Exercices
Cours de Master « optimisations de code ».

Alain Darte

1 Pipeline logiciel

On considère un graphe de dépendances $G = (V, E, d, w)$ où V est l'ensemble des instructions et E l'ensemble des dépendances. Pour chaque arête $e \in E$, $d(e) \in \mathbb{N}^*$ indique un délai et $w(e) \in \mathbb{Z}$ une distance de dépendance. Chaque sommet $u \in V$ représente un ensemble d'opérations (u, i) pour tout i dans un intervalle $\mathbb{I}$ de $\mathbb{Z}$. Une arête $e = (u, v)$ signifie que l'opération (u, i) doit être terminée au plus tard $d(e)$ unités de temps avant que l'opération $(v, i + w(e))$ ne commence. Ainsi, $\sigma : V \times \mathbb{I} \mapsto \mathbb{N}$ est un ordonnancement du graphe si $\sigma(v, i + w(e)) \geq \sigma(u, i) + d(e)$ pour tout $i \in \mathbb{I}$ et toute arête $e = (u, v)$. On suppose que chaque opération (u, i) nécessite une ressource de calcul pendant $\delta(u)$ unités de temps et que, pour toute arête $e = (u, v)$, $d(e) = \delta(u)$.

1.1 Rappels de cours

Question 1

Montrer que si $w(C) > 0$ pour tout circuit C de G, il existe une fonction de retiming $q : V \mapsto \mathbb{Z}$ telle que, pour toute arête $e = (u, v) \in E$, $q(v) - q(u) + w(e) \geq 0$. □

Question 2

Montrer qu'il existe un ordonnancement sans contraintes de ressources (c'est-à-dire avec autant de ressources que nécessaire), pour tout intervalle $\mathbb{I}$ borné, si et seulement si, dans chaque composante fortement connexe de G, les poids $w(C)$ des circuits C sont soit tous strictement positifs soit tous strictement négatifs. Indiquer un procédé constructif d'un tel ordonnancement lorsqu'il existe. □

Dans la suite, on considère $\mathbb{I} = \mathbb{N}$. On dit qu'un ordonnancement σ est cyclique s'il est de la forme $\sigma(u, i) = \lambda.i + \rho(u)$ où $\lambda \in \mathbb{N}^*$ et $\rho(u) \in \mathbb{Z}$ pour tout $u \in V$.

Question 3

Montrer qu'il existe un ordonnancement cyclique pour p ressources de calcul identiques si et seulement si $w(C) > 0$ pour tout circuit C de G. □

Question 4

Rappeler pourquoi, si σ est un ordonnancement cyclique, alors $\lambda \geq d(C)/w(C)$ pour tout circuit C et $p.\lambda \geq \sum_{u \in V} \delta(u)$ pour p ressources de calcul identiques. □

1.2 Cas d'un seul circuit

On fait ici deux hypothèses :

1. pour tout $u \in V$, il existe une arête $e = (u, u)$ telle que $w(e) = 1$: toutes les opérations correspondant à un même sommet doivent donc être effectuées séquentiellement ;
2. le reste des dépendances forme un unique circuit C tel que $w(C) > 0$.

On illustrera les différentes questions sur le graphe à 5 sommets A, B, C, D et E, de durées $d(A) = 3$, $d(B) = d(D) = d(E) = 2$, $d(C) = 5$, avec les distances $w(A \to B) = w(B \to C) = w(D \to E) = 0$, $w(C \to D) = 2$, $w(E \to A) = 1$.

Question 5

Montrer que si on déroule complètement le circuit C, c'est-à-dire si on considère le graphe des opérations (u, i), on obtient $w(C)$ chemins de dépendance disjoints (si on ignore les arêtes de (u, i) vers $(u, i + 1)$). En déduire, lorsque $p \geq w(C)$, l'existence d'un ordonnancement cyclique $\sigma(u, i) = \lambda i + \rho(u)$ de période λ optimale. Comment calculer un tel ordonnancement de façon pratique ? Illustrer sur l'exemple. □

Question 6

Soit e_0 une dépendance du circuit telle que $w(e_0) > 0$. On définit $G' = (V, E, d, w')$ le graphe obtenu à partir de G avec $w'(e) = w(e)$ si $e \neq e_0$ et $w'(e_0) = w(e_0) - 1$. Montrer que tout ordonnancement valide pour G' est valide pour G. En déduire une méthode de construction d'un ordonnancement cyclique optimal pour G même lorsque $p < w(C)$. Illustrer sur l'exemple. □

1.3 Ordonnancement avec allocation de permutation : FACULTATIF

On revient aux hypothèses générales sur G, toujours avec p ressources identiques. Une fonction d'allocation est une fonction de $V \times \mathbb{N}$ dans $[1..p]$ qui indique la ressource calculant l'opération (u, i). Une fonction d'allocation r est valide pour un ordonnancement σ si, à tout instant, chaque ressource ne calcule qu'au plus une opération, c'est-à-dire que $r(u, i) = r(v, j)$ implique $\sigma(u, i) + \delta(u) \leq \sigma(v, j)$ ou $\sigma(v, j) + \delta(v) \leq \sigma(u, i)$. On dit que r est une allocation de permutation s'il existe une permutation π de $[1..p]$ telle que $r(u, i + 1) = \pi(r(u, i))$ pour tout $i \in \mathbb{N}$ et $u \in V$. L'allocation est alors entièrement spécifiée par la donnée d'une allocation initiale $r(u, 0) = r_u \in [1..p]$.

Question 7

Soit un ordonnancement cyclique σ de période λ et une allocation r pour p ressources identiques. Montrer qu'il existe une allocation de permutation r' valide pour l'ordonnancement σ. Penser à examiner un motif de longueur λ et à réaffecter les opérations en examinant celles à cheval sur plusieurs motifs. □

On considère une allocation de permutation r valide pour un ordonnancement cyclique σ. Si (v, j) suit directement (u, i) dans l'utilisation d'une même ressource, on rajoute au graphe G une arête $e = (u, v)$ telle que $w(e) = j - i$ et $d(e) = \delta(u)$. On note A l'ensemble de ces arêtes.

Question 8

Montrer que les arêtes ainsi rajoutées (celles de A donc) forment q circuits disjoints $C_1, \ldots, C_q$ tels que $w(C_1) + \cdots + w(C_q) \leq p$ et que le graphe complet (formé par les arêtes de E et de A) n'a que des circuits C tels que $w(C) > 0$. Réciproquement, si un tel ensemble d'arêtes A existe, montrer qu'on peut en déduire un ordonnancement et une allocation de permutation pour $w(C_1)+\cdots+w(C_q)$ ressources identiques. □

L'objet des dernières questions est de montrer que s'il existe un ordonnancement et une allocation de permutation pour un graphe G et p ressources identiques, alors il existe un ordonnancement et une allocation circulaire, c'est-à-dire une allocation définie par une permutation ne comportant qu'un cycle de longueur p. Pour ce faire, on va montrer qu'à partir d'une allocation de permutation comportant q circuits disjoints formés par les arêtes dans A, on peut construire une allocation ne comportant que $q-1$ tels circuits. Pour chaque sommet $u \in V$, on note C_u le circuit d'arêtes de A auquel il appartient et $\gamma(u)$ son prédécesseur dans ce circuit.

Pour $u \in V$ et $v \in V$ tels que $C_u \neq C_v$, on définit l'opération **fusion**(G, u, v, w) de la façon suivante : on remplace les arêtes $e_1 = (\gamma(u), u)$ et $e_2 = (\gamma(v), v)$ par les arêtes $e_3 = (\gamma(u), v)$ et $e_4 = (\gamma(v), u)$ avec $d(e_3) = \delta(\gamma(u))$, $d(e_4) = \delta(\gamma(v))$, $w(e_3) = w(e_1) + w$ et $w(e_4) = w(e_2) - w$.

Question 9

Montrer que si $C_u \neq C_v$ et $\rho_v - \rho_{\gamma(u)} - \delta(\gamma(u)) = n.\lambda$ avec $n \in \mathbb{Z}$ (c'est-à-dire, une occurrence de v commence juste à la fin d'une occurrence de $\gamma(u)$), alors il existe un ordonnancement pour le graphe complet (c'est-à-dire les arêtes de E et A) après l'opération **fusion**$(G, u, v, -n-w(e_1))$ avec $e_1 = (\gamma(u), u)$. □

On définit à présent l'opération **décale**(σ, C, t), pour un ordonnancement σ, un circuit C d'arêtes de A et $t \in \mathbb{N}$, qui consiste à retarder de t unités de temps toute opération u telle que $C_u = C$. Pour $u \in V$ et $v \in V$, on définit $\Delta_{u,v} = (\rho_v - \rho_u - \delta(u)) \bmod \lambda$ et $\Delta^* = \min\{\Delta_{u,v} \mid C_u \neq C_v\}$.

Question 10

Montrer que, quel que soit le circuit C d'arêtes de A, l'ordonnancement obtenu par l'opération **décale**(σ, C, Δ^*) est valide et qu'il existe deux sommets u et v vérifiant les hypothèses de la question précédente. Conclure. □

1.4 Calcul simplifié de λ_∞ pour durées unitaires : FACULTATIF

On note λ_∞ le temps de cycle optimal sans contraintes de ressources :

$$\lambda_\infty = \max\{d(C)/w(C) \mid C \text{ circuit de } G\}$$

Question 11

Rappeler l'algorithme du cours pour calculer $\lceil \lambda_\infty \rceil$, ainsi que sa complexité. □

On suppose que $d(u) = 1$ pour tout $u \in V$, ce qui va permettre de calculer λ_∞ plus rapidement. Pour un circuit $C = (e_1, \ldots, e_k)$, on note $\mu(C) = \frac{1}{k}\sum_{i=1}^{k} w(e_i)$ et μ^* l'inverse de λ_∞, c'est-à-dire $\mu^* = \min\{\mu(C) \mid C \text{ circuit de } G\}$.

Pour chaque sommet v, on note $\delta(v)$ le poids (somme des $w(e)$ de ses arcs constituants) minimal (éventuellement $-\infty$) d'un chemin se terminant en v et $\delta_k(v)$ le poids minimal d'un chemin ayant exactement k arcs et se terminant en v ($\delta_k(v) = +\infty$ s'il n'y a pas de tel chemin). On note $n = |V|$.

Remarque : dans les questions suivantes, les poids $w(e)$ peuvent être quelconques, ceci pour pouvoir appliquer les résultats des deux premières questions à la question 14.

Question 12

Si $\mu^* \geq 0$, montrer que pour tout $v \in V$, $\max_{0 \leq k < n}(\delta_n(v) - \delta_k(v)) = \delta_n(v) - \delta(v) \geq 0$. □

Question 13

Si $\mu^* = 0$, montrer qu'il existe $v \in V$ tel que $\delta_n(v) = \delta(v)$. En déduire :

$$\mu^* = 0 = \min_{v \in V} \max_{0 \leq k < n} \frac{\delta_n(v) - \delta_k(v)}{n - k}$$

□

Question 14

Que se passe-t-il si on ajoute une même valeur w au poids $w(e)$ de chaque arc e du graphe ? En déduire la relation :

$$\mu^* = \min_{v \in V} \max_{0 \leq k < n} \frac{\delta_n(v) - \delta_k(v)}{n - k}$$

□

Question 15

Donner un algorithme de complexité $O(|V||E|)$ pour calculer μ^* et donc λ_∞. □

2 Ordonnancement et barrières de synchronisation

On s'intéresse au problème du placement de barrières de synchronisation dans un programme « multi-threadé » de type SPMD (Single Program Multiple Data), comme il se pose par exemple, sous diverses variantes, en HPF, OpenMP, UPC, Cray Fortran, etc. Pour illustrer ce problème, considérons par exemple le cas d'un langage de programmation parallèle qui serait basé sur les concepts suivants : chaque processeur (ou « thread ») a le même programme, travaillant soit sur des variables privées, soit sur certaines variables partagées spécifiées et analysables par le compilateur. Certaines instructions en revanche peuvent ne pas être exécutées, selon le numéro de processeur.

Premier cas, la sémantique du programme est imposée dans le code par des barrières de synchronisation explicites. Si tous les threads se synchronisent toujours sur les mêmes barrières (« structural correctness »), le compilateur peut analyser le code et calculer, pour chaque barrière, quelles sont les dépendances écriture/lecture des variables partagées résultant de cette barrière et se servir de ces informations pour déplacer et supprimer des barrières tout en conservant la sémantique du code. Par exemple, dans la séquence suivante (ou seules a et b sont des variables partagées) :

S : if (...) $a = \ldots$;
barrier ;
T : $T_1 = a$;
U : if (...) $b = \ldots$;
barrier ;
V : $T_2 = b$;

on peut reconnaître que la première (resp. deuxième) barrière impose un ordre strict entre S et T (resp. U et V) uniquement du fait de la variable a (resp. b). On peut alors ré-écrire le code en :

S : if (...) $a = \ldots$;
U : if (...) $b = \ldots$;
barrier ;
T : $T_1 = a$;
V : $T_2 = b$;

par un ré-ordonnancement des instructions et ainsi supprimer une barrière. En revanche, si le calcul de b dépendait de T_1, cette optimisation ne serait bien entendu pas possible.

Second cas, la sémantique du programme parallèle est donnée par le fait que, grâce à des directives, l'utilisateur précise, pour chaque variable partagée, quel processeur est responsable de ses modifications (c'est par exemple le cas en HPF pour les données distribuées et l'« owner computes rule », c'est-à-dire seul celui qui « possède » la donnée effectue l'opération d'écriture). Il n'y a alors pas de barrières de synchronisation explicites, mais des barrières implicites que le compilateur doit analyser. Après cette analyse, le compilateur doit placer lui-même des barrières de synchronisation dans l'implémentation effective, et le problème du ré-ordonnancement des instructions du programme pour minimiser le nombre de barrières est alors similaire au cas précédent.

Dans cet exercice, on ne s'intéresse ni au problème d'analyse du code, ni au problème de modélisation, mais uniquement au ré-ordonnancement des instructions, une fois cette analyse effectuée. On suppose que le problème est modélisé par un graphe dirigé, avec des arcs « mous » et des arcs « durs ». Chaque sommet correspond à une tâche (instruction, bloc d'instructions, appel de fonction, etc.) dont la durée est sans importance. Un arc mou entre deux tâches u et v signifie que u doit

être effectuée avant v mais nul besoin de barrière, c'est la séquentialité au sein de chaque processeur qui garantit la sémantique du code (comme dans l'exemple si le calcul de b dépend de T_1, il y aurait un arc mou de T vers U). Un arc dur entre deux tâches u et v signifie que u doit être effectuée avant v et qu'une barrière de synchronisation doit être placée quelque part entre l'exécution de u et celle de v (dans l'exemple, il y aurait un arc dur entre S et T et un arc dur entre U et V).

Question 1

Lorsque le code n'a pas de boucles (« straight-line code »), le graphe est sans circuit. Donner alors un algorithme permettant de trouver l'ordre dans lequel écrire les instructions et où placer les barrières pour conserver la sémantique du code et minimiser le nombre de barrières nécessaires. Quel est la complexité de votre algorithme ? □

On s'intéresse à présent au cas du corps d'une boucle interne dans lequel on souhaite permuter l'ordre des instructions pour minimiser le nombre de barrières. Le graphe de dépendances contient à présent plus d'information : en plus de la nature d'un arc (mou ou dur), un arc est soit intra-itération (c'est-à-dire dans le corps de la boucle), soit inter-itération (c'est-à-dire d'une instruction vers une autre à une itération ultérieure). Le graphe peut comporter à présent des circuits mais le sous-graphe des arcs intra-itération est bien sûr sans circuit (dépendances du corps de boucle).

Si D est le nombre minimal de barrières nécessaires en ne considérant que les dépendances au sein du corps de la boucles, on peut appliquer l'algorithme de la question 1 pour ré-ordonnancer les instructions, puis on rajoute une barrière en fin de corps de boucle pour satisfaire tout arc inter-itération. L'optimal est donc au moins D et au plus $D + 1$. Par exemple, pour 3 instructions S, T et U, un arc dur intra-itération de S vers T et un arc dur inter-itération entre T et U, on peut écrire les instructions avec 2 barrières dans le corps de la boucle (code ci-dessous à gauche). Mais on pourrait aussi les écrire avec une seule barrière, la dépendance de T vers U à l'itération suivante étant traitée par la première barrière de l'itération suivante (code de droite).

```
Loop :
    S ; U ;
    barrrier ;
    T ;
    barrier ;
EndLoop
```

```
Loop :
    S ;
    barrrier ;
    U ; T ;
EndLoop
```

Question 2

Comment déterminer si D barrières sont suffisantes ? Montrer que le problème est NP-complet ou bien donner un algorithme polynomial de complexité la plus faible possible. □

Question 3

Quelles transformations de code plus complexes qu'une simple permutation des instructions du corps de boucle pourrait-on imaginer pour réduire encore plus le nombre de barrières, statiquement et/ou dynamiquement ? Proposez des algorithmes. □

3 Itérations parallèles dans les boucles et retiming

On considère une boucle simple (sans boucle interne), représentée, comme pour le pipeline logiciel, par un graphe $G = (V, E, w)$, où $w(e) \geq 0$ si $e \in E$ et $w(C) > 0$ si C est un circuit. Ici, on ne s'intéresse pas à la durée des instructions (pas de fonction d) et on cherche à affecter chaque itération à un "thread" tout en maximisant le nombre de threads pouvant être exécutés en parallèle.

Question 1

Montrer qu'on peut exécuter $w^{\min} = \min\{w(e) \mid e \in E, w(e) > 0\}$ threads en parallèle. □

On va chercher à maximiser $w^{\min}$ par retiming. On note G_r le graphe obtenu après retiming r, c'est-à-dire $G_r = (V, E, w_r)$ avec $w_r(e) = w(e) + r(v) - r(u)$. On rappelle qu'une fonction r de V dans $\mathbb{Z}$ n'est un retiming que si $w_r(e) \geq 0$ pour tout $e \in E$.

Question 2

Rappeler pourquoi un retiming ne change pas le poids $w(C)$ d'un circuit C. □

Question 3

Si G est un circuit simple, montrer comment trouver un retiming r tel que $w_r^{\min}$, c'est-à-dire $\min\{w_r(e) \mid e \in E, w(e) > 0\}$, soit maximal. □

Question 4

Même question si G n'a que des circuits simples, c'est-à-dire si chaque composante fortement connexe de G ne comporte qu'un circuit. □

On va traiter le cas de G quelconque, cyclique, par programmation linéaire en nombres entiers.

Question 5

Pour une composante fortement connexe C de G, on note $w(C)$ la somme des poids des arcs qui la constituent et K le maximum de ces $w(C)$. Montrer que $w_r^{\min} \leq K$ pour tout retiming r de G. □

Question 6

Soit deux contraintes affines $\phi(\vec{x}) \leq 0$ et $\psi(\vec{x}) \leq 0$. Montrer que si on peut se restreindre à des solutions $\vec{x}$ telles que $\phi(\vec{x}) \leq K_\phi$ et $\psi(\vec{x}) \leq K_\psi$ alors la contrainte $\{(\phi(\vec{x}) \leq 0)$ **ou** $(\psi(\vec{x}) \leq 0)\}$ est équivalente aux contraintes :

$$\begin{cases} 0 \leq \epsilon \leq 1 \text{ et } \epsilon \text{ entier} \\ \phi(\vec{x}) \leq \epsilon K_\phi \\ \psi(\vec{x}) \leq (1 - \epsilon) K_\psi \end{cases}$$

□

Question 7

En déduire une méthode utilisant de la programmation linéaire en nombres entiers pour trouver un retiming r maximisant $w_r^{\min}$, pour un graphe G quelconque possédant au moins un cycle. □

4 Propriétés des graphes de flot de contrôle

On rappelle qu'un graphe de flot de contrôle est un graphe orienté $G = (V, E)$ possédant une racine $r \in V$, c'est-à-dire tel que pour tout sommet $u \neq r$, il existe un chemin de r à u. On appelle « back-edge » toute arête (u, v) telle que v domine u. Pour un parcours en profondeur (« dfs »), on appelle « dfs-back-edge » toute arête (u, v) telle que u est visité dans un parcours en profondeur issu de v. On dit qu'un graphe $G = (V, E)$ est réductible si et seulement si E peut être partitionné en $E = E_1 \cup E_2$, $E_1 \cap E_2 = \emptyset$, tels que :

- le graphe (V, E_1) est acyclique et tous ses sommets y sont atteignables depuis r.
- pour toute arête $(u, v) \in E_2$, v domine u, c'est-à-dire (u, v) est une « back-edge ».

On va montrer quelques caractérisations équivalentes des graphes réductibles.

Question 1

Montrer que si G est réductible, une telle partition en E_1 et E_2 est unique. □

Question 2

Montrer qu'un graphe de flot de contrôle est réductible si et seulement si le graphe obtenu en supprimant toutes les « back edges » est acylique. □

Question 3

Montrer qu'un graphe de flot de contrôle est réductible si et seulement si toute « dfs-back-edge » est une « back-edge », quel que soit le « dfs ». □

Question 4

Montrer qu'un graphe de flot de contrôle est réductible si et seulement si, pour toute « loop forest » au sens de Ramalingam, chaque boucle n'a qu'une seule entrée. Montrer que, dans ce cas, il n'existe qu'une seule « loop forest ». □

Question 5

Montrer qu'un graphe de flot de contrôle est réductible si et seulement si l'application répétée des transformations T1 et T2 conduit à un graphe à un sommet, où :

- T1 supprime une « self-loop », c'est-à-dire une arête (u, u).
- T2 fusionne deux sommets u et v si u est le seul prédécesseur de v. □

On va terminer par une propriété, sans rapport avec la réductibilité, énoncée en cours pour la construction SSA, mais non démontrée. Pour $S \subseteq V$, sous-ensemble de sommets de G, on note $J(S)$ l'ensemble des « join points » de S, c'est-à-dire l'ensemble des $w \in V$ tels qu'il existe deux chemins disjoints (hormis en w) de u à w et de v à w, avec $u \in S$ et $v \in S$. On note $J^1(S) = J(S)$, $J^k(S) = J(S \cup J^{k-1}(S))$ et $J^+(S)$ (« iterated join set ») la limite de la suite $J^k(S)$.

Question 6

Montrer que $J(S) = J^+(S)$. □