Compas'2015 : Parallélisme / Architecture / Système
Lille, France, du 30 juin au 3 juillet 2015

Simplifier le verrouillage d'assemblages de composants haute performance à pile d'appel

Vincent Lanore

ENS de Lyon, LIP, équipe Avalon
vincent.lanore@ens-lyon.fr

Résumé
La programmation à base de composants facilite grandement l'adaptation d'applications. Cette propriété est très utile en HPC où les applications ont une grande durée de vie et sont très complexes. Toutefois, les modèles à composants existants supportent très mal les applications HPC dont la structure change à l'exécution, comme le raffinement de maillage adaptatif (AMR). L'un des problèmes qui se posent est le verrouillage efficace de sous-assemblages, en particulier lorsque le modèle d'exécution repose sur des piles d'appel. Cet article présente un algorithme capable d'automatiser certaines tâches de verrouillage et évalue cet algorithme sur un assemblage d'AMR.

Mots-clés : Composants logiciels ; reconfiguration ; algorithmes de verrouillage ; calcul haute performance.

1. Introduction

La programmation par composants est un paradigme de programmation connu pour faciliter la réutilisation et l'adaptation de code. Ce paradigme propose de construire une application en assemblant des morceaux de codes aux interfaces bien définies appelés *composants logiciels*. La programmation par composants a déjà été appliquée avec succès aux applications HPC [1] et au cas particulier de l'adaptation statique [7, 2].
Certaines applications à haute performance sont dynamiques, c'est-à-dire que la topologie des communications ou des données est susceptible de changer en cours d'exécution. Les exemples de telles applications incluent le raffinement de maillage adaptatif (AMR) et les applications faisant de l'équilibrage de charge dynamique. Lorsque les modification de la structure d'une application sont opérées de façon concurrente, la question du maintien de la cohérence au cours de l'exécution se pose.
dont la structure change à l'exécution est
un problème de synchronisation qui peut être très difficile, d'autant plus que les travaux récents prônent des approches de moins en moins centralisées [5].
Malheureusement, les modèles à composants HPC traditionnels, comme L2C [2] ou CCA [1], fournissent les fonctionnalités de base permettant la reconfiguration (création, destruction et connexion de composants) mais laissent la synchronisation à la charge des programmeurs de composants. Certains modèles à composants de la littérature non-HPC proposent de simplifier le problème en introduisant la notion de *quiescent state* : si une portion d'assemblage n'est pas

en train de s'exécuter ou d'être reconfigurée alors on peut la reconfigurer sans difficulté. Pour qu'une portion d'assemblage atteigne un *quiescent state*, celle-ci doit être *verrouillée*. Malheureusement, le verrouillage d'un assemblage en cours d'exécution est un problème difficile qui est résolu dans la littérature en utilisant des approches coûteuses comme le verrouillage global. De plus, les algorithmes de verrouillage de la littérature reposent souvent sur des modèles d'exécution avec de bonnes propriétés (par ex., messages asynchrones, continuations) ce qui est a priori difficile à implémenter sans perte de performance par dessus un langage HPC à pile d'appel type C/FORTRAN.
directMOD est un modèle à composants qui a été proposé pour faire de la reconfiguration d'assemblage dans un contexte HPC [6]. *directMOD* propose que la reconfiguration et le verrouillage soient écrits au niveau de l'assemblage plutôt que laissés à la charge des programmeurs de composants. Cette approche permet d'écrire la reconfiguration et le verrouillage sur un modèle de plus haut niveau et permet de mieux compartimenter le code. Malheureusement, le code de verrouillage demeure très dépendant des propriétés de contrôle de l'assemblage et est difficile à réutiliser.
Le présent article propose un algorithme capable d'automatiser certaines tâches de verrouillage pour simplifier le développement et l'adaptation des applications HPC dynamiques à pile d'appel. Une évaluation préliminaire de l'approche est faite en s'appuyant sur deux assemblages d'AMR à composants l'un développé avec *directMOD* et notre algorithme et l'autre dans un modèle à composant HPC classique.
La structure du présent article est la suivante : la section 2 présente un modèle à composants générique qui va nous permettre de présenter l'algorithme ; la section 3 introduit notre algorithme ; la section 4 présente une évaluation préliminaire de l'approche ; enfin, la section 5 conclut.

2. Modèle

Cette section présente un modèle à composants qui permet de présenter notre algorithme de verrouillage tout en restant le plus général possible. Le modèle proposé peut être vu comme une généralisation de modèles comme L2C ou CCA.

2.1. Modèle d'assemblage

Le modèle est constitué des éléments suivants :
– Un *composant* est une instance d'un *type de composant* et possède une *interface* constituée d'une liste de *propriétés*.
– Une propriété est soit un *port* qui expose des fonctionnalités (par exemple, une interface objet ou un service) soit une *référence* vers un port.
– Un *assemblage* est un ensemble de composants.
On appelle *sous-assemblage* d'un assemblage A tout sous-ensemble de composants et de propriétés de A. Un sous-ensemble peut notamment figurer des propriétés sans le composant auquel elles appartiennent ou bien figurer des composants qui n'ont pas toutes leurs propriétés (*e.g*, figure 1.b).
La figure 1.a) présente un assemblage qui implémente une grille 2D de composants. La figure 1.b) présente un sous-assemblage de l'assemblage de 1.a).
Par rapport à la littérature, ce modèle a la particularité d'être *primitif* (il ne permet pas d'implémenter un composant par un sous-assemblage) et de n'avoir que des connexions point-à-point.

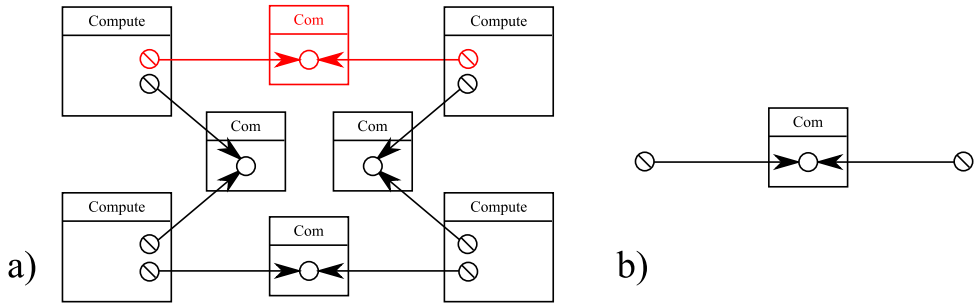


FIGURE 1 – Les composants sont représentés par des rectangles, les ports par des cercles, et les références par des cercles barrés qu'une arête relie au port référencé. Chaque composant a un entête avec son type (par exemple, `Compute`). a) Un assemblage de composants implémentant une grille 2D avec communications asynchrones. b) Un sous-assemblage de l'assemblage de la figure a) correspondant à la partie en rouge.

2.2. Modèle d'exécution

Afin de capturer les difficultés inhérentes aux modèles à composants à pile d'appel, nous avons choisi de définir un modèle d'exécution *multithreadé à pile d'appel*.

À un instant donné, l'état du système est donné par un *ensemble de piles d'appel* Chaque *pile d'appel* est une liste ordonnée de ports. À tout moment, les opérations suivantes peuvent se produire de façon non-déterministe :

– Un port p est ajouté à une pile d'appel dont le port de tête appartient à un composant qui possède une référence vers p (appel sur un port).
– Un port est dépilé d'une pile d'appel (retour).
– Une nouvelle pile d'appel est créée (création de thread).
– Une pile d'appel vide est détruite (fin d'un thread).

L'état du système est dit *bien formé* si et seulement si toute paire des ports successifs dans une pile correspond à un appel possible dans l'assemblage. Un système pourrait se retrouver dans un état non bien formé si par exemple des composants étaient supprimés alors qu'ils avaient des appels de méthode en cours.

La figure 2 illustre la définition de l'état d'un système sur un exemple et présente des exemples mal formés. Les contraintes illustrées sur cet exemple sont typiques des langages à pile d'appel comme le C++ et des langages à composants qui en dérivent.

Le fait que ces opérations se passent de façon non-déterministe modélise le fait que les composants sont des boîtes noires. A priori, aucune hypothèse ne peut être faite sur le comportement des composants au niveau de l'assemblage.

3. Algorithme de verrouillage

Maintenant qu'un modèle a été posé et que le type d'assemblage qui nous intéresse a été défini, on étend ce modèle avec des opérations de verrouillage et un algorithme utilisant ces opérations.

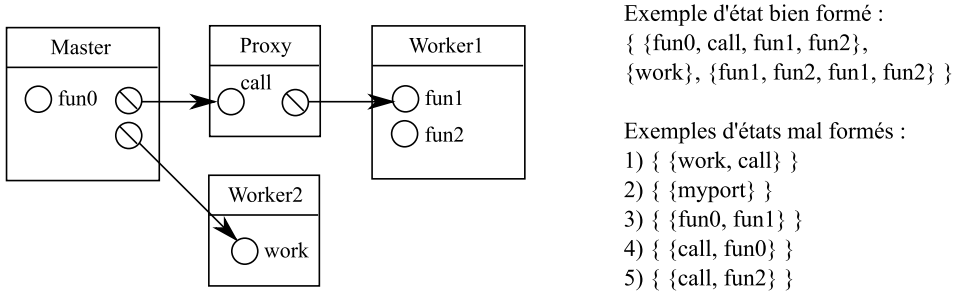


FIGURE 2 – Assemblage accompagné d'exemples de piles d'appels bien en mal formées. L'état bien formé est composé d'une pile d'appel partant du `Master` et allant jusqu'à `fun2` en passant par le `Proxy`, d'un thread local créé par `Worker2` et d'un thread interne à `Worker1` qui alterne des appels imbriqués à `fun1` et `fun2`. Les raisons pour lesquelles les piles sont mal formées sont les suivantes : 1) pas de référence, appel impossible 2) port qui n'existe pas ou plus 3) les références ne sont pas transitives 4) les références sont orientées 5) les références pointent vers un port spécifique.

3.1. Le verrouillage par mutex

Le modèle présenté jusqu'ici se contente de modéliser l'assemblage et le comportement des composants à l'exécution. Afin de permettre d'agir sur un assemblage en train de s'exécuter, par exemple pour le verrouiller, on introduit un ensemble d'opérations qui peuvent être faites au niveau de l'assemblage :

– `lock(ref)` verrouille la référence `ref`. Une référence verrouillée ne peut plus être utilisée pour faire un appel de port. Si une pile d'appel utilise `ref` au moment où l'on essaye de la verouiller alors l'opération est bloquante jusqu'à ce que plus aucune pile d'appel ne traverse `ref`.
– `unlock(ref)` déverrouille la référence `ref` afin qu'elle puisse à nouveau être utilisée par les composants.

De plus, on dit qu'un sous-assemblage `subassembly` est *verrouillé* (ce qui correspond à être en *quiescent state* au sens de la littérature) si et seulement si aucune pile d'appel n'inclut de port appartenant à `subassembly`, toutes les références menant à `subassembly` sont verrouillées et aucune reconfiguration n'est en cours sur `subassembly`.

Si chaque reconfiguration est faite sur un sous-assemblage verrouillé, cela garantit qu'aucune pile d'appel ne contienne d'élément du sous-assemblage reconfiguré et ainsi garantit que la reconfiguration préserve le caractère bien formé des piles d'appel.

3.2. Métadonnées de contrôle

Verrouiller un sous-assemblage grâce aux opérations définies plus haut n'est pas a priori un problème facile. En effet, sans plus de garanties sur le comportement des composants, une opération `lock` peut provoquer un interblocage. De plus, les ports verrouillés par `lock` ne sont plus utilisables et peuvent provoquer des interblocages.

Afin de résoudre ce problème, on propose de munir les composants de métadonnées qui décrivent certaines propriétés simples de contrôle. Ces informations doivent à la fois être pertinentes pour le problème du verrouillage et être faciles à obtenir.

On propose d'associer à chaque port `p` les deux propriétés suivantes :

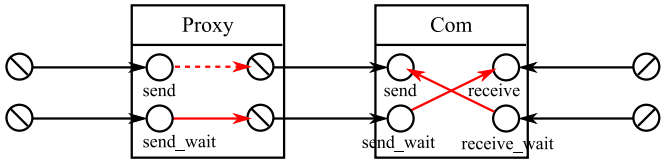


FIGURE 3 – Un sous-assemblage implémentant un connecteur asynchrone et un proxy, munis de leurs métadonnées de contrôle. Les flèches rouges pointillées représentent les dépendances d'appel tandis que les flèches rouges pleines représentent les dépendances de terminaison.

– la liste des propriétés appartenant à la même interface susceptibles d'être utilisées par p lors d'une exécution ; on appelle chaque propriété une *dépendance d'appel* ;
– si un appel sur p est susceptible d'être bloquant, la liste des propriétés dont la terminaison dépend ; on appelle chaque propriété une *dépendance de terminaison*.

La figure 3, présente un sous-assemblage implémentant une connexion asynchrone et un proxy. Les ports $send$ et $receive$ sont des ports non bloquants tandis que les ports $send\_wait$ et $receive\_wait$ sont bloquants et dépendent respectivement de $receive$ et de $send$ pour terminer. Le port $send\_wait$ a par exemple une dépendance de contrôle vers $receive$ car les appels à $send\_wait$ sont bloquants et ont besoin que des appels soient faits sur $receive$ pour terminer.

3.3. Algorithme

Nous proposons un algorithme qui, à partir d'un sous-assemblage muni de ses métadonnées de contrôle, calcule un algorithme de verrouillage à certaines conditions. La figure 4 illustre cet algorithme sur l'exemple du connecteur asynchrone présenté dans la figure 3.

L'algorithme est constitué des étapes suivantes :
– un graphe de dépendances de contrôle est construit à partir du graphe de propriétés du sous-assemblage ;
– un tri topologique du graphe de dépendances de contrôle est effectué ;
– un algorithme de verrouillage est déduit du graphe trié.

Tout d'abord, on construit un graphe dont les sommets sont les références du sous-assemblage et dont les arrêtes sont des contraintes de terminaison. Les arrêtes sont construites à l'aide des règles suivantes :
– La référence a est connectée à la référence b si il existe dans l'assemblage un chemin orienté de a à b passant par au moins une dépendance de terminaison (dans le sens a vers b) et un nombre quelconque d'arêtes de référence et de dépendances d'appel dans n'importe quel sens. Cette arête représente le fait que si b est verrouillé avant a alors a risque de se retrouver dans un état bloquant et de provoquer un interblocage.
– La référence a est connectée à la référence b si il existe dans l'assemblage un chemin orienté de a à b passant par au moins une dépendance d'appel (dans le sens a vers b) et un nombre quelconque d'arêtes de référence dans n'importe quel sens. Cette arête représente le fait que a risque d'appeler b et que si b est bloqué avant a on peut se retrouver dans un cas d'interblocage.

Une fois ce graphe construit, on effectue un tri topologique. Par construction, ce tri nous donne un ordre de verrouillage qui respecte toutes les contraintes du graphe de contraintes de terminaison. Cet algorithme garantit le verrouillage d'un sous-assemblage aux conditions suivantes :
– les références entrantes ne sont jamais bloquées ;

– le sous-assemblage verrouillé ne contient pas de boucle de dépendance (si c'est le cas, l'algorithme ne termine pas car le tri topologique ne peut pas être fait).

3.4. Discussion et travaux connexes

Au registre des critiques, on notera que les conditions d'application de notre algorithme ne sont pas toujours possibles à garantir, typiquement lorsque le sous-assemblage à verrouiller est complexe (forte probabilité de boucle de dépendance). On notera également que notre approche ne garantit pas le temps de terminaison et nécessite que les interfaces soient décomposées à grain assez fin pour que les métadonnées soient pertinentes.

Toutefois, cet algorithme est bien adapté au cas des assemblages de connecteurs et de proxys comme celui présenté dans la figure 4. De tels assemblages sont typiques des exemples type stencil distribué (qui incluent l'AMR) où de nombreuses variantes de connecteurs peuvent coexister (différentes arités, connexions locales ou distantes, proxies) et où ne pas avoir à écrire d'algorithme de verrouillage pour chacune fait gagner beaucoup de temps.

Comme discuté dans l'introduction, ce travail est nouveau par rapport à la littérature composants dans le sens où il est le seul à notre connaissance à s'intéresser au problème du verrouillage d'assemblage à pile d'appel dans un contexte HPC.

Outre la littérature composants, il existe une littérature très riche sur la prévention et la détection d'interblocages. En particulier, notre travail adopte une méthode proche de celle des travaux qui font de l'analyse de code objet annoté pour empêcher les interblocages (par ex., [4, 3]). Ces travaux reposent toutefois sur de l'analyse de code, procédé incompatible avec l'approche composants où l'utilisation d'un composant doit se faire à partir de sa seule interface. De plus, notre algorithme repose sur des métadonnées simples à obtenir et peut s'exécuter en ligne pendant l'exécution.

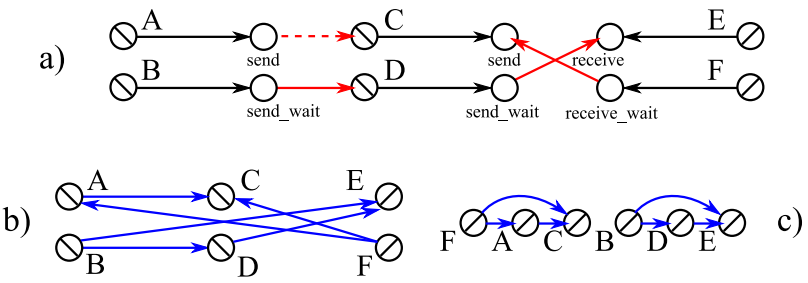


FIGURE 4 – Exécution de notre algorithme de verrouillage sur le sous-assemblage de la figure 3. La figure a) est le graphe des propriétés, la figure b) le graphe des contraintes de terminaison et la figure c) est un tri topologique de la figure b). Les flèches bleues représentent les contraintes de terminaison.

4. Évaluation

Afin d'évaluer les bénéfices de notre approche, nous avons implémenté un ensemble de composants qui utilise *directL2C*, une implémentation de *directMOD* basée sur L2C qui intègre les

propositions du présent article. Ces composants permettent de construire une série de benchmarks reprenant la structure de l'AMR.
Dans un premier temps, nous nous sommes intéressé à AMRdirect, un assemblage implémentant une AMR 2D multithreadée uniquement descendante (pas de déraffinement). Cet assemblage a une structure similaire à celle présentée dans la figure 1 et le raffinement d'un composant de calcul correspond à la séparation en quatre composants identiques et la reconfiguration des connexions avec les voisins. La figure 5b présente les performances de cet assemblage sur une expérience "à vide" (les composants ne calculent pas mais se synchronisent quand même avec leurs voisins et se raffinent) et régulière (chaque composant initial va subir le même nombre de raffinements au cours de l'expérience). On constate que les performances de notre assemblage passent bien à l'échelle jusqu'à 4096 threads. Les performances mesurées dépendent largement de l'ordonnanceur de threads mais montrent tout de même que notre modèle n'introduit pas de surcoût mesurable à cette échelle lorsque la taille de l'assemblage augmente.
Dans un second temps nous avons comparé ce premier assemblage à AMRL2C, assemblage programmé en L2C (un modèle à composants HPC qui laisse le verrouillage à la charge du programmeur de composants) qui implémente le même benchmark et qui a également été programmé par nos soins. Le tableau 5a présente la taille des codes respectifs des composants des deux assemblages. L'assemblage *directL2C* est beaucoup plus petit car de nombreuses fonctionnalités sont fournies par *directL2C* et parce que notre algorithme permet de ne pas avoir à écrire une seule ligne de code de verrouillage.
De plus, il est possible d'implémenter à faible coût (cf ComHybrid et MpiProxy dans le tableau 5a) d'autres types de connecteurs comme des connecteurs MPI distribués. Notre algorithme de verrouillage permet d'obtenir gratuitement le verrouillage de toutes ces variantes de connecteurs. Des variantes distribuées hybrides (MPI pour les communications distantes et C++ pour les communications locales) ont été implémentée en s'appuyant sur MadMPI [8] (pour gérer MPI_THREAD_MULTIPLE) et l'étude de leurs performances est en cours sur la plate-forme Grid'5000.

5. Conclusion

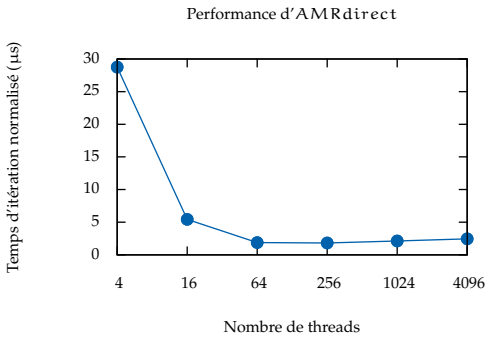
Le présent article a présenté le problème du verrouillage d'assemblage de composants HPC parallèles à pile d'appel, a présenté un algorithme qui permet de le résoudre automatiquement dans certains cas et a évalué cette approche sur une implémentation d'un benchmark AMR. L'algorithme proposé permet d'automatiser le verrouillage des connecteurs et, couplé à au modèle *directMOD* et à son implémentation *directL2C*, permet de simplifier l'écriture d'assemblages dynamiques.
Les perspectives incluent une étude de l'impact en termes de performance du verrouillage par mutex. Il s'agit a priori d'une approche efficace et il serait intéressant de le quantifier et, plus globalement, de valider les performances de l'approche sur de grandes architectures distribuées. D'autre part, il serait intéressant d'établir de façon plus formelle la classe de sous-assemblages verrouillée par notre algorithme.

Bibliographie

1. Bernholdt (D. E.), Allan (B. A.), Armstrong (R.), Bertrand (F.), Chiu (K.), Dahlgren (T. L.), Damevski (K.), Elwasif (W. R.), Epperly (T. G.), Govindaraju (M.) et al. – A component

Composant	AMRL2C	AMRdirect
Init	137	23 *
Compute		16
Iter		29
AMR2D	166	52
Com	198	26
Transformation		19 *
(ComHybrid)		(43)
(MpiProxy)		(16)
TOTAL MT	501	146

(a) Taille du code des deux assemblages AMR. Les tailles sont en nombre de lignes signifiantes (ni un commentaire ni une fermeture de boucle). Tous les codes sont en C++ sauf les deux marquées d'une astérisque qui sont des transformations *directL2C*. La ligne TOTAL MT est la taille totale du code pour l'exemple multithreadé seulement.



(b) Temps d'itération normalisé par le nombre de connecteurs en fonction du nombre de threads. Les composants ne calculent pas. Les temps sont les médianes sur > 10 exécutions pour chaque point. Les expériences ont été faites sur un Intel i7-3520M bi-cœur à 2.9GHz.

FIGURE 5 – Comparaison de AMRL2C et de AMRdirect en termes de performance et de taille de code.

architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, vol. 20, n2, 2006, p. 163–202.

2. Bigot (J.), Hou (Z.), Pérez (C.) et Pichon (V.). – A low level component model easing performance portability of HPC applications. *Computing*, 2013, p. 1–16.
3. Boyapati (C.), Lee (R.) et Rinard (M.). – Ownership types for safe programming : Preventing data races and deadlocks. – In *ACM SIGPLAN Notices*, number 11, p. 211–230. ACM, 2002.
4. Joshi (P.), Naik (M.), Sen (K.) et Gay (D.). – An effective dynamic analysis for detecting generalized deadlocks. – In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, p. 327–336. ACM, 2010.
5. Langer (A.), Lifflander (J.), Miller (P.), Pan (K.-C.), Kale (L.) et Ricker (P.). – Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. – In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, p. 100–107, Oct 2012.
6. Lanore (V.) et Pérez (C.). – A Reconfigurable Component Model for HPC. – In *The 18th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'2015)*. ACM, 2015.
7. Richard (J.), Lanore (V.) et Pérez (C.). – Evaluating Component Assembly Specialization for 3D FFT, juillet 2014. PRACE whitepaper.
8. Trahay (F.), Brunet (E.) et Denis (A.). – An analysis of the impact of multi-threading on communication performance. – In *CAC 2009 : The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*, Rome, Italy, mai 2009. IEEE Computer Society Press.