds racines, et ne correspond donc pas à l'objectif fixé.

En se basant une nouvelle fois sur la thèse d'Hélène Coullon, une nouvelle étude nous a mené à établir les quatre fonctions présentées Partie 1.A, (voir Partie 2 pour l'implémentation). Ainsi, avec la nouvelle simulation, accompagnée par exemple des quatre fonctions suivantes : une fonction racine donnant 10 unités à t=0, une fonction feuille vidant une feuille, une fonction noeud et une fonction arête considérants que tous les flots sont passés entièrement en remplissant les arêtes de gauche en premières, on obtient :

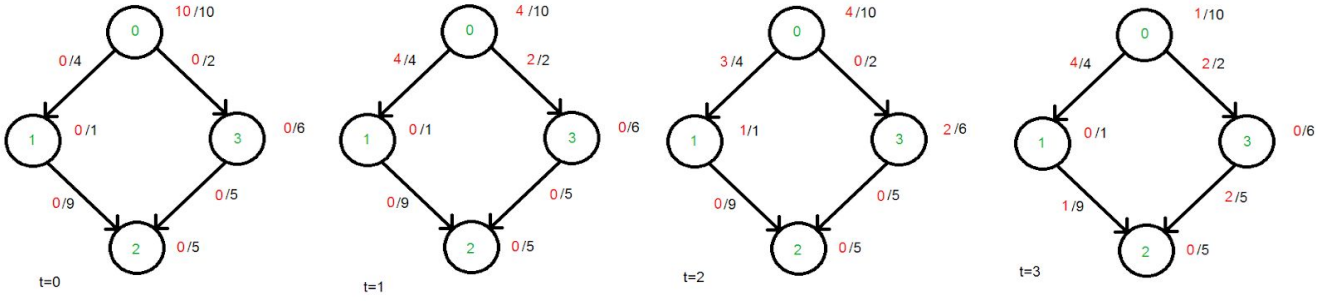


figure 2 : nouvelle étude d'une évolution d'un DAG

Cette évolution considère que chaque noeud se met à jour en fonction de ses arêtes entrantes et sortantes, et que chaque arête se met à jour en fonction de son noeud entrant et son noeud sortant (et dans ce cas des autres arêtes entrantes et sortantes respectivement du noeud sortant et du noeud entrant, voir Partie 2.A. pour plus d'explications). Cette simulation est ainsi considérée dans la suite pour l'implémentation séquentielle en Python, puis l'implémentation parallèle en Python et le début de l'implémentation parallèle en C++.

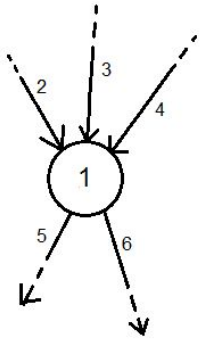
2. Réalisations

A) Interprétation d'un DAG

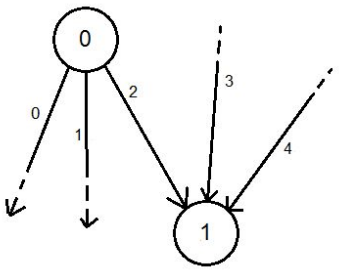
Pour stocker et traiter le graphe fourni par l'utilisateur et le retranscrire dans le format CSR, nous avons développé une fonction d'importation qui demande à l'utilisateur de fournir 3 fichiers. Ces fichiers, un "*.dot*" et deux "*.pmap*" représentent respectivement les arêtes pour le "*.dot*" et les capacités sur les noeuds et les arêtes pour les "*.pmap*". Nous avons adapté notre fonction d'importation de sorte à lire les trois fichiers à la fois et procéder en même temps à la construction du format CSR tel qu'il est décrit dans la première partie.

Pour réaliser les premiers tests, nous avons développé des exemples des quatres fonctions décrites dans la première partie. Ces quatre fonctions ont été implémentées à partir d'un DAG à un temps t pour calculer les nouvelles valeurs du DAG au temps t+1, de manière à simuler différents comportements pour réaliser les tests. A partir d'un DAG en paramètre des fonctions noeuds et arêtes, on peut accéder aux noeuds entrants et sortants d'une arête, et aux arêtes sortantes et entrantes d'un noeud.

Les fonctions noeuds et arêtes demandent des informations des noeuds et arêtes de son voisinage. Plus précisément, pour suivre son évolution, un noeud a besoin de ses arêtes entrantes et de ses arêtes sortantes. Une arête a besoin de son noeud entrant, de son noeud sortant, des arêtes sortantes de son noeud entrant (appelées arêtes parallèles sortantes dans la suite du rapport) et des arêtes entrantes (appelées arêtes parallèles entrantes). Les exemples suivants illustrent ces informations :



A partir d'une fonction "noeud", ici, pour calculer son évolution, le noeud 1 a besoin de ses arêtes entrantes 2, 3 et 4, et de ses arêtes sortantes 5 et 6.



A partir d'une fonction "arête", ici, pour calculer son évolution, l'arête 2 a besoin de récupérer les données du noeud 0, du noeud 1, et de toutes les arêtes 0, 1, 3 et 4.

Par exemple, en considérant que le flot est réparti équitablement à l'entrée et à la sortie d'un noeud, pour calculer le nouveau flot d'un noeud à l'aide de la fonction "noeud", le noeud devra d'abord estimer ce que chaque arête peut lui donner en simulant cette répartition. Le noeud devra également estimer le flot qu'il peut actuellement passer à ses arêtes sortantes. Finalement, le noeud aura comme nouvelle valeur son ancienne valeur ajoutée à celle qu'il peut prendre puis soustraite à celle qu'il peut donner.

Pour une arête avec ce même exemple, l'arête devra simuler ce que son noeud entrant peut lui donner, c'est-à-dire refaire le même calcul que lorsque le noeud estime sa répartition sur plusieurs arêtes. L'arête devra ensuite simuler ce que son noeud sortant peut lui prendre, c'est-à-dire refaire le même calcul que lorsque le noeud estime l'accumulation de flot de plusieurs arêtes.

D'autres exemple peuvent être implémentés pour simuler un écoulement de fluides. Parmis ces autres exemples, on peut imaginer que les fonctions feuilles peuvent vider une feuille entièrement, petit à petit ou pas du tout; les fonctions noeuds et arêtes peuvent considérer que les noeuds distribuent et récupèrent du flot équitablement sur toutes les arêtes, ou ne récupèrent rien à certains temps. A partir de ces quatre fonctions un CSR à un temps t+1 peut être calculé à partir de l'algorithme séquentiel suivant:

```
#Racines
for idNoeud in csr.getNoeudsRacines():
    (flot, capacite) = csr.getValeurNoeud(idNoeud)
    #fctNoeud
    new_flot1 = fctNoeuds(temps, idNoeud, csr)
    #fctRacine
    new_flot2 = fctRacines(temps, idNoeud)

    valeurNoeuds2[idNoeud] = (min(new_flot1+new_flot2, capacite), capacite)

#Internes
for idNoeud in csr.getNoeudsInternes():
    (flot, capacite) = csr.getValeurNoeud(idNoeud)
    #fctNoeud
    new_flot1 = fctNoeuds(temps, idNoeud, csr)

    valeurNoeuds2[idNoeud] = (new_flot1, capacite)

#Feuilles
for idNoeud in csr.getNoeudsFeuilles():
    (flot, capacite) = csr.getValeurNoeud(idNoeud)
    #fctNoeud
    new_flot1 = fctNoeuds(temps, idNoeud, csr)
    #fctFeuille
    new_flot2 = fctFeuilles(temps, idNoeud, flot)

    valeurNoeuds2[idNoeud] = (min(max(new_flot1 - new_flot2, 0), capacite), capacite)

#Aretes
for idArete in range(len(csr.getValeursAretes())):
    #fctArete
    new_flot = fctArete(temps, idArete, csr)

    valeurAretes2.append((new_flot, csr.getValeurArete(idArete)[1]))
```

figure 3 : script Python du programme principal séquentiel, passer à un DAG au temps t+1 Ici, csr correspond au DAG au temps t. valeurNoeuds2 et valeurAretes2 correspondent aux tableaux mis à jour pour le DAG au temps t+1.

Afin que l'utilisateur puisse visualiser l'avancement de l'écoulement de fluide, une fonction d'exportation du graphe CSR au format *".dot"* est utilisée. Cette fonction peut être appelée à la fin de la simulation de chaque t+1 pour pouvoir suivre cette simulation pas à pas.

B) Répartition pour processeurs

Pour obtenir un programme plus efficace, un autre objectif de notre travail est d'exécuter celui-ci en parallèle. Parmis les choix de parallélisation possibles, nous pouvions choisir la méthode de mémoire distribuée avec OpenMP, cela reviendrait globalement à paralléliser les quatres boucles présentées sur la figure 3 du programme principal (Partie A.3.). Nous avons décidé de réaliser cette parallélisation par la méthode de mémoire partagée avec MPI, ici adaptée pour Python avec le module mpi4py.

Par mémoire partagée, la première étape est de partager nos données pour les envoyer à plusieurs processeurs. Chaque processeur devra contenir des données équivalentes pour effectuer des calculs similaires, c'est-à-dire qu'un processeur devra s'occuper des calculs sur autant de noeuds et d'arêtes qu'un autre processeur. A partir de l'étude de la thèse de Hélène Coullon et de nos productions, une répartition optimale pour effectuer ces calculs est de donner à un processeur des noeuds et des arêtes liés en "blocs" pour permettre l'accès aux données directement au sein d'un même processeur.

Chaque processeur devra être informé des noeuds et des arêtes qu'il a à traiter. D'après la Partie 2.A, chaque noeuds a besoin de ces arêtes entrantes et sortantes, et chaque arêtes a besoin de ses noeuds entrants et sortants et de ses arêtes parallèles entrantes et sortantes pour être traités. Chaque processeur devra donc également être informé des noeuds et des arêtes dont il a besoin pour traiter ses noeuds et ses arêtes, ces noeuds et arêtes sont dit extérieurs au processeur. Enfin, pour que le processeur soit également informé des liaisons entre les noeuds et arêtes à traiter et extérieur, le processeur doit être informé du "bloc", c'est-à-dire un morceau du CSR donné par l'utilisateur. Ce morceau forme un nouveau CSR plus petit que le CSR initial.

Pour partager les données, nous avons développé un algorithme dans un premier temps pour montrer que cet algorithme est faisable, mais n'est pas parfait et mériterait sûrement un approfondissement dans son concept. L'idée choisie ici est d'abord de partager les arêtes puis les noeuds en ajoutant le moins d'arêtes extérieures, puis le moins de noeuds extérieurs. L'algorithme va choisir un groupe d'arêtes à traiter puis choisira des noeuds à traiter liés à ces arêtes.

Puisque ajouter des arêtes extérieurs signifie d'abord d'ajouter des arêtes parallèles à une arête traitée, la première étape de l'algorithme consiste donc à trouver le noeud qui possède le plus grand degré, c'est-à-dire le plus d'arêtes entrantes ou sortantes. En cas d'égalités, cette première recherche privilégie les noeuds feuilles ou racines, pour espérer obtenir le moins de noeuds esseulés à la fin de cet algorithme de séparation.

Une fois le premier groupe d'arêtes entrantes ou sortantes d'un noeud choisi, l'algorithme va chercher toutes les arêtes candidates pour être ajoutées au traitement du processeur. Les arêtes candidates sont les arêtes (entrantes ou sortantes, ici la distinction n'a pas d'importance) liées aux noeuds incidents aux arêtes déjà choisies. L'algorithme compte ensuite le nombre de noeuds et d'arêtes extérieurs que chaque ajout d'arête à traiter engendrerait, et choisit l'arête qui en engendre le moins. Cette heuristique implique que l'algorithme est tenté de choisir en priorité les arêtes liées à des noeuds feuilles ou racines, pour obtenir le moins de noeuds esseulés. Si aucune arête candidate n'est trouvée, l'algorithme fouille tout le graphe en gardant la règle impliquant de n'ajouter le moins d'éléments extérieurs possibles, résultant dans ce cas à des sous-graphes non connexe.

L'étape précédente est répétée tant que le processeur accepte des arêtes, pour avoir un nombre d'arêtes à traiter équivalent aux autres processeur. Une fois que l'algorithme a établit les arêtes que le processeur doit traiter, l'algorithme va choisir les noeuds parmis les noeuds extérieurs requis pour les arêtes choisies. L'algorithme suppose ici que le nombre de noeuds extérieurs est suffisant pour ne pas avoir à chercher de noeuds autre part dans le graphe.

Pour choisir les noeuds, l'algorithme donne un score à chaque noeud et choisi celui avec le meilleur score. Ce score est plus élevé en fonction d'un nombre d'arêtes extérieures ajoutées bas. Ce score est également incrémenté si on considère le noeud lié au premier groupe d'arêtes choisies. Le score permet donc la priorité sur les noeuds n'ayant aucune arête extérieure engendrée, incluant donc une priorité sur les noeuds feuilles ou racines, puis sur les noeuds ayant le moins d'arêtes extérieures en priorisant le noeud incident au tout premier groupe d'arêtes choisis au début de l'algorithme.

Cette étape est répétée tant que le processeur accepte des noeuds (voir Annexes 1 et 2 pour un exemple de répartition avec cet algorithme). Une fois les noeuds et les arêtes choisis, un nouvel algorithme permet de définir les noeuds et les arêtes que chaque processeur doit envoyer et recevoir de chaque processeur, en fonction des noeuds et des arêtes que chaque processeur doit calculer. Un troisième algorithme recompose ensuite les morceaux de DAG pour chaque processeur, puisque la répartition a coupé des arêtes entrantes et sortantes des noeuds.

C) Paraléllisation, communication

Lorsque les données du DAG ont été divisé de manière “équitable”, elles sont prêtes à être transférées aux processeurs pour simuler le DAG jusqu'à un instant t définit par l'utilisateur.

Le processeur racine qui a créé le DAG global va appeler les fonctions de répartitions du DAG global pour diviser celui-ci en plusieurs "sous-DAG". Le nombre de sous-DAG est égal au nombre total de processeurs. Ils sont chacuns envoyés à leur processeur respectif. En plus d'un sous-DAG, un processeur reçoit aussi un dictionnaire qui contient les éléments à envoyer et à recevoir de chaque autre processeur, ainsi que deux autres tableaux cumulatifs qui contiennent le nombre de chaque type de noeuds cumulé avec le nombre de noeuds précédents pour l'un et d'arêtes pour l'autre. On peut distinguer 5 types de noeuds déduits de la thèse de Hélène Coullon :

[*locaux_internes* | *locaux_externes* | *feuilles* | *racines* | *noeuds externes*]

Ces noeuds sont définis dans l’ordre lors de la construction d’un sous-DAG. On peut donc parcourir chaque type en trouvant l’intervalle d’indices auxquelles ils sont stockés. Le but des tableaux cumulatifs est de retrouver ces intervalles. Ainsi, le tableau cumulatif des noeuds va contenir le nombre de noeuds locaux internes à son premier indice, puis celui des noeuds locaux externes cumulé avec le nombre de locaux internes à son deuxième, et ainsi de suite. Son cinquième et dernier indice contient le nombre de noeuds externes cumulés avec tous ceux d’avant, soit le nombre de noeuds total du sous-DAG. Par exemple, pour trouver l’intervalle d’indice des feuilles, il faut regarder dans le tableau cumulatif à son deuxième et à son troisème indice. L’intervalle est donc définit comme suit :

[*tab_noeud_cumu[2]* ; *tab_noeud_cumu[3]*]

Ce procédé est le même pour les arêtes qui sont triées par type dans le DAG : celles locales et celles externes au processeur.

[*arêtes locales* | *arêtes externes*]

On retrouve le même tableau cumulatif que pour les noeuds pour retrouver l’intervalle d’indice des arêtes internes ou externes.

Une fois tous ces éléments envoyés à chaque processeur, la fonction de simulation est appelée. Cette fonction va d'abord initialiser les données nécessaires à la communication entre les processeurs. Il est important de préciser que le format de données n'est pas encore tout à fait optimisé. En effet, étant donné que nous avons révisé certaines parties de notre travail, il existe certaines structures de données à revoir. Pour plus de facilité dans le développement de la parallélisation,

certaines données fournies en entrée sont retraitées pour être dans un format plus pratique pour effectuer les communications entre processeurs. Cette structure est disséminée dans certaines parties importantes du code et nous avons préféré convertir ces données dans un autre format.

Lorsque toutes les données sont chargées, chaque processeur va rentrer dans sa boucle de simulation. Chaque itération de cette boucle est une simulation de l'instant t basé sur t-1. Chaque processeur va traiter les noeuds qui lui sont possibles de traiter, c'est-à-dire les noeuds locaux internes. Il peut ensuite envoyer une partie de ses données aux processeurs qui les demandent. Les fonctions “mpi4py” de communications entre processeurs isend(...) et irecv(...) sont utilisées car elles sont non-bloquantes. Un processeur va envoyer avec des “isend” toutes les informations que lui demandent les processeurs dépendants. Etant donné que isend est non bloquant, il n’y a pas besoin d’avoir finalisé l’envoi pour qu’il demande lui-même à recevoir.

Pour continuer l'exécution de son programme, il faut qu'un processeur ait reçu toutes les informations qu'il a demandé ainsi que celles qu’il a envoyé. Il y a donc un requete.wait() sur les requêtes de réception et d'envoi. La fonction wait() sur toutes les requêtes d’un processeur bloque la suite du programme si au moins une requête n’est pas finalisée. Quand un processeur a reçu et envoyé toutes les informations nécessaires, il va appliquer le reste des fonctions sur les racines, les feuilles, les noeuds locaux externes puis sur les arêtes. Il n’y a pas d’ordre de traitement précis car chaque traitement d’un type de données est indépendant. Ces instructions sont exécutées “temps_max” fois, où “temps_max” est le paramètre de temps donné par l'utilisateur.

Une fois la simulation finie, il faut évidemment restituer le graphe d'origine avec les nouvelles valeurs sur chaque noeud et arête. Etant donné que la structure interne du graphe global ne change jamais, elle est conservé du début à la fin dans le processeur racine. Stocker le graphe global durant toute l’exécution peut être assez coûteux en mémoire, il s’agit ici d’un choix glouton que nous avons fait pour éviter de reconstituer toute la structure du DAG initial à partir des sous-DAG.

Pour obtenir le graphe final, le processeur racine va demander à chaque processeur son sous-DAG local final. Lorsqu’il a obtenu tous les sous-DAG, il va seulement remplacer la valeur initiale de chaque noeud et arête par la valeur trouvée dans le sous-DAG qui contient ce noeud ou cette arête.

D) Python vers C++

Python avait dans un premier temps été utilisé dans ce projet pour faciliter le développement, l'écriture et la lisibilité des algorithmes. Python étant un langage de plus haut niveau que C++, et intégrant un garbage collector, il fonctionne à l'aide d'un système de mémoire allouée dynamiquement, et perd donc en efficacité de traitement de la mémoire. Dans le but d'éviter les défauts de mémoire, d'optimiser les calculs parallèles et les communications entre les différents processeurs, C++ est donc une meilleure solution pour sa performance et sa rapidité par rapport à python, notamment en matière de temps d'exécution.

Comme le travail présenté jusqu'à présent est réalisé en Python, nous avons donc besoin d'une solution qui nous permettra d'invoquer des fonctions écrites en C++ depuis notre code Python. De cette façon, le calcul et la répartition des données seront réalisées en Python, pendant que les calculs parallèles et les communications entre les différents processeurs seront gérés en C++.

Pour réaliser cette tâche, nous avons utilisé Cython qui est une extension de Python pour compiler des instructions Python avec le typage C. Avec une syntaxe très proche de celle de Python, cette extension est également un langage, et supporte à la fois l'écriture Python mais aussi un sous-ensemble du langage C et C++ comme la déclaration de variables et de fonctions. Ce langage permet d'écrire des extensions compilées pour Python, une fois l'écriture des fonctions de la parallélisation en C++ est terminée et après adaptation dans le fichier Cython, ce dernier après compilation des fichiers produira un module directement exploitable sous Python.

N'ayant pas eu le temps nécessaire pour approfondir notre travail sur ce point, nous tenions tout de même à nous intéresser à la conversion avec Cython. Dans cette partie, nous avons commencé par définir une classe 'Process' qui contient toutes les méthodes et paramètres nécessaires aux calculs et à la communication entre les différents processeurs. Elle contient également une fonction de simulation qui pour un noeud racine distribue son flot équitablement sur ses voisins, pour un noeud interne récupère le flot sur ses arêtes entrantes et le distribue sur ses arêtes sortantes et enfin pour un noeud feuille récupère le flot sur ses arêtes entrantes, et donc cette fonction ne permet pas d'appliquer les fonctions données par l'utilisateur.

3. Perspectives

A) Récapitulatif

Pour récapituler tout notre travail et pour faciliter une approche ultérieure de notre projet par d'autres personnes, cette partie rassemble les informations nécessaires :

- La classe **CSR.py** rassemble les 6 tableaux nécessaires présentés en Partie 1.B. et possède les getters utiles pour les accès extérieurs.
- Le fichier **readFile.py** rassemble les fonctions de lecture et d'écriture des fichiers. La fonction de lecture étant réalisée au tout début du projet, n'ayant pas été retravaillée depuis les multiples changements de directions de notre travail, elle reste fonctionnelle, mais peut ne pas être parfaite car ce n'est pas l'objectif central du projet. La lecture n'est possible qu'avec un groupe de 3 fichiers : .dot et deux .pmap (voir Partie 2.A).
- Le fichier fonctions_old.py rassemble les fonctions utilisateurs avec l'étude de l'évolution impliquant les 4 fonctions "racines", "feuilles", "accumulation" et "séparation". Ce fichier n'est plus utilisé, mais peut rester utile dans le cas d'une nouvelle étude avec ces fonctions. Ce fichier ne compile plus en raison de la modification du format CSR en 6 tableaux, et n'a pas été mis à jour car il ne s'agit plus de la direction de notre projet.
- Le fichier main_old.py permettait d'établir l'évolution du DAG de manière séquentielle avec les fonctions_old.py. Ce fichier ne compile plus.
- Le fichier **fonctions.py** présente des exemples fonctionnels des fonctions "racines", "feuilles", "noeud" et "arête".
- Le fichier **main.py** utilise les fonctions.py pour établir une évolution du DAG de manière séquentielle. Ici, tout est fonctionnel.
- Le fichier **repartition.py** inclut les fonctions en Partie 2.B. pour la répartition du DAG sur différents processeurs. La première fonction de répartition était basée sur des choix réfléchis pour inclure le moins de noeuds et d'arêtes extérieurs, mais ces choix peuvent être remis en question. De plus, une hypothèse a été émise, impliquant qu'après le choix des arêtes, le nombre de noeuds inclus dans le processeur est suffisant pour en choisir parmis ceux-ci afin d'être traités par le processeur. Cette hypothèse peut être réfutée. Ce fichier inclut également la fonction permettant de convertir cette répartition en des instructions pour les envois et les réceptions entre les processeurs. Cette fonction produit actuellement des dictionnaires, mais on s'est rendu compte qu'elle peut envoyer des tableaux sous la forme index/voisins comme les tableaux de CSR. Ce fichier inclut également la fonction de réindexation (Partie 2.C). Ce fichier inclut également la fonction de re-création de chaque sous-CSR pour chaque processeur à partir de la réindexation.

- Le fichier **mainPara.py** utilise la répartition.py, les fonctions.py, le format CSR.py et le readFile.py pour présenter les communications en processeurs et l'évolution du DAG de manière parallèle.
- Le dossier **cython** présente les débuts de l'exploration du programme par Cython, notamment la classe Process.

B) Améliorations possibles

Les parties précédentes ont montré que nous avons réussi à proposer un programme capable de simuler l'évolution d'un DAG dans le temps de manière séquentielle puis parallèle. Le format CSR a été montré comme le plus efficace pour stocker et accéder aux données. Cependant, en raison d'un manque de temps pour ce projet, quelques aspects mineurs peuvent être perfectionnés. Cette partie donne de nouvelles indications sur l'évolution future et reprend également des points cités tout au long du rapport.

Tout d'abord, puisque le programme a été amélioré et repensé tout au long du projet, il aurait été intéressant d'étudier en détails le temps d'exécution du programme. Parmis ces études, et pour reprendre les tests d'Hélène Coullon dans sa thèse lorsqu'elle cite le "recouvrement", la réindexation au sein d'un processeur peut être un aspect intéressant à considérer.

Dans notre projet, Cython n'a été testé que pour réaliser la parallélisation. Il aurait fallu de meilleures connaissances sur ce sujet pour exploiter cet outil au mieux et permettre l'accès aux données rapidement, tout en évitant les allocations dynamique tout au long de l'exécution du programme et éviter les défauts de mémoire.

Le format CSR actuel ne permet d'enregistrer que les flots et les capacités des noeuds et des arêtes sous la forme de tuples en Python. Une nouvelle méthode pour enregistrer les données sur un noeud ou une arête pourrait être d'utiliser des tableaux de taille indéfinie, pour pouvoir considérer par exemple qu'un noeud peut enregistrer toutes sortes données, comme la densité ou la viscosité du fluide.

Lors du développement de notre programme, nous avons remarqué qu'il était étrange que les fonctions noeuds et arêtes fassent un travail similaire, comme de repartir du flot équitablement sur des arêtes à partir d'un même noeud. Ce travail similaire est effectué sur chaque noeud, mais également à partir de chaque arête. Cette question ne s'était pas posée lors de la première étude avec les fonctions racines, feuilles, accumulation et répartition (revoir partie 1.C. pour cette étude), mais cette étude avait été mise de côté car elle ne pouvait pas être parallélisée en raison de l'ordre imposé de traitement des noeuds. Une étape supplémentaire serait donc de réfléchir à modifier le programme ou d'en créer un nouveau qui n'exécute pas ce travail en double, mais toujours avec la possibilité d'être parallélisé.

L'algorithme de répartition étant basé sur des choix réfléchis pour inclure le moins de noeuds et d'arêtes extérieurs, ces choix et les étapes menant au résultat peuvent être remis en question. Cet algorithme a des chances d'isoler des noeuds feuilles ou racines, ce qui engendre de nombreux noeuds et arêtes extérieurs.

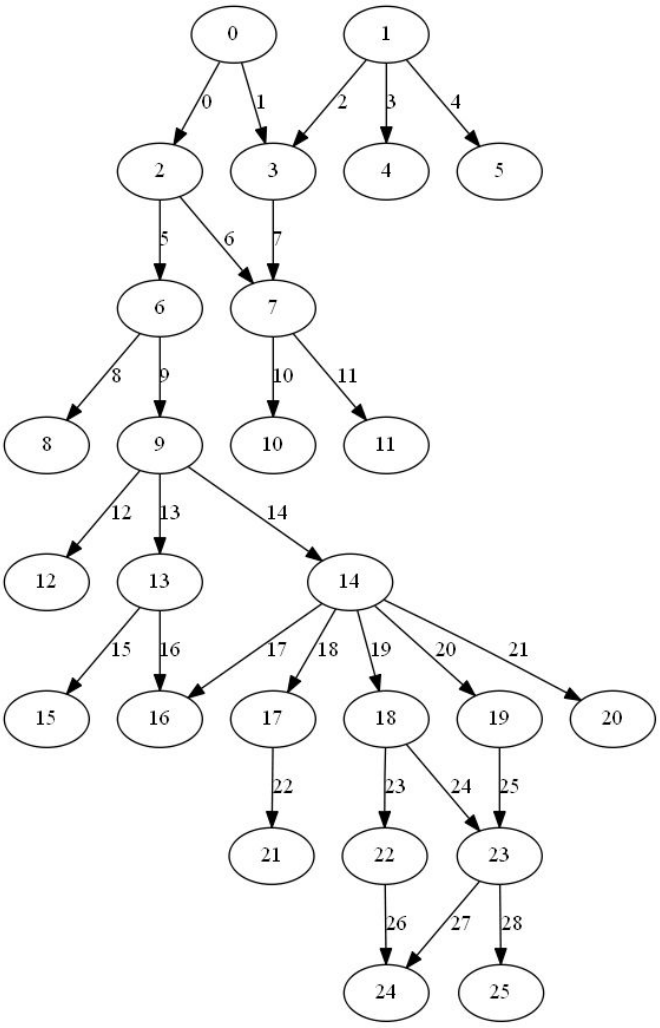
Un point cependant important à modifier concerne l'hypothèse impliquant qu'après le choix des arêtes, le nombre de noeuds inclus dans le processeur est suffisant pour en choisir parmis ceux-ci afin d'être traités par le processeur. Cette hypothèse peut être réfutée, c'est-à-dire que le programme devrait être capable de chercher des noeuds à l'extérieur du blocs d'arêtes déjà conçu lorsqu'il ne trouve plus de noeuds dans ce bloc.

L'algorithme parallélisé traite le DAG à partir de structures de données différentes et sont modifiées de nombreuses fois. En effet, à la suite de l'algorithme de répartition, on obtient des listes des noeuds et arêtes à traiter et des noeuds et arêtes à recevoir. Ces listes sont ensuite modifiées pour devenir des dictionnaires avec comme clés les processeur à communiquer et les éléments à envoyer ou recevoir. Et par la suite ces éléments sont reconvertis en listes au sein d'un processeur. Toutes ces conversions peuvent être remplacés, et les éléments peuvent être enregistrées sous la forme de deux tableaux index/voisins, plutôt que de manipuler toutes sortes de structures différentes. Par exemple, pour la liste des noeud à recevoir, la liste index de taille n contiendrait le nombre d'éléments à recevoir des processeurs 0 à n, et la liste voisins contiendrait les éléments demandés aux processeurs 0 à n.

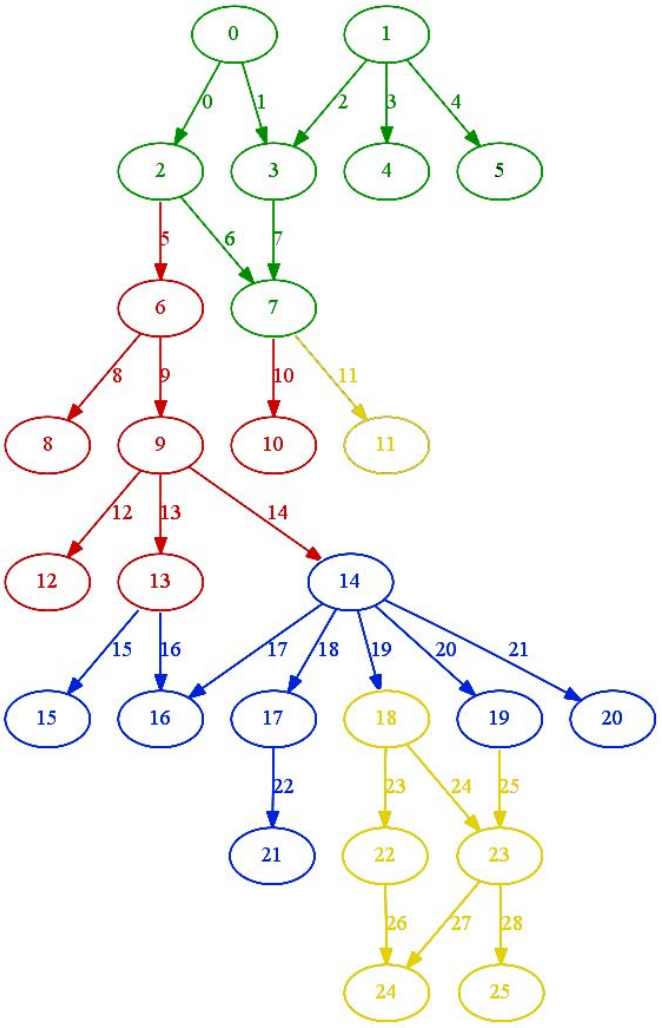
Pour terminer, le programme ne propose la lecture de DAG qu'à partir de 3 fichiers. L'expérience utilisateur peut être améliorée de ce côté. Un DAG peut être représenté par un simple fichier .dot, de la même forme que l'exportation d'un DAG sous un .dot. Cette fonctionnalité avait été commencée, il reste des traces de cette fonctionnalité dans readFile.py, mais a été mise de côté pour se concentrer sur l'aspect central du projet.

Annexes

Annexe 1 : exemple de répartition d'un graphe sur 4 processeurs

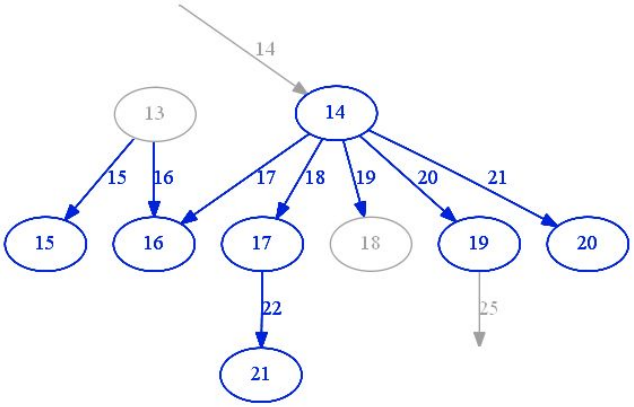


DAG initial

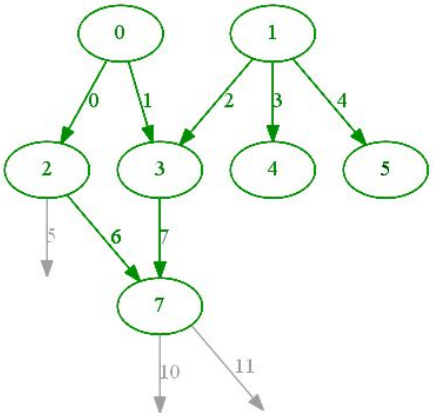


DAG séparé sur 4 processeurs.
Chaque couleur représente les arêtes et les noeuds à traiter sur chaque processeur.

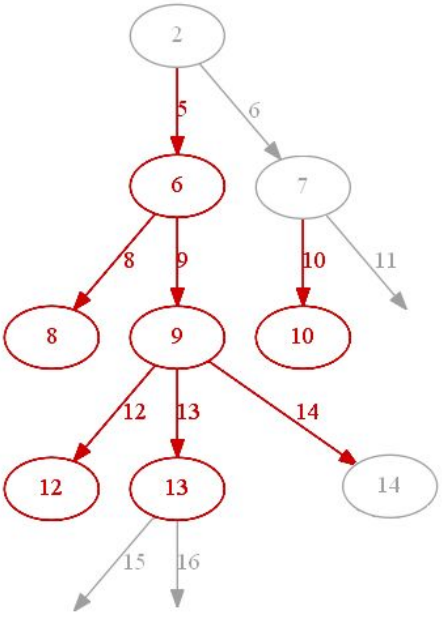
Annexe 2 : nouveaux CSR avec exemple précédent sur 4 processeurs



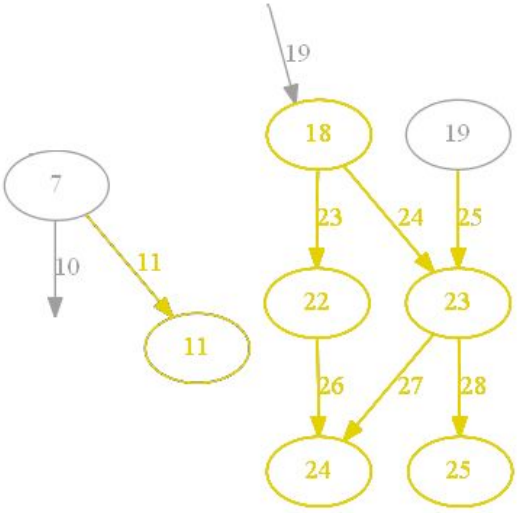
Sur processeur 0



Sur processeur 1



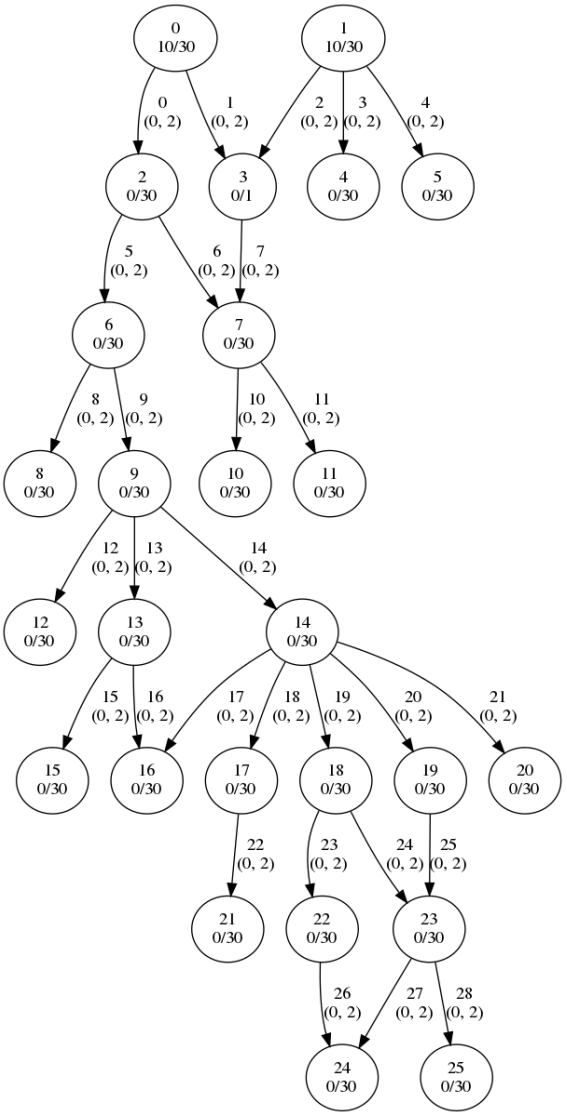
Sur processeur 2



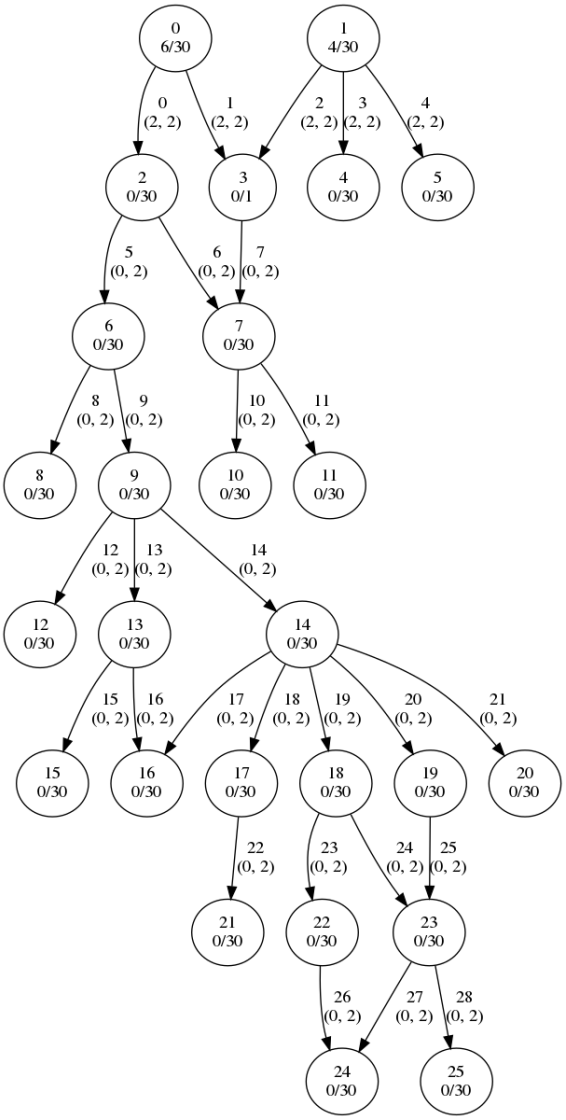
Sur processeur 3

Les arêtes et les noeuds colorés sont à traiter par le processeur. Ceux en gris sont extérieurs et sont requis pour les traitements. Des noeuds “fictifs” (ici non-représentés) sont ajoutés dans les CSR locaux à chaque processeur pour éviter d’avoir des arêtes qui ne pointent sur rien. Dans la simulation, ces noeuds sont inutiles, et ne sont jamais appelés, et possèdent tous l’identifiant -1.

Annexe 3 : Exemple d'évolution d'un DAG dans le temps

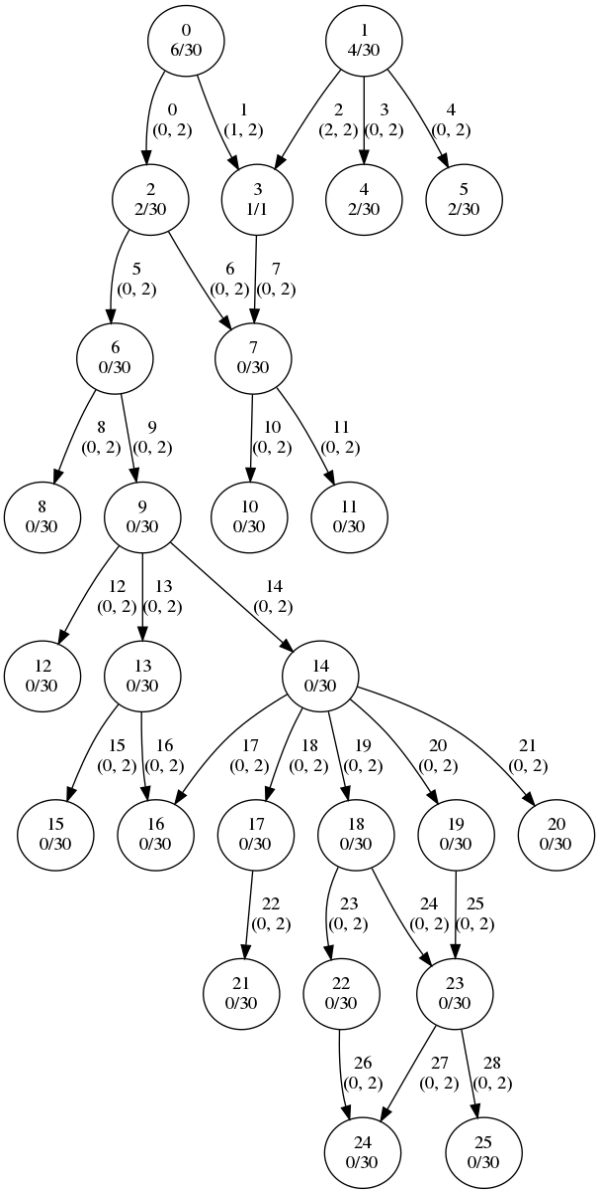


DAG au temps t=0

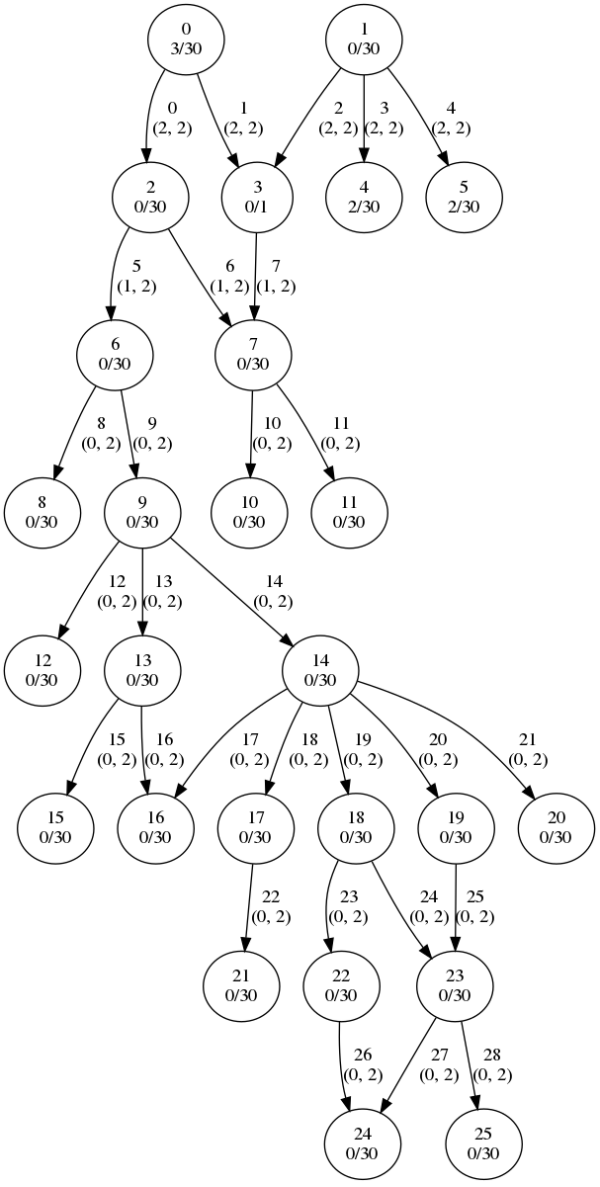


DAG au temps t=1

Chaque noeud de ce DAG a une capacité de 30 (sauf le noeud 3 qui a une capacité de 1).
Chaque arête de ce DAG a une capacité de 2.
La fonction racine impose 10 unités au temps t=0, puis ne retourne plus rien.
La fonction feuille impose de garder le flot dans le noeud.
Les fonctions noeuds et arêtes imposent une répartition équitables entre les arêtes entrantes et sortantes des noeuds.



Dag au temps t=2



Dag au temps t=3

Bibliographie

Modélisation et implémentation de parallélisme implicite pour les simulations scientifiques basées sur des maillages, thèse de Hélène Coullon, 29 septembre 2014
http://helene-coullon.fr/download/coullon_helene.pdf

Cython : https://cython.readthedocs.io/en/latest/

Utilisation de C++ en Cython:
https://cython.readthedocs.io/en/latest/src/userguide/wrapping_CPlusPlus.html#using-c-in-cython