Concurrence et allocation de ressources locales instantanément stabilisante

Karine Altisen, Stéphane Devismes et Anaïs Durand

VERIMAG UMR 5104, Université Grenoble Alpes, France

Cet article est un résumé de [ADD15] où nous étudions la notion de concurrence dans les problèmes d'allocation de ressources. Nous proposons des propriétés générales permettant d'exprimer la qualité de concurrence d'une solution à un problème d'allocation de ressources et établissons quelle qualité de concurrence peut être atteinte par un algorithme résolvant le problème d'allocation de ressources locales. Enfin, nous proposons un algorithme d'allocation de ressources locales instantanément stabilisant qui réalise cette qualité de concurrence.

Mots-clefs : Algorithmes distribués, allocation de ressources locales, concurrence, stabilisation instantanée

1 Introduction

L'*exclusion mutuelle* est un problème d'allocation de ressources fondamental dans lequel il faut gérer un accès équitable à une unique ressource non partageable. Ce problème est intrinsèquement séquentiel car deux processus ne peuvent accéder simultanément à cette ressource. Cependant, il existe de nombreux autres problèmes d'allocation de ressources où plusieurs ressources sont partagées et peuvent être utilisées de façon concurrente. Par exemple, le problème de ℓ-*exclusion* est une généralisation de l'exclusion mutuelle, où non pas une seule mais ℓ copies d'une ressource peuvent être utilisées simultanément.

Pour résoudre efficacement ces problèmes, un algorithme doit permettre le plus possible d'accès simultanés aux différentes ressources. Il est donc important d'exprimer la qualité de la concurrence à l'aide d'une propriété dédiée et d'évaluer les algorithmes d'allocation de ressources selon cette propriété. Cependant, la majorité des solutions aux problèmes d'allocation de ressources ne considèrent pas la question de la concurrence, bien que, comme Fischer *et al* le soulignent dans [FLBB79], ne pas le faire puisse mener à des solution dégénérées. Par exemple, un algorithme d'exclusion mutuelle répond, certes inefficacement, à la spécification de la ℓ-exclusion sans la propriété de concurrence.

Ainsi, Fischer *et al* [FLBB79] proposent une propriété *ad hoc* qui traduit la concurrence dans le problème de ℓ-exclusion. Cette propriété, nommée *évitement de ℓ-interblocage*, peut s'exprimer informellement comme suit : si moins de ℓ processus sont en train d'exécuter leur section critique [†] et s'il y a au moins un processus demandeur, alors un processus demandeur finira par obtenir sa section critique, même si aucun autre processus ne quitte sa section critique entre-temps. D'autres propriétés de concurrence ont été proposées pour d'autres problèmes d'allocation de ressources, comme la k-parmi-ℓ exclusion [DHV03] et la coordination de comités [BDP11]. Cependant, toutes ces propriétés sont dédiées à un problème particulier.

Nous proposons ici une généralisation de la propriété d'évitement de ℓ-interblocage à tout problème d'allocation de ressources. Cette propriété est nommée *concurrence maximale*. Il existe des problèmes, comme la ℓ-exclusion [FLBB79], pour lesquels la concurrence maximale peut être satisfaite, alors que pour d'autres problèmes d'allocation de ressources, c'est impossible. Par exemple, Datta *et al* [DHV03] ont prouvé qu'il est impossible de concevoir un algorithme satisfaisant la (k,ℓ)-*équité stricte* qui est l'instance de la concurrence maximale dédiée au problème de la k-parmi-ℓ exclusion. Nous proposons en conséquence une généralisation de la concurrence maximale, nommée *concurrence partielle*, permettant d'exprimer des qualités de concurrence plus faibles.

Nous nous intéressons ensuite à la concurrence dans une classe de problèmes d'allocation de ressources particulière : l'*allocation de ressources locales*. L'allocation de ressources locales [CDP03] est définie grâce

†. La *section critique* est la portion de code exécutée pour accéder aux ressources allouées au processus.

à la notion de *compatibilité* entre ressources : deux ressources x et y sont dites *compatibles* si deux processus voisins peuvent les utiliser en même temps. Sinon, x et y sont dites *incompatibles*. Le problème d'allocation de ressources locales consiste à assurer que tout processus demandant l'accès à une ressource x l'obtient en temps fini (vivacité) et ce alors qu'aucun de ses voisins n'utilise une ressource incompatible avec x (sûreté). De nombreux problèmes classiques sont des instances de l'allocation de ressources locales, comme le dîner de philosophes ou l'exclusion mutuelle locale.

La spécification initiale de l'allocation de ressources locales [CDP03] ne considérant pas la question de la concurrence, nous étudions la qualité de concurrence qu'il est possible de réaliser pour ce problème. Nous obtenons deux résultats. Premièrement, nous montrons qu'il est impossible de concevoir un algorithme d'allocation de ressources locales qui satisfasse la concurrence maximale. Cependant, nous exprimons la meilleure qualité de concurrence qu'il est possible d'obtenir *via* la propriété de *concurrence forte*, instance particulière de la concurrence partielle, et proposons un algorithme *instantanément stabilisant* vérifiant cette dernière propriété.

Après un nombre fini de fautes transitoires (corruption mémoire, perte de message, *etc.*), un algorithme instantanément stabilisant retrouve immédiatement un comportement correct. Un algorithme instantanément stabilisant n'est bien sûr pas insensible aux pannes transitoires. En fait, il garantit uniquement que toute tâche démarrée après la dernière panne sera exécutée correctement à condition qu'aucune nouvelle panne ne se produise. Ainsi, aucune garantie n'est assurée pour les tâches exécutées tout ou partie durant une période de pannes transitoires. Par exemple, pour le problème de l'allocation de ressources locales, un algorithme instantanément stabilisant assure qu'un processus demandeur exécutera sa section critique sans conflit avec ses voisins, si aucune faute ne se produit entre sa demande et la fin de sa section critique.

Plan. Dans la section 2, nous présentons brièvement le modèle utilisé par notre algorithme. Puis, nous présentons nos propriétés de concurrence et leur utilisation pour le problème d'allocation de ressources locales dans la section 3. Nous présentons les idées principales de notre algorithme instantanément stabilisant réalisant la concurrence forte dans la section 4. Enfin, nous concluons dans la section 5.

2 Modèle

Nous considérons des systèmes distribués bidirectionnels et connexes, composés de n processus. Nous supposons que chaque processus est identifié de manière unique. Un processus p peut communiquer avec un sous-ensemble de processus appelés *voisins via* des *variables localement partagées* : p peut lire ses variables et celles de ses voisins mais ne peut écrire que dans ses propres variables. Nous notons $p.x$ la variable x du processus p. L'*état* d'un processus est défini par la valeur de ses variables. Une *configuration* du système est définie par l'état de chaque processus. Une *exécution* est une succession de *pas (atomiques) de calcul*. À chaque pas de calcul, chaque processus détermine, en fonction de son état et de celui de ses voisins, s'il est *activable*, *i.e.*, s'il doit modifier la valeur de ses variables. Un adversaire, le *démon*, choisit un sous-ensemble non vide de processus activables. Les processus choisis sont alors *activés* et mettent à jour leurs variables de façon atomique. Nous supposons que le démon est *faiblement équitable*, *i.e.*, si un processus est continûment activable, il est activé par le démon en temps fini.

3 Concurrence

Définition de la concurrence maximale. Intuitivement, un algorithme satisfait la concurrence maximale si, lorsque des processus demandeurs peuvent entrer en section critique sans violer la sûreté du problème considéré, alors au moins l'un d'entre eux finit par obtenir sa section critique, même si aucun autre processus ne termine sa section critique entre-temps.

Notons P_{Free} le sous-ensemble de processus demandeurs qui peuvent entrer en section critique sans violer la sûreté du problème considéré. Précisément, la *concurrence maximale* est composée de deux propriétés : une propriété d'*absence d'interblocage* et une propriété d'*absence de famine*. L'*absence d'interblocage* assure que, tant qu'il est possible de satisfaire la demande de certains processus (*i.e.*, tant que $P_{Free} \neq \emptyset$), l'algorithme ne se bloque pas (même s'il n'y a pas de nouvelle requête ni de processus qui termine sa section critique). L'*absence de famine* assure que, s'il est possible de satisfaire la demande de certains processus et

s'il n'y a ni nouvelle requête, ni sortie de section critique pendant un temps suffisamment long, alors l'un des processus demandeurs finit par quitter P_{Free} et entrer en section critique.

On peut définir de manière équivalente l'absence de famine par : s'il n'y a ni nouvelle requête ni sortie de section critique pendant un temps suffisamment long, alors tous les processus dont il était possible de satisfaire la demande ont fini par entrer en section critique, autrement dit, P_{Free} est devenu vide.

Généralisation de la concurrence maximale. Il n'est pas toujours possible d'assurer la concurrence maximale sans violer la vivacité du problème. Nous avons alors généralisé la concurrence maximale par la propriété de *concurrence partielle* afin d'exprimer des qualités de concurrence plus faible. Un algorithme satisfait la *concurrence partielle* pour $\mathcal{P}$, où $\mathcal{P}$ est un prédicat portant sur ensemble de processus et la configuration courante de l'algorithme, s'il satisfait les deux propriétés suivantes. L'*absence d'interblocage* pour $\mathcal{P}$ assure désormais que l'algorithme ne se bloque pas (même s'il n'y a pas de nouvelle requête ni de processus qui termine sa section critique), tant que P_{Free} n'est pas inclus dans un ensemble vérifiant $\mathcal{P}$ dans la configuration courante. L'*absence de famine* pour $\mathcal{P}$ assure que, s'il n'y a ni nouvelle requête ni sortie de section critique pendant un temps suffisamment long, alors P_{Free} finit par être inclus dans un ensemble vérifiant $\mathcal{P}$ dans la configuration courante.

Concurrence et allocation de ressources locales. Nous étudions la qualité de concurrence qui peut être réalisée par un algorithme d'allocation de ressources locales. Nous expliquons ici les idées principales de cette étude.

Tout d'abord, prenons un réseau quelconque et un processus p de ce réseau. Considérons l'exclusion mutuelle locale, une instance du problème d'allocation de ressources locales pour laquelle il y a un seul type de ressources qui est incompatible avec lui-même. Considérons l'exécution suivante. Supposons qu'un voisin de p, disons q_0, demande l'accès à une ressource. Les autres processus ne sont pas demandeurs. Par vivacité, q_0 obtient la ressource demandée en temps fini. Supposons ensuite que tous les voisins de p (sauf q_0 qui est déjà en section critique), et p lui-même, demandent une ressource (voir Figure 1). p ne peut entrer en section critique à cause de q_0 mais il est possible de servir les voisins demandeurs de p (sauf ceux qui sont eux-mêmes voisins avec q_0). Si l'algorithme satisfait la concurrence maximale et si q_0 conserve sa ressource pendant un temps suffisamment long, un des voisins de p, disons q_i, finit par être servi. q_0 relâche sa ressource au bout d'un moment (les sections critiques sont supposées finies), mais p ne peut toujours pas obtenir de ressource à cause de q_i. Si q_0 recommence à être demandeur, le système est dans une configuration similaire à celle de la Figure 1. En répétant ce principe, les voisins de p sont successivement en section critique mais la demande de p n'est jamais satisfaite. La vivacité est violée. Il est donc impossible de concevoir un algorithme d'allocation de ressources locales qui réalise la concurrence maximale.

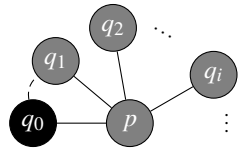

FIGURE 1: Voisinage de p. Les nœuds gris sont demandeurs, les nœuds noirs en section critique.

Ensuite, nous avons caractérisé avec la notion de *concurrence forte* la qualité de concurrence que nous pouvons obtenir. Appelons *voisinage conflictuel* du processus p dans la configuration γ, l'ensemble des voisins de p noté $\mathcal{CN}_p(\gamma)$ qui demandent ou utilisent une ressource incompatible avec la demande de p. On peut remarquer que pour prévenir le scénario ci-dessus, il faut empêcher les processus du voisinage conflictuel de p, sauf q (qui est déjà en section critique) et les voisins communs à p et q (déjà bloqués par q) d'entrer en section critique avant p, même s'ils peuvent être servis sans violer la sûreté. Ainsi, nous définissons la *concurrence forte* comme étant la concurrence partielle pour le prédicat $\mathcal{P}_{strong}$, où, pour tout ensemble de processus X et toute configuration γ, $\mathcal{P}_{strong}(X,\gamma)$ vaut $\exists p, \exists q \in \mathcal{CN}_P(\gamma), X = \mathcal{CN}_P(\gamma) \setminus \big(\{q\} \cup \mathcal{CN}_q(\gamma)\big)$. Dans la Section 4, nous proposons un algorithme d'allocation de ressources locales satisfaisant cette propriété.

4 Algorithme

Principe de l'algorithme. Chaque processus dispose d'une entrée *req* donnée par la couche applicative et qui représente la ressource demandée par le processus : si $p.req$ vaut $\perp$, le processus p n'est pas demandeur, sinon, p demande l'accès à la ressource $p.req$. L'accès à la section critique est géré grâce à une priorité

donnée par les identités des processus : le processus demandeur de plus grande identité est le seul autorisé à entrer en section critique. Une variable *status* permet de représenter le statut du processus : Out si le processus n'est pas demandeur, Wait si le processus est demandeur mais n'a pas encore obtenu la section critique et In si le processus est en section critique. Nous utilisons un statut supplémentaire, Blocked, pour permettre plus de concurrence. Lorsqu'un processus demandeur p ne peut entrer en section critique à cause d'un voisin q en train d'utiliser une ressource incompatible, p prend le statut Blocked. Ainsi, les voisins demandeurs de p ne le prennent plus en compte dans la compétition pour entrer en section critique et peuvent potentiellement obtenir l'accès à la ressource demandée, même si p a une identité plus grande.

Cependant, à cause de cette priorité basée sur l'identité, un minimum local pourrait ne jamais entrer en section critique (par exemple, si ses voisins de plus grandes identités demandent constamment des ressources incompatibles). Nous utilisons une circulation de jeton autostabilisante composée à notre algorithme pour assurer la vivacité dans ce cas. Lorsqu'un processus a le jeton, il est prioritaire sur les autres processus pour obtenir sa ressource, même sur ceux ayant une plus grande identité. De plus, un porteur de jeton de statut Blocked empêche ses voisins demandant une ressource incompatible d'entrer en section critique. Ces derniers prennent également le statut Blocked. Ainsi, un minimum local obtiendra le jeton en temps fini et ainsi finira par entrer en section critique. Notez que la circulation de jeton étant autostabilisante, il peut y avoir plusieurs jetons dans une même configuration, mais uniquement pendant la phase de stabilisation. Si deux voisins demandeurs sont tous deux porteurs d'un jeton, c'est celui de plus grande identité qui est prioritaire.

Concurrence forte. Nous avons ensuite étudié la qualité de concurrence de notre algorithme. Supposons que les processus en section critique ne relâchent pas leur ressource et qu'il n'y ait plus de nouvelle requête pendant un temps arbitrairement long. Au bout d'un moment, les processus ne vont plus changer de statut. En effet, les processus en section critique gardent le statut In, les processus non demandeurs gardent le statut Out et le processus demandeurs ne pouvant entrer en section critique sans violer la sûreté gardent le statut Blocked. S'il reste des processus dans P_{Free}, ils sont bloqués par le porteur du jeton, lui même bloqué par un voisin en section critique utilisant une ressource incompatible. P_{Free} contient donc au pire le voisinage conflictuel du porteur de jeton p moins son voisin q déjà en section critique et les voisins communs à p et q bloqués par q lui-même. L'algorithme satisfait donc la concurrence forte.

5 Conclusion

Par manque de place, des résultats sont omis de cet article. En particulier, nous avons étudié, pour l'algorithme proposé, le temps d'attente d'un processus entre la demande d'accès à une ressource et l'obtention de cet accès. Ce temps d'attente est en $O(n)$ rondes [‡]. Une perspective naturelle de ce travail est d'étendre l'étude de la qualité de concurrence à des classes de problèmes d'allocation de ressources plus grandes.

Références

[ADD15] Karine Altisen, Stéphane Devismes, and Anaïs Durand. Concurrency in snap-stabilizing local resource allocation. In *NETYS*, 2015.

[BDP11] Borzoo Bonakdarpour, Stéphane Devismes, and Franck Petit. Snap-stabilizing comittee coordination. In *IPDPS*, pages 231–242, 2011.

[CDP03] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *SSS*, pages 102–112, 2003.

[DHV03] Ajoy Kumar Datta, Rachid Hadid, and Vincent Villain. A self-stabilizing token-based k-out-of-l-exclusion algorithm. *Concurrency and Comput. Pract. Exper.*, 15(11-12) :1069–1091, 2003.

[FLBB79] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *FOCS*, pages 234–254, 1979.

‡. La ronde est une unité de mesure en fonction de la vitesse du processus le plus lent.