



Ordonnancement parallèle de tâches sur un graphe orienté acyclique

Contexte

Un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est dit acyclique s'il ne possède pas de circuit (cycle orienté), i.e. s'il n'existe pas de chemin partant de v_i qui revienne sur v_i . Un tel graphe est alors nommé DAG, de l'anglais *Directed Acyclic Graph*.

Il est souvent utile de modéliser un problème par un ensemble de tâches v_i à réaliser. Parfois, une tâche v_i ne peut être commencée que lorsque une autre tâche v_j est terminée ; on dit alors que v_i dépend de v_j . On peut représenter un tel modèle par un graphe de dépendances (les arcs) entre tâches (les nœuds). Le problème admet une solution si et seulement si deux tâches ne sont pas interdépendantes, i.e. si le graphe le représentant ne possède pas de cycle. Dans ce cas, le graphe est un DAG.

Cette définition extrêmement générale permet aux DAG de modéliser divers objets dans de nombreux domaines : les circuits logiques, en électronique ; les graphes PERT, en management ; les graphes de dépendance de compilateurs, tableurs, et autres réseaux de traitement de données ; en particulier, les réseaux de calcul (*Cloud Computing*, *Grid Computing*, architectures multicœurs, etc.). La Section 1 de ce projet aborde plusieurs de ces exemples.

Le reste du projet (Sections 2-4) s'intéresse plus en détail au cas des réseaux de calcul ; spécifiquement, au problème d'ordonnancement et d'exécution (éventuellement parallèle) de tâches sur réseau de calcul. Chaque nœud v_i du DAG peut donc être vu comme une tâche, dont l'exécution a un certain coût c_i , et qui ne peut être exécutée qu'après les tâches la précédant (i.e. les nœuds v_j tel qu'il existe un arc $e_{ji} = (v_j, v_i)$; cf. Figure 3). Un nœud n'ayant aucun arc entrant est appelé une source ; un nœud n'ayant aucun arc sortant est appelé un puit.

En pratique le nombre de ressources disponibles pour traiter l'ensemble des tâches est limité. Dans ce contexte, le problème posé est l'exécution de toutes les tâches du graphe, dans un ordre respectant les dépendances imposées par le graphe, et en optimisant l'utilisation des ressources disponibles de manière à minimiser le temps total d'exécution.

1 Compréhension et modélisation

1.1 Cas d'application : Codage progressif de maillages

Les maillages surfaciques représentant des objets 3D sont parfois très détaillés, et donc très lourds. Pour certaines applications, la transmission d'un maillage détaillé n'est pas souhaitable (par exemple, la bande passante est réduite, ou le terminal de visualisation est de petite taille). Une représentation progressive de maillages triangulaires (surface manifold fermée, i.e. il y a toujours deux faces autour d'une arête), a été proposée par Hugues Hoppe en 1996. Elle permet de raffiner successivement un maillage (Figure 1)

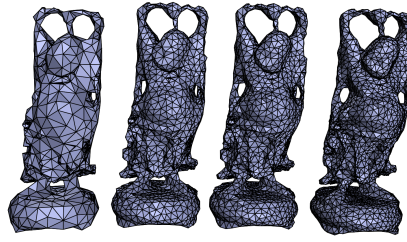


FIGURE 1 – Un maillage progressif : du maillage le plus basique (à gauche) au maillage le plus détaillé (à droite).

grâce à une opération de raffinement appelée *vertex split* (Figure 2). Pour transmettre le maillage, on transmet d'abord un maillage de base (*base mesh*) et une suite d'opération de *vertex split*. À l'inverse, la représentation simplifiée s'obtient à partir d'un maillage détaillé, en appliquant une succession de simplifications appelées *edge collapse* –opération inverse du *vertex split* (Figure 2). Un *vertex split* est

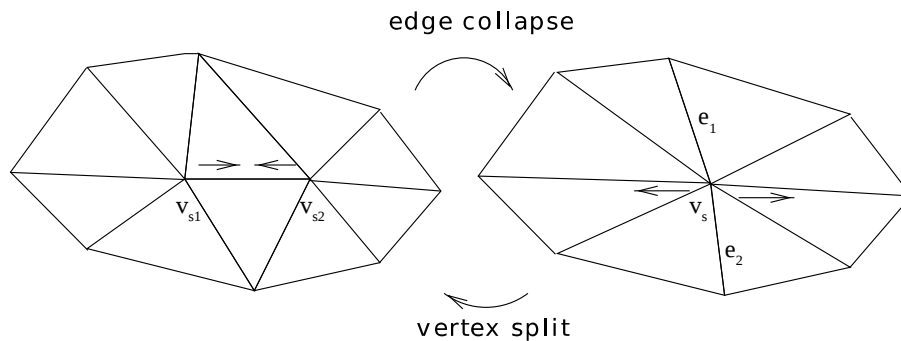


FIGURE 2 – Les deux opérations basiques inverses, *vertex split* et *edge collapse*.

défini par $(v_s, v_{s1}, v_{s2}, e_1, e_2, \mathcal{N}(v_s))$, où v_s est le sommet divisé en deux nouveaux sommets v_{s1} et v_{s2} , e_1 et e_2 les arêtes divisées, et $\mathcal{N}(v_s)$ est l'ensemble des voisins de v_s au moment du split. Un *vertex split* ne pourra être effectué que si le sommet et tous ses sommets voisins ont déjà été générés. De ce fait, les opérations de *vertex split* correspondant au raffinement d'un maillage sont dépendantes les unes des autres.

Question 1:

On considère un maillage basique que l'on veut raffiner par une série d'opérations. On s'intéresse à la modélisation du raffinement de ce maillage par un DAG. Proposez une modélisation par un DAG. En particulier, vous préciserez

- ce que représentent les nœuds du DAG ;
- comment les dépendances entre nœuds sont établies, i.e. sous quelle condition un nœud v_1 doit être traité avant un nœud v_2 .




1.2 Réseau à mémoire distribuée

Dans le reste de ce projet, on supposera par simplicité qu'on dispose d'un réseau à mémoire partagée (ex : processeur multicœurs). Dans ce type de réseau, toutes les ressources (les processeurs) partagent le même espace mémoire, où elles rangent le résultat temporaire de l'exécution d'une tâche. Elles peuvent donc toutes accéder aux données nécessaires à l'exécution des tâches suivantes.

En revanche, dans un réseau à mémoire distribuée, chaque ressource a son propre espace mémoire et n'a pas accès aux espaces des autres ressources. Dans ce cas, lorsqu'une ressource commence l'exécution d'une tâche v donnée, si elle n'a pas elle-même traité les prédécesseurs de v , le résultat de l'exécution des prédécesseurs doit lui être envoyé par les ressources les ayant exécutés. Par exemple, sur la Figure 3, si une ressource r_1 traite la tâche **a** et une ressource r_2 traite la tâche **d**, r_1 doit envoyer à r_2 le résultat produit par **a**. Ces envois (ou communications) ont un coût, qui dépend de la taille du résultat à envoyer.

Question 2:

- 
- (a) Comment pourrait-on inclure le coût des communications dans notre modélisation par DAG ?
 - (b) On appelle *mapping* la distribution des tâches aux ressources. En général, un bon mapping assure à la fois un équilibrage de charge (i.e. quantité de travail équilibrée par ressource) et un volume total de communications réduit. Cela ramène le calcul d'un mapping à un problème de partitionnement de graphes (i.e. le calcul d'une partition des nœuds en r parties, chacune attribuée à une ressource différente). Comment se traduisent les propriétés d'un bon mapping (équilibrage de charge, volume de communications réduit) en termes de graphes ?
 - (c) (**bonus**) Le calcul d'un mapping satisfaisant ces contraintes est un problème NP-complet. Néanmoins, des solutions sont connues pour certains graphes particuliers. Proposez un mapping de r ressources sur un arbre binaire de profondeur p (on supposera que r est une puissance de 2 par simplicité) et calculez le volume de communications (i.e. nombre d'envois).

Pour aller plus loin : <https://www.mcs.anl.gov/~itf/dbpp/text/node19.html>

1.3 DAGs et résilience

La résilience est la capacité d'un système ou d'une architecture réseau à continuer de fonctionner en cas de panne. Quand le nombre de ressources d'un réseau devient de plus en plus important, la probabilité qu'une panne survienne lors de l'exécution d'un graphe de tâches augmente également. Par exemple, il peut arriver qu'un processeur commette une faute lors de l'exécution d'une tâche. On suppose ici que

l'on dispose d'un moyen de détecter immédiatement une faute lorsqu'elle est commise.

Question 3:

- (a) Si un processeur commet une faute lors de l'exécution d'une tâche v_i donnée, quelle(s) tâche(s) du graphe faut-il réexécuter ? En particulier, vous prendrez en compte le fait que les résultats intermédiaires calculés par les tâches ne sont pas conservés.
- (b) Pour limiter le surcoût de la réexécution de tâches en cas de faute, une stratégie courante est de sauvegarder le résultat d'un certain ensemble \mathcal{C} de tâches. Si $\mathcal{C} = \mathcal{V}$ (i.e. toutes les tâches sont sauvegardées après leur exécution), quel est le nombre moyen de tâches réexécutées par faute (noté μ dans la suite) ? On supposera que la probabilité d'apparition d'une faute lors de l'exécution d'une tâche donnée est la même pour toutes les tâches.
- (c) Sauvegarder le résultat d'une tâche est coûteux (en temps et en mémoire), et donc en général on n'en sauvegarde qu'un certain nombre s . On peut donc se poser la question de comment choisir au mieux les tâches sauvegardées, i.e. calculer l'ensemble \mathcal{C} qui minimise le nombre moyen de tâches réexécutées par faute μ . Par exemple, l'algorithme par force brute consiste à tester tous les ensembles \mathcal{C} possibles pour déterminer le meilleur. Sa complexité est donc égale au nombre de possibilités fois le coût de calculer μ . Calculez cette complexité en fonction de $n = \#\mathcal{V}$ le nombre de tâches, $s = \#\mathcal{C}$ le nombre de tâches sauvegardées et $m = \#\mathcal{E}$ le nombre d'arcs dans le graphe.
- (d) (**bonus**) Au vu de la complexité rédhibitoire de l'algorithme par force brute, proposez une heuristique de complexité raisonnable qui vous semble appropriée.

Pour aller plus loin : voir [2] (disponible ici : <http://www.netlib.org/utk/people/JackDongarra/PAPERS/icl-utk-722-2014.pdf>).

2 Ordonnancement séquentiel

On considère dans un premier temps le cas séquentiel : si une tâche v_i est en cours d'exécution, les autres sont en attente. Elles ne peuvent commencer leur exécution qu'après la fin de v_i . Par simplicité, on suppose qu'une tâche ne peut pas être interrompue en milieu d'exécution. On suppose également que chaque tâche a un coût uniforme $c_i = 1$, donc qu'elle est intégralement traitée par une ressource en une étape. Le cas séquentiel survient par exemple lorsqu'on ne dispose que d'une seule ressource de calcul (ex : un seul processeur).

Question 4:

Que peut-on immédiatement conclure sur le temps total d'exécution ? En déduire que le problème se ramène à uniquement respecter les dépendances du graphe.

2.1 Tri topologique

Un DAG, à travers l'expression de ses dépendances, définit automatiquement un ordre *partiel* sur les tâches : en effet, $(v_i, v_j) \in \mathcal{E} \Rightarrow v_i \preceq v_j$.

Pour rappel, un ordre partiel sur \mathcal{V} possède les mêmes propriétés qu'un ordre total (réflexivité, antisymétrie et transitivité), mais en revanche il n'existe pas nécessairement de relation d'ordre entre tout couple d'éléments, i.e. $(v_i \preceq v_j \text{ ou } v_j \preceq v_i)$ n'est pas toujours vrai.

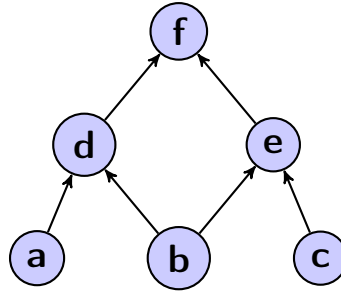


FIGURE 3 – La tâche **f** dépend des tâches **d** et **e**. La tâche **d** dépend des tâches **a** et **b**; la tâche **e** dépend de **b** et **c**. Les tâches **a**, **b** et **c** sont sans dépendance : ce sont des sources. La tâche **f** est un puit.

Par exemple, sur la Figure 3, les nœuds **a**, **b**, **c** ne sont pas ordonnés entre eux, tout comme **d**, **e**, ainsi que **a**, **e**, ou encore **c**, **d**.

Un algorithme de tri topologique consiste à trouver un ordre topologique sur \mathcal{V} , i.e. numéroté les nœuds de manière à établir entre eux un ordre *total* qui respecte l'ordre partiel du DAG. Un tel algorithme peut prendre la forme suivante :

```

1  Y = SansDep( $\mathcal{V}$ )
2  Z = []
3  tant que Y  $\neq$  [] faire
4     $v_i$  = Retirer(Y).
5    Numéroté( $v_i$ ).
6    Ajouter( $v_i$ , Z).
7     $\forall v_j \in \text{Succ}(v_i)$  :
8      si  $\text{Prec}(v_j) \subset Z$  alors
9        Ajouter( $v_j$ , Y).
10   fin
11 fin
  
```

A tout moment de l'algorithme, chaque nœud du graphe est dans un des trois possibles états :

- non numéroté et avec certains de ses prédécesseurs non numérotés ;
- non numéroté et avec tous ses prédécesseurs numérotés ;
- numéroté.

Question 5:

- (a) Identifiez ces états avec les listes Y , Z et $X = \mathcal{V} \setminus (Y \cup Z)$. Quelle propriété garantit que l'ordre produit par cet algorithme est topologique, c'est-à-dire qu'il respecte l'ordre partiel défini par les dépendances du graphe ? Quelle propriété garantit qu'il est total ?
- (b) En supposant que les fonctions Succ et Prec ont un coût de c , calculer l'ordre de complexité de l'algorithme en fonction de c , du nombre de nœuds n et du nombre d'arcs m .

L'ordre topologique produit par cet algorithme dépend du format de liste utilisé pour Y . Par exemple, sur la Figure 3, il existe 16 ordres topologiques possibles. En particulier, les ordres **a,b,c,d,e,f** et **a,b,d,c,e,f**

sont respectivement obtenus en utilisant un format de file, et un format de pile, pour Y .



Question 6:

Comment appelle-t-on les parcours induits par l'utilisation d'une pile ? Et d'une file ?



Question 7:

Programmez un algorithme de tri topologique utilisant un format de file pour Y .

3 Ordonnancement parallèle

On considère désormais le cas parallèle : on suppose que l'on peut traiter en même temps r tâches indépendantes, où r est le nombre de ressources de calcul (ex : processeurs) dont on dispose. On continue de supposer que chaque tâche a un coût uniforme $c_i = 1$.

A chaque étape, on va donc avoir au plus r tâches exécutées, que l'on stocke dans une liste. On représente la trace d'exécution par la liste de chacune des étapes, i.e. une liste de listes.

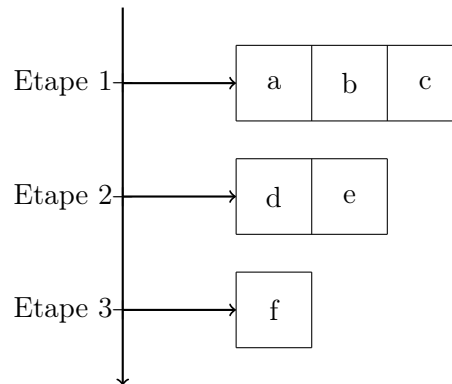


FIGURE 4 – Trace d'exécution des tâches du graphe de la Figure 3, en ayant un nombre de ressources au moins égal à 3, et en respectant l'ordre partiel du graphe. A chaque étape correspond une liste de tâches à exécuter, d'où la représentation en liste de listes.



Question 8:

- Donnez une borne inférieure du temps total d'exécution en fonction du nombre de nœuds n et du nombre de ressources r . Dans le cas où cette borne inférieure est atteinte, que peut-on dire sur l'utilisation des ressources à chaque étape ?
- Comment se traduit la contrainte de ressources limitées sur les listes à chaque étape ?



Question 9:

Programmez un algorithme produisant une trace d'exécution à partir d'un DAG, pour un nombre de ressources donné r .

**Question 10:**

Analysez l'efficacité de votre algorithme sur différents graphes, en faisant varier le nombre de ressources r .

Lorsque le nombre de tâches prêtes à être traitées est supérieur au nombre de ressources disponibles, il faut choisir un ordre de priorité sur les tâches (i.e., choisir quelles tâches sont traitées immédiatement et quelles tâches sont reportées pour plus tard).

Trouver un ordonnancement optimal est un problème NP-complet. En général, les stratégies d'ordonnancement ont donc recours à une heuristique.

On appelle chemin critique le chemin le plus long reliant une source à un puit.

**Question 11:**

Proposez une heuristique pour choisir les tâches traitées en priorité (i.e., les r premières tâches de Y).

**Question 12:**

Implémentez votre heuristique.

**Question 13:**

Analysez l'efficacité de l'heuristique sur différents graphes, en faisant varier le nombre de ressources r .

4 Ordonnancement parallèle sous contrainte

Dans la section précédente, le problème posé était simplement de minimiser le temps d'exécution d'un graphe de tâches. Dans la vie réelle, il est fréquent de devoir prendre en compte d'autres considérations, i.e. calculer un ordonnancement sous contrainte. Par exemple, on peut vouloir limiter la consommation d'énergie ou de mémoire par les ressources. Dans la suite, nous prenons l'exemple de la consommation mémoire.

On associe à chaque tâche v_i une quantité de mémoire m_i donnée, taille de l'espace temporaire à allouer pour l'exécution de la tâche. On suppose que l'on dispose d'une mémoire totale M . On définit la *mémoire active* m_j^{act} à l'étape j comme la somme des coûts mémoire des tâches exécutées à cette étape. Le problème est donc désormais de calculer un ordonnancement parallèle sous la contrainte $(C) : \forall j, m_j^{act} \leq M$.

**Question 14:**

Dans le cas d'une exécution séquentielle ($r = 1$), en quelle contrainte simplifiée la contrainte (C) se convertit-elle ?

**Question 15:**

Adaptez le programme codé en Section 3 au cas sous contrainte. Analysez les résultats obtenus sur différents graphes, en faisant varier r et M .

Vous devriez avoir remarqué que le temps d'exécution de l'ordonnancement sous contrainte est plus long,

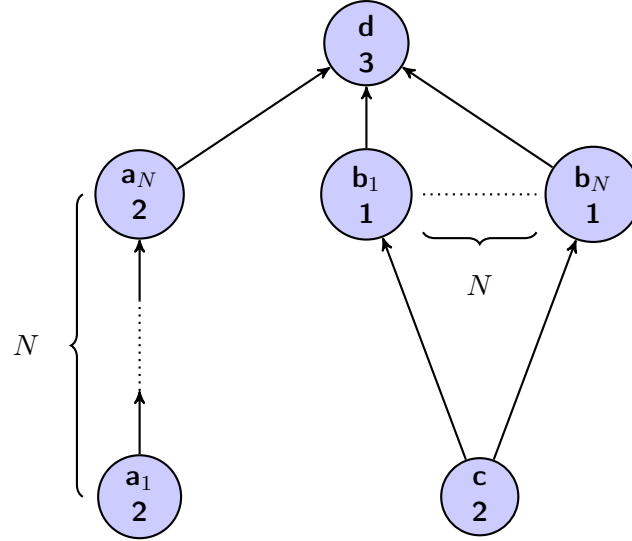


FIGURE 5 – Graphe de la question 16(b). Les chiffres sous les noms des nœuds indiquent la valeur des m_i .

augmentation qui traduit la *séquentialisation* forcée de certaines parties du graphe. Plus intéressant, il se peut qu'un ordonnancement optimal (calculé par votre heuristique en Section 3) ne le soit plus dans le cas sous contrainte (i.e. qu'il existe un autre ordonnancement qui atteint un meilleur temps d'exécution tout en respectant la contrainte mémoire).

Question 16:

- Donner un exemple simple (moins de 10 nœuds) de graphe avec $r = 3$ où la priorité $P_1 = P(r = 3, M = \infty)$ calculée par votre heuristique est clairement non-optimale (i.e. proposez une autre priorité $P_2 = P(r = 3, M = cste)$ telle que $T(P_1, r = 3, M = cste) > T(P_2, r = 3, M = cste)$, où $T(S, r, M)$ est le temps d'exécution avec l'ordonnancement S , r ressources et M mémoire.
- Calculez $T(P_1, r = 2, M = 3)$ (où $P_1 = P(r = 2, M = \infty)$) en fonction de N sur le graphe de la Figure 5. Proposez une priorité P_2 tel que $T(P_2, r = 2, M = 3) < T(P_1, r = 2, M = 3)$.
- (bonus)** Inspirez-vous de cet exemple pour proposer une heuristique modifiée plus adaptée au cas sous contrainte.

Question 17:

- (bonus)** Modifiez l'heuristique et analysez les résultats obtenus sur différents graphes.

5 Evaluation du projet

Le projet sera réalisé en binôme.

Vous devez rendre un code commenté, auquel vous ajouterez les tests de vos fonctions. Vous remplirez vos réponses aux questions sur Moodle et vous rendrez une archive nommée “**Nom1_Nom2.tgz**”.

Barème (approximatif) :

- Code + commentaires + réponses aux questions (16 pts)
 - Section 1 : 4pts
 - Section 2 : 4pts
 - Section 3 : 4pts
 - Section 4 : 4pts
- Tests (4 pts) : une séance de tests en salle de TP aura lieu à une date à définir. Les interfaces des fonctions demandées sont fournies sous Moodle (fichier “interface.mli”).

Echéance : L’archive contenant votre code devra être envoyée à votre enseignant de TP avant le **12 juin à minuit**.

Pour aller plus loin :

Références

- [1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK : A portable linear algebra library for high-performance computers*. SIAM, Philadelphia, PA, 1992.
- [2] C. Cao, T. Herault, G. Bosilca, and J. Dongarra. Design for a soft error resilient dynamic task-based runtime. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 765–774, May 2015.
- [3] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4) :406–471, December 1999.