Un modèle pour l'adaptation dynamique des programmes parallèles

Jérémy Buisson

RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Un modèle pour l'adaptation dynamique des programmes parallèles

Jérémy Buisson

IRISA/INSA de Rennes
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 RENNES CEDEX - France

Résumé
Les environnements d'exécution tels que les grilles de calcul ou les fédérations de grappes ont leurs caractéristiques (telles que la bande passante) qui fluctuent au cours de l'exécution des programmes. C'est pourquoi il est nécessaire que les programmes soient capables de prendre en compte les changements de leur environnement d'exécution. L'adaptation dynamique est une technique qui a pour but de résoudre cette problématique. Cet article présente un modèle d'adaptation dynamique qui a été étendu au cas des codes parallèles.

Mots-clés : adaptation dynamique, parallélisme, composants logiciels

---

1. Introduction

L'émergence d'environnements d'exécution tels que les fédérations de grappes et les grilles de calcul apporte aux environnements parallèles une propriété additionnelle : leurs caractéristiques fluctuent. En effet, le partage des ressources et leur distribution dans des domaines d'administration distincts ne permet plus de garantir qu'une ressource attribuée à un programme le sera jusqu'au terme de son exécution. Symétriquement, lorsqu'un programme se termine, les ressources libérées peuvent être mise à la disposition d'un autre programme avant que ce programme ne se termine. Ainsi, l'allocation des ressources à un programme peut changer au cours de son exécution. Il est nécessaire que le programme tienne compte de ces changements afin de profiter à chaque instant de l'intégralité des ressources qui lui sont affectées.
L'adaptation dynamique vise à permettre à un programme de se modifier au cours de son exécution pour tenir compte des avertissements qu'il reçoit et qui décrivent les changements de son environnement d'exécution. Il s'agit d'une approche complémentaire aux mécanismes de tolérance aux fautes. Ces derniers permettent de prendre en compte les disparitions de ressources. Cependant, contrairement à l'adaptation dynamique, ils ne tiennent généralement pas compte de l'attribution de nouvelles ressources au programme, ni ne tirent profit d'un environnement qui reprendrait sciemment des ressources à un programme (alors qu'on peut raisonnablement attendre d'un tel environnement qu'il avertisse à l'avance le programme de ses décisions).
La section 2 décrit un modèle d'adaptation dynamique et ses conséquences sur l'exécution des programmes. La section 3 présente comment ce modèle se transpose au cas des programmes parallèles. La section 4 présente les autres travaux traitant du problème de l'adaptation dynamique.

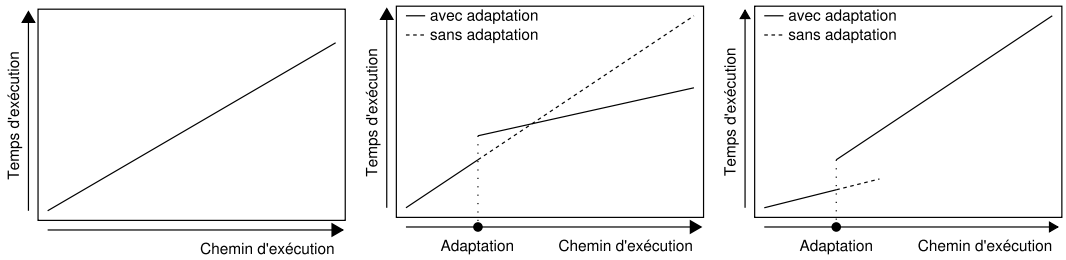
2. Modèle de l'adaptation dynamique d'un programme

Conceptuellement, l'adaptation dynamique consiste à modifier un programme pendant qu'il s'exécute. L'objectif de cette modification est d'optimiser ce programme selon un certain critère dans un environnement où les paramètres de ce critère sont connus et fluctuent au cours de l'exécution du programme. Dans ce travail, il s'agit de minimiser le temps nécessaire au programme pour se terminer compte-tenu

des ressources qui lui sont attribuées. Pour cela, nous considérons que le programme peut être modifié en changeant ses paramètres (tels que le nombre de processeurs qu'il utilise) ou en remplaçant son algorithme par un autre.

2.1. Modèle d'exécution

Un programme a schématiquement son exécution qui progresse de manière uniforme comme le montre la figure 1(a). Si ce programme s'adapte dynamiquement, sa vitesse de progression de l'exécution varie. La figure 1(b) montre la progression d'un programme qui s'adapte suite à l'attribution de nouvelles ressources : après adaptation, le programme utilise les nouvelles ressources qui lui sont allouées. Il s'exécute plus rapidement s'il est capable de tirer parti de ces ressources.
La figure 1(c) montre la progression d'un programme qui s'adapte en prévision de la disparition de ressources : après adaptation, le programme n'utilise plus les ressources qui lui seront reprises. Il s'exécute alors plus lentement. S'il ne s'était pas adapté et en l'absence de mécanisme de tolérance aux fautes, ce programme se serait arrêté brutalement.



(a) Sans adaptation

(b) Adaptation suite à l'attribution de nouvelles ressources

(c) Adaptation en prévision de la disparition de ressources

FIG. 1 – Progression de l'exécution d'un programme

Lorsque le programme s'adapte, il y a une discontinuité dans le temps d'exécution dûe au temps requis pour modifier le programme. Ce surcoût a un impact à court terme sur le temps nécessaire au programme pour se terminer. Ceci implique que si le programme est trop proche de son terme, il n'est pas intéressant de l'adapter.

2.2. Questions de l'adaptation

L'adaptation dynamique d'un programme pose trois questions.

Quand le programme doit-il s'adapter ?

Comment le programme doit-il être modifié ?

Où dans son chemin d'exécution le programme doit-il être modifié ?

2.2.1. Quand et comment adapter un programme

La définition donnée à l'adaptation dynamique répond intrinsèquement à ces deux questions.
Il faut adapter un programme lorsque les ressources qui lui sont attribuées changent. L'adaptation dynamique peut faire suite à l'ajout de ressources ; ou bien précéder le retrait de ressources. La modification du programme doit lui permettre de prendre en compte ces changements. Elle est propre à chaque programme.
Si le programme ne peut se modifier avant le retrait de ressources, ce changement est une faute de l'environnement d'exécution. Le programme doit alors intégrer le mécanisme nécessaire pour la tolérer (que nous n'étudions pas ici).

2.2.2. Où adapter un programme

La figure 2 donne les éléments pour répondre à cette question. Cette figure indique pour un programme qui s'adapte la durée qui lui sera nécessaire pour terminer son exécution en fonction de sa position courante (*présent*) dans le chemin d'exécution et de l'endroit où il s'adapte.

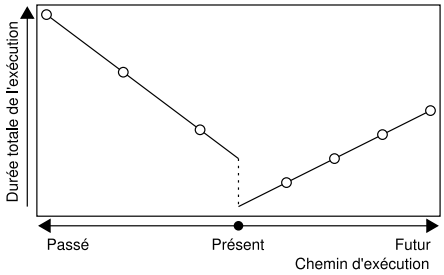


FIG. 2 – Modèle de performance d'un programme qui s'adapte

Performance

La position courante sépare le chemin d'exécution en deux vis-à-vis du critère de performance :

- Le programme peut être modifié dans une position qui succède à la position courante (adaptation dans le *futur*). Dans ce cas, plus cette position est éloignée de la position courante, plus longtemps l'exécution du programme est sous-optimale, plus il lui faut de temps pour terminer son exécution.
- Le programme peut être modifié dans une position qui précède la position courante (adaptation dans le *passé*) grâce à un mécanisme similaire aux points de reprise. Il est alors nécessaire de restaurer un état antérieur du programme. Dans ce cas, plus cette position est éloignée de la position courante, plus d'instructions doivent être réexécutées, plus il faut de temps au programme pour terminer son exécution.

Les deux cas extrêmes sont celui où la modification n'est faite qu'à la fin du programme, ce qui est équivalent à ne pas adapter le programme, et celui où la modification se fait au début du programme, ce qui est équivalent à arrêter le programme pour le réexécuter depuis le début plutôt que de l'adapter dynamiquement.

La discontinuité de la courbe au niveau de la position courante montre le surcoût de la restauration d'un état antérieur. En effet, cette restauration est nécessaire pour modifier le programme juste avant la position courante alors qu'elle ne l'est pas pour modifier le programme juste après.

Intégrité

Un programme qui s'adapte doit conserver l'intégrité de son exécution. La modification du programme doit le transformer de telle sorte que le résultat du programme soit (sémantiquement) inchangé. Pour cette raison, il n'est pas possible de modifier un programme depuis n'importe quelle position dans son chemin d'exécution. On appelle *point* les endroits où un programme peut être modifié tout en conservant l'intégrité de son exécution.

Ces points forment une discrétisation du chemin de l'exécution au regard de l'adaptation dynamique. Ils sont représentés par les disques blancs sur la courbe de la figure 2. Sur cette courbe, les points sont plus fréquents dans le futur que dans le passé. En effet, dans le cas où aucune adaptation ne se produit, un point dans le passé requiert la mémorisation d'un état du programme ; alors qu'un point dans le futur ne demande aucune action. Le surcoût engendré par un point dans le passé est donc *a priori* supérieur à celui d'un point dans le futur.

3. Modèle de l'adaptation dynamique d'un programme parallèle

Un programme parallèle est un programme dans lequel plusieurs activités concurrentes communiquent pour accomplir ensemble un même calcul. Chacune de ces activités a son propre chemin d'exécution. Cela a une incidence sur le modèle décrit à la section 2.2.2 pour déterminer où le programme doit être modifié.
La figure 3 montre le modèle de performance d'un programme parallèle qui s'adapte. Ce modèle dépend de la position courante de chacunes des activités du programme parallèle. Comme les activités ne sont pas synchronisées, ces positions courantes ($P1$, $P2$ et $P3$ sur la figure) ne sont pas au même endroit dans le chemin d'exécution. De plus, ces activités ne partagent pas nécessairement le même chemin. C'est pourquoi la présence d'un point pour une activité n'implique pas que le point soit présent pour toutes les activités.

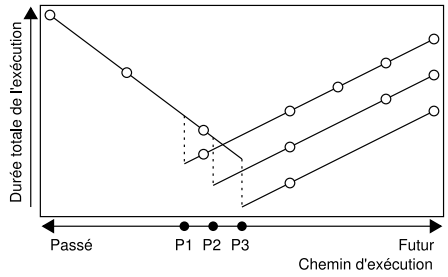


FIG. 3 – Modèle de performance d'un programme parallèle qui s'adapte

3.1. Performance

Chaque activité est modifiée à partir d'un point de sa courbe. Cette collection de points forme un *point global*. Comme le temps nécessaire au programme parallèle pour se terminer est le maximum des temps nécessaire à chaque activité pour terminer, le coût d'un point global est déterminé comme le maximum des coûts des points pour chacune des activités.

3.2. Intégrité

Le programme parallèle est modifié lorsqu'il atteint un point global. Pour garantir l'intégrité de son exécution, il est nécessaire que ce point global vérifie une relation de cohérence. Cette relation est propre à chaque programme.
Dans le cas particulier d'un point global dans le passé, la modification doit pouvoir être exécutée et le programme redémarré une fois modifié. Ceci suppose qu'à ce point global, l'état mémorisé reflète les communications en cours et respecte la relation de causalité.

3.3. Exemple de relation de cohérence : cas des programmes SPMD

Les programmes SPMD sont dans le contexte de l'adaptation dynamique des programmes dont toutes les activités concurrentes partagent le même graphe de flot de contrôle entre les points. Pour cette classe de programmes, on peut choisir l'identité comme relation de cohérence.

> Un point global est cohérent si et seulement si toutes les activités concurrentes sont au même point local.

Par même point, on entend même nœud dans le graphe de flot de contrôle et mêmes valeurs pour les indices dans les espaces d'itération. Lorsque les activités d'un programmes SPMD vérifient cette relation de cohérence, il est possible de restructurer le programme. En effet, les données étant dans le même état, il est possible de les redistribuer pour ajouter ou retirer au programme des processeurs.

4. Travaux apparentés

Plusieurs travaux ont étudié le problème de l'adaptation dynamique. Une méthodologie est définie dans [12] pour la construction de programmes capables de s'adapter. Certains de ces travaux tels que [4, 13] sont focalisés sur les mécanismes de prise de décision pour l'adaptation dynamique. D'autres tels que [1, 6, 7, 9, 10, 15] ont étudié plus particulièrement comment modifier un programme. Parmi ceux-ci, les projets [1, 7, 9, 10] ont étudié cette problématique dans le cadre de programmes parallèles. Des travaux tels que [5, 8] ont également étudié la problématique de l'adaptation dynamique dans le contexte de plusieurs programmes distribués coopérants.
Une proposition a été faite dans [3] pour permettre aux mécanismes de tolérance aux fautes par points de reprise de restructurer une application. Les restructurations consistent à ajouter ou supprimer des activités en cours d'exécution. Pour cela, ce travail suppose que le programme est construit sous forme SPMD comme une séquence de phases de calculs séparées par des barrières de synchronisation. Comme ce mécanisme est indépendant du programme, il impose des contraintes techniques au programmeur notamment sur la gestion de la mémoire et l'allocation des variables.
Parmi ces travaux, peu ont étudié les problèmes d'intégrité que l'adaptation peut causer.

5. Conclusion

Ce travail vise principalement à définir un modèle pour l'adaptation dynamique qui permette de prendre en compte les codes parallèles. Il complète l'architecture décrite dans [2]. Actuellement, la notion de point est définie spécifiquement à chaque programme. De même, la relation de cohérence sur les points globaux nécessaire dans le cas des codes parallèles a été proposée uniquement dans le cas de codes SPMD. Il faudra donc poursuivre le travail de définition.
Dans le futur de ce travail, nous étudierons la relation entre points et points de reprise. En effet, il apparaît que tolérance aux fautes et adaptation dynamique sont deux approches complémentaires pour gérer la dynamicité de l'environnement d'exécution. Ainsi, il serait possible d'exploiter l'infrastructure de l'adaptation dynamique pour coordonner statiquement et sans synchronisation la prise de points de reprise. Symétriquement, l'adaptation dynamique pourrait tirer profit d'un mécanisme de points de reprise pour mettre en œuvre les adaptations dans le passé.
Un second axe de notre travail concernera les mécanismes de prise de décision. Nous étendrons le travail déjà effectué dans le cadre du projet ACEEL [5, 6]. En particulier, les règles explicites d'adaptation peuvent être complétées par des modèles de performance des différents comportements et réactions du composant logiciel. Ceci devrait permettre de générer automatiquement certaines des règles de la politique d'adaptation. Ces modèles de performance pourront être affinés par des mesures de performance des ports du composant.
Le travail sur le mécanisme de prise de décision s'articulera également autour d'une extension du rôle du décideur. Il ne sera plus simplement responsable de déclencher l'adaptation ; il pourra également décider de négocier des changements dans l'environnement d'exécution. Dans ce contexte, nous pensons que la notion de contrat de qualité de service est primordiale. Cette notion offre un cadre uniforme d'interaction entre le composant et son environnement d'exécution. Combinée avec l'hypothèse d'honnêteté des composants, elle offre une description fiable de l'environnement d'exécution au mécanisme de prise de décision. Cette hypothèse pourrait être approchée par asservissement des ports des composants ; les violations de cette hypothèse gérées grâce à des mécanismes de type tolérance aux fautes.
Nous étudions également les techniques de tissage d'aspects [11]. En effet, nous pensons que la programmation orientée aspects peut profiter à l'adaptation dynamique à deux niveaux. D'une part, l'adaptation dynamique peut être vue comme une fonctionnalité ajoutée statiquement à un programme. Nous nous intéressons donc à déterminer comment l'ajout de cette fonctionnalité peut s'exprimer en termes d'aspects. L'objectif est d'obtenir une solution semi-automatique des modifications nécessaires pour rendre un programme adaptable : les points seraient placés manuellement ; les intéractions avec l'exécutif d'adaptation insérées automatiquement par une collection d'aspects. D'autre part, les réactions mettent en œuvre des modifications dynamiques du programme. Nous étudions donc comment ces réactions peuvent être mise en œuvre à l'aide des deux primitives « tisser » et « détisser » dynamiquement un aspect que μDyner [16] et JAC [14] proposent.

A terme, nous serons en mesure de développer un cadre complet pour le développement de composants logiciels capables de s'adapter dynamiquement aux changements de leurs environnements d'exécution.

Bibliographie

1. Marco ALDINUCCI, Sonia CAMPA, Massimo COPPOLA, Marco DANELUTTO, Domenico LAFORENZA, Diego PUPPIN, Luca SCARPONI, Marco VANNESCHI, et Corrado ZOCCOLO. « Components for high performance grid programming in the Grid.It project ». Dans *Workshop on Component Models andd Systems for Grid Applications*, juin 2004.
2. Jérémy BUISSON, Françoise ANDRÉ, et Jean-Louis PAZAT. « Adaptation dynamique de codes parallèles ». Dans *Journées Composants 2004*, mars 2004.
3. Gilbert CABILLIC. « *Exécution d'applications parallèles sur architectures distribuées hétérogènes : propositions et mise en œuvre* ». PhD thesis, Université de Rennes 1, octobre 1996.
4. João W. CANGUSSU, Kendra COOPER, et Changcheng LI. « A Control Theory Based Framework for Dynamic Adaptable Systems ». Dans *2004 ACM Symposium on Applied Computing*, pages 1546–1553, mars 2004.
5. Djalel CHEFROUR et Françoise ANDRÉ. « Auto-adaptation de composants ACEEL coopérants ». Dans *3ème Conférence Française sur les Systèmes d'Exploitation (CFSE'3)*, octobre 2003.
6. Djalel CHEFROUR et Françoise ANDRÉ. « Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif Aceel ». Dans *Langages et Modèles à Objets LMO'03. Actes publiés dans la Revue STI*, volume 9 de *L'objet*, février 2003.
7. Jack DONGARRA et Victor EIJKHOUT. « Self-adapting Numerical Software for next generation application », août 2002.
8. Brian ENSINK et Vikram ADVE. « Coordinating Adaptations in Distributed Systems ». Dans *24th International Conference on Distributed Computing Systems*, pages 446–455, mars 2004.
9. Brian ENSINK, Joel STANLEY, et Vikram ADVE. « Program Control Languae : A Programming Language for Adaptive Distributed Applications ». accepted but not yet printed in Journal of Parallel and Distributed Computing, mai 2002.
10. Ken KENNEDY, Mark MAZINA, John MELLOR-CRUMMEY, Keith COOPER, Linda TORCZON, Fran BERMAN, Andrew CHIEN, Holly DAIL, Otto SIEVERT, Dave ANGULO, Ian FOSTER, Dennis GANNON, Lennart JOHNSSON, Carl KESSELMAN, Ruth AYDT, Daniel REED, Jack DONGARRA, Sathish VADHIYAR, et Rich WOLSKI. « Toward a Framework for Preparing and Executing Adaptive Grid Programs ». Dans *Proceedings of NSF Next Generation Systems Program Workshop (IPDPS)*, avril 2002.
11. Gregor KICZALES, John LAMPING, Anurag MENDHEKAR, Chris MAEDA, Cristina LOPES, Jean-Marc LOINGTIER, et John IRWIN. « Aspect-Oriented Programming ». Dans Mehmet AKŞIT et Satoshi MATSUOKA, éditeurs, *Proceedins of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg and New-York, 1997. Springer-Verlag.
12. Malcolm MCILHAGGA, Ann LIGHT, et Ian WAKEMAN. « Towards a design methodology for adaptive applications ». Dans *Mobile Computing and Networking*, pages 133–144, mai 1998.
13. Daniel PAUL, Sudhakar YALAMANCHILI, Karsten SCHWAN, et Rakesh JHA. « Decision Models for Adaptive Resource Management in Multiprocessor Systems ». http ://www.htc.honeywell.com/projects/arm/, 1998.
14. Renaud PAWLAK, Lionel SEINTURIER, Laurence DUCHIEN, G. FLORIN, F. LEGOND-AUBRY, et L. MARTELLY. « JAC : An Aspect-based Distributed Dynamic Framework ». *Software Practise and Experience (SPE)*, 34(12) :1119–1148, octobre 2004.
15. Pierre-Guillaume RAVERDY, Hubert Le Van GONG, et Rodger LEA. « DART : a reflective middleware for adaptive applications ». Dans *OOPSLA'98 Workshop #13 : Reflective programming in C++ and Java*, octobre 1998.
16. Marc SÉGURA-DEVILLECHAISE et Jean-Mard MENAUD. « μDyner : Un noyau efficace pour le tissage dynamique d'aspects sur processus natif en cours d'exécution ». Dans *LMO 2003*, pages 119–133. Hermès, février 2003.