Composition générique d'algorithmes d'exclusion mutuelle pour les grilles de calcul

Julien Sopena, Fabrice Legond-Aubry, Luciana Arantes, et Pierre Sens
email : [julien.sopena,fabrice.legond-aubry,luciana.arantes,pierre.sens]@lip6.fr

LIP6 - Université de Paris 6 - INRIA Rocquencourt
4, Place Jussieu 75252 Paris Cedex 05, France.

Résumé
Nous proposons une approche générique pour composer des algorithmes d'exclusion mutuelle dans le but de déployer des applications sur des grilles de calculs composés de fédérations de clusters. Ceci nous permet de pallier des limitations de certains algorithmes classiques en prenant en compte l'hétérogénéité des latences de communications. Nous proposons pour cela de déployer différentes instances d'algorithmes classiques pour chaque niveau de composition. Des mesures de performances effectuées sur Grid'5000 ont prouvé la capacité de passage à l'échelle de cette solution. Elles nous ont aussi permet de déterminer des couples optimum d'algorithmes à déployer en fonction du type d'application à exécuter et de la structure de la grille.

Mots-clés : Algorithme, exclusion mutuelle distribuée, grille, mesures de performance, composition générique

1. Introduction

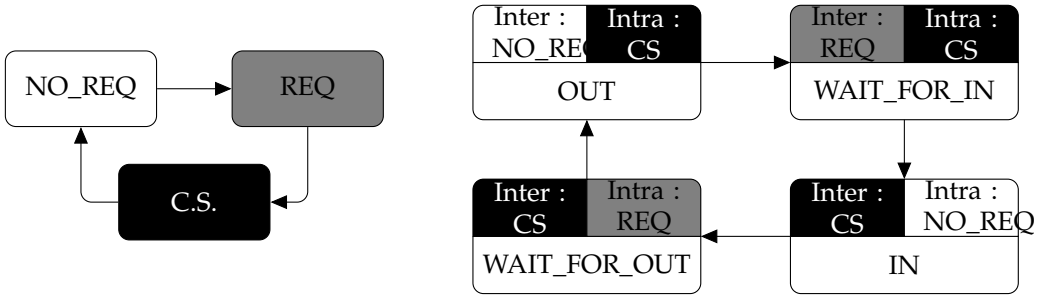
Ces dernières années ont vu l'émergence des Grilles de calculs. Ces structures informatiques de grandes tailles font coopérer un grand nombre de machines regroupées dans des grappes (cluster). Ainsi, en agrégeant les ressources des milliers d'ordinateurs, les grilles offrent aux applications de nouvelles opportunités en termes de puissance de calcul, de capacité de stockage et de distribution des programmes.
Les communications au sein d'une grille sont intrinsèquement hétérogènes. Les latences de communication sont beaucoup plus faibles entre nœuds d'un même cluster et il est important de prendre en compte cette propriété dans les applications réparties.
Ces applications reposent souvent sur des mécanismes d'exclusion mutuelle répartie afin de protéger l'accès à des ressources partagées dans des sections critiques (SC). Le but de cet article est d'étudier et comparer des algorithmes d'exclusion dans le contexte des grilles. Nous nous concentrons plus particulièrement sur l'impact du nombre de cluster dans grille et étudions différentes compositions d'algorithmes hiérarchiques.
On distingue généralement deux classes d'algorithmes d'exclusion mutuelle. Dans les approches à base de permission [5, 14, 7], un nœud peut entrer en SC uniquement si il a obtenu la permission de tous les nœuds (ou d'une majorité d'entre eux [7]). Dans les autres approches un jeton unique donne le droit d'accès à la ressource critique [17, 13, 11, 8]. Les algorithmes à jeton se distinguent sur la façon dont les requêtes sont transmises. Ils définissent ainsi des topologies logiques sur lesquelles circulent les requêtes de SC. Nous nous concentrons ici sur les algorithmes à base de jeton car ils sont souvent moins coûteux en nombre de messages et sont intrinsèquement plus extensibles que les algorithmes à permission. Malheureusement, ces

algorithmes ne prennent pas en compte l'hétérogénéité des communications pour réduire les latences de transmissions des requêtes ou du jeton.
Nous proposons donc une approche générique basée sur la composition d'algorithmes à jeton. Notre architecture permet de composer facilement des algorithmes à deux niveaux : en inter et intra grappe. Nous évaluons et comparons différentes compositions. Ces études ont été réalisées sur les neuf sites de la plate-forme nationale française Grid'5000. Nous montrons ainsi le gain apporté par la hiérarchisation et l'impact des choix des algorithmes sur les performances. Nous avons choisi de composer trois algorithmes, Martin, Naimi-Tréhel et Suzuki-Kazami qui se distinguent par des topologies logiques différentes, respectivement un anneau, un arbre et un graphe complet.
D'autres auteurs ont proposé des approches permettant de faire des hiérarchies. Par exemple, Mueller [9] introduit des priorités dans l'algorithme de Naimi-Tréhel permettant de changer la façon dont le jeton circule. Bertier et al. [1] adopte une approche similaires afin de satisfaire plusieurs requêtes intra-grappe avant de traiter des requêtes issues de grappes distantes. Certains auteurs ont également proposé des algorithmes de composition. Par exemple, Housni et al. [4] et Chang et al. [2] proposent des rassembler les nœuds en groupes logiques. Dans les deux cas un algorithme différent est utilisé en inter et intra-groupe. Ainsi, dans [4] l'algorithme de Raymond basé sur un arbre est utilisé au sein d'un groupe tandis que l'algorithme à permission de Ricard et Agrawala est utilisé entre les groupes. Chang et al. [2] et Omara et al. [12] propose quant à eux des algorithmes à permission hybrides : l'algorithme de Singhal [15] est utilisé en local et celui de Maekawa [7] entre les groupes. Dans [6], les auteurs proposent également une architecture à deux niveaux avec un algorithme centralisé au niveau bas et Ricart-Agrawala au plus haut niveau. Enfin, Erciyes [3] présente une approche similaire à la notre fondée sur un anneau de grappes et adapte l'algorithme de Ricart-Agrawal à cette topologie.
Notre proposition est proche de ces algorithmes hybrides. Cependant, la plupart de ceux ci ne considèrent pas explicitement l'hétérogénéité des communications comme critère principale de regroupement de machines. De plus, nous avons une approche plus générique dans le sens où nous essayons de trouver une bonne combinaison d'algorithmes en fonction non seulement de la topologie cible réelle (une grille) mais également du comportement de l'application.
L'article est organisé de la façon suivante : la section 2 présente notre système de composition. Un exemple est décrit dans la section 3 tandis que les sections 4, 5 et 6 détaillent l'environnement de test, les performances obtenues en fonction de la nature de l'application et de la structure de la grille.

2. Approche par composition pour l'exclusion mutuelle

2.1. Algorithme générique de composition

Notre approche consiste à composer de façon hiérarchique et générique des algorithmes d'exclusion mutuelle.
Dans chaque cluster les nœuds participent à une instance d'un algorithme d'exclusion mutuelle que l'on nommera par la suite algorithme *intra-cluster*. L'ensemble de ces algorithmes locaux constitue la couche basse de notre architecture et chacun d'eux n'utilise pour communiquer que des messages *intra* circulant sur les réseaux locaux des clusters.
Pour inter-connecter les différentes instances de cette couche basse, on ajoute une instance transversale répondant au nom d'algorithme *inter-cluster*. Ne participe à cet algorithme qu'un unique représentant de chacun des clusters. Ces sites communiquent entre eux au travers d'un WAN en échangeant des messages dits *inter*.
Le choix de ces algorithmes est libre et rien n'empêche d'instancier des algorithmes différents. Mais pour des raisons de lisibilité nous supposerons ici que tous les algorithmes *intra-cluster* sont des instances d'un même algorithme qui, par contre, pourra être différent de l'algorithme



(a) Automate classique (b) Automate des coordinateurs

FIG. 1 – Automates d'exclusion mutuelle

inter-cluster. Une composition se définira donc comme le choix des deux algorithmes *intra* et *inter*.

De cette structure hiérarchique ressortent deux types de nœuds, les nœuds *applicatifs* qui exécutent le code de l'application et qui ne participent qu'à l'algorithme *intra-cluster* de leur cluster et des nœuds *coordinateurs* qui participent à la fois à l'algorithme *intra-cluster* de leur cluster et à l'algorithme *inter-cluster*. Ces derniers, dont le rôle s'apparente à celui d'un proxy, sont les seuls à avoir une vision de la grille extérieure à leur cluster. Cette vision se limite néanmoins aux autres coordinateurs de la grille.

Les nœuds coordinateurs ne font pas de requêtes d'entrée en section critique pour leur propre compte mais uniquement pour assurer la circulation du jeton entre les couches. Un algorithme de composition s'exécute dans chaque coordinateur pour régir l'envoi de ces requêtes.

A l'inverse les nœuds applicatifs n'ont qu'une vision locale du système et ignorent tout du mécanisme de composition. Ainsi lorsque l'application désire accéder à la section critique, le nœud appelle la fonction *CS_Request()* de l'algorithme *intra-cluster*. Et au sortir de sa section critique, il utilisera la fonction *CS_Release()* de ce même algorithme pour libérer le jeton local.

Cette architecture mettant en jeux de multiples instances d'algorithme d'exclusion mutuelle, plusieurs nœuds sont susceptibles de posséder un jeton et donc d'accéder à la section critique. Chaque cluster possède un jeton local, le jeton *intra*, tandis que les coordinateurs s'échangent un unique jeton global, le jeton *inter*. Assurer globalement la propriété de sûreté de l'exclusion mutuelle revient alors à garantir qu'à tout moment, un seul des nœuds applicatifs sur l'ensemble de la grille possède un jeton *intra*.

À cette propriété de sûreté doit s'ajouter un mécanisme assurant une vivacité globale : tout nœud applicatif de la grille voulant accéder à la section critique finira par recevoir le jeton *intra* de son cluster.

2.2. Automate des coordinateurs

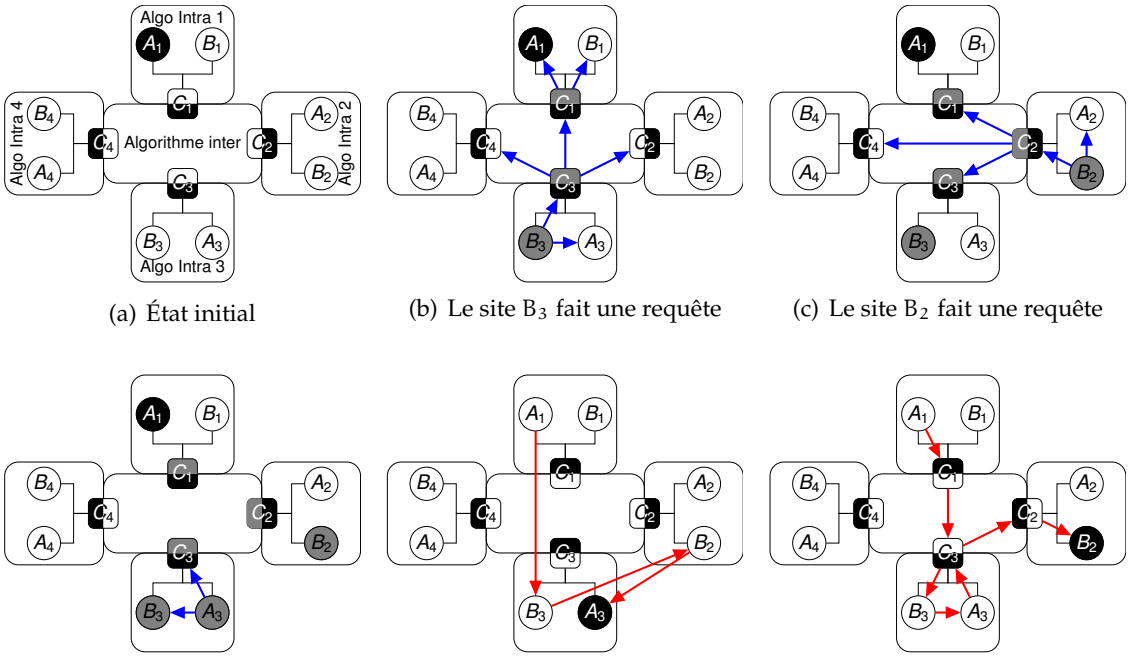
Comme on peut le voir sur la figure 1.(a), l'automate classique de l'exclusion mutuelle est composé de trois états : un état NO_REQ pour les sites ne demandant pas l'accès à la section critique, un état REQ pour les sites en attente d'une accès et enfin un état CS pour celui qui accède à la section critique.

Les *coordinateurs* exécutent deux instances d'algorithmes d'exclusion, les algorithmes *inter* et *intra*. Leur automate possède 4 états globaux OUT, IN, WAIT_FOR_OUT, WAIT_FOR_IN. Chacun de ces états est caractérisés par un état vis-à-vis de l'algorithme *inter* et un état vis-à-vis de l'algorithme *intra*.

Un coordinateur se trouve dans l'état OUT si aucun nœud applicatif de son cluster n'est intéressé par la section critique. Dans ce cas, le coordinateur possède le jeton *intra*. Il se trouve donc dans l'état CS pour l'instance *intra*. Par ailleurs ne demandant pas le jeton de l'instance *inter*, il

se trouve dans l'état NO_REQ de cet algorithme.
Un coordinateur passe de l'état OUT à l'état WAIT_FOR_IN dès qu'il reçoit une requête de l'un de ses nœuds applicatifs. Pour relâcher son jeton *intra* et donc satisfaire la requête, le coordinateur doit récupérer celui de l'algorithme *inter*. Il émet donc une requête *inter* (état *inter* = REQ) tout en conservant le jeton *intra* (état *intra* CS).
Le coordinateur passe à l'état IN lorsqu'il reçoit le jeton *inter*. Il peut dès lors libérer son jeton *intra* (état *intra* NO_REQ) tout en conservant celui de l'algorithme *inter* qu'il vient d'acquérir (état *inter* CS).
Enfin, l'état WAIT_FOR_OUT correspond à celui où un coordinateur reçoit une requête d'un autre coordinateur alors qu'il possède le jeton *inter*. Dans ce cas, il envoie une requête *intra* (état *intra* = REQ) et demeure dans l'état CS de l'algorithme *inter* jusqu'à l'arrivée du jeton *intra*. Ce n'est qu'après avoir reçu cette garantie de ne plus avoir de sites applicatifs en section critique dans son cluster qu'il libérera le jeton *inter* pour retourner dans l'état OUT.
Initialement tous les coordinateurs sont dans l'état OUT sauf un qui est dans l'état IN.

3. Exemple



(a) État initial (b) Le site B_3 fait une requête (c) Le site B_2 fait une requête

(d) Le site A_3 fait une requête (e) Accès dans l'ordre des requêtes (f) Préemption naturelle

FIG. 2 – Exemple

Pour mieux comprendre comment ce mécanisme de composition permet de limiter la complexité en nombre de messages et améliore les temps d'accès à la section, intéressons nous à l'exemple suivant (figures 2). On considère ici une grille composée de 4 clusters eux même composés de 3 nœuds. Ainsi, dans chaque cluster i on aura 2 nœuds applicatifs A_i et B_i auxquels s'ajoutera 1 nœud coordinateur C_i. Pour simplifier l'étude de la composition on choisit ici d'instancier sur les 2 niveaux un algorithme simple basé sur la diffusion. Comme pour l'algorithme de Suzuki et Kasami [17], un site voulant entrer en section critique diffuse une requête à tous les participants du système et lorsqu'un site quitte sa section critique il envoie le jeton au premier nœudlui lui ayant demandé le jeton.

Comme on peut le voir sur la figure 2(a) à l'état initial le nœud A_1 possède le jeton local du cluster 1 et son coordinateur celui de l'algorithme *inter*. Tous les autres jetons locaux étant détenus par les autres coordinateurs (C_2, C_3, C_4).
Dans un premier temps, le nœud B_3 veut entrer en section critique. Il diffuse sa requête *intra* dans son cluster (figure 2(b)). Elle est ensuite "forwarder" dans l'algorithme *inter* par son coordinateur qui passe dans l'état WAIT_FOR_IN. Seul le coordinateur C_1 rediffuse la requête dans son cluster. On peut voir ici un premier gain en terme de nombre de message : non seulement on se limite à 3 messages *inter* du fait de la diffusion hiérarchique, mais on évite aussi de rediffuser la requête dans les clusters 3 et 4. En effet comme les coordinateurs C_2 et C_4 sont dans l'état OUT, aucun des nœuds applicatifs de leur cluster ne peut être en section critique.
Après le nœud B_3, le nœud B_2 diffuse une requête *intra* dans le cluster 2. Comme le montre la figure 2(c), la composition économise toujours la rediffusion dans les clusters n'exécutant pas la section critique mais elle évite aussi de diffuser une requête dans le cluster 1 dont le coordinateur est déjà dans l'état WAIT_FOR_OUT.
Enfin si une deuxième requête a lieu dans le cluster 3, on peut observer sur la figure 2(d) qu'il n'y a plus de rediffusion au niveau *inter*. En effet le coordinateur C_3 qui est déjà dans l'état WAIT_FOR_IN n'a pas besoin de renouveler sa requête au niveau *inter*. L'algorithme de composition agrège donc naturellement les requêtes locales limitant ainsi le nombre de message *inter*.
On a maintenant trois requêtes concurrentes dans la grille. Si l'algorithme d'exclusion mutuelle est équitable, il satisfera ces dernières dans leur ordre d'émission. Après le nœud A_1, les nœuds B_3, B_2 et A_3 accéderont successivement à la section critique, comme le montre la figure 2(e). Ainsi le jeton passera du cluster 1 au cluster 3, puis du cluster 3 au cluster 2, pour finir par revenir dans le cluster 3. Ces aller-retour entre les cluster 2 et 3 augmentent considérablement la latence d'accès à la section critique. En effet, contrairement au temps de propagation des requêtes qui peut être partiellement (parfois même totalement) recouvert par des exécutions de sections critiques, pendant les transmissions de jeton aucun nœud ne peut accéder à la section critique. Minimiser le coût de ces transmissions a donc un impact direct sur la latence.
Dans le cas de notre algorithme de composition on peut observer, sur la figure 2(f), un réordonnancement de l'accès à la section critique. Ce nouvel ordre d'accès à la section critique est induit naturellement par le mécanisme de composition et favorise les accès locaux (dans un même cluster).
En effet au sortir de la section critique du nœud A_1, C_1 récupère le jeton *intra* du cluster 1 et transmet le jeton *inter* qu'il détenait jusque là au coordinateur C_3. Ce dernier peut alors envoyer le jeton *intra* du cluster 3 au nœud B_3 passant ainsi de l'état WAIT_FOR_IN à l'état IN. Mais dans le même temps, du fait de la requête pendante du nœud B_2, il diffuse une requête *intra* et passe dans l'état WAIT_FOR_OUT. Donc lorsque le nœud B_3 termine d'exécuter son code concurrent, il reste deux requêtes pendantes dans le cluster 3 : celle du nœud A_3 et celle du coordinateur C_3. L'ordre d'accès au jeton dans une même instance d'algorithme respectant celui des émissions des requêtes, B_3 envoie le jeton au nœud A_3. Finalement, B_2 accédera à la section critique via les retransmissions par les coordinateurs C_3 et C_2.

4. La grille de test

Cette section présente une étude de la composition et de ses effets. Cette évaluation a été réalisée sur la plate-forme nationale de tests Grid'5000 [1]. Grid'5000 regroupe 17 clusters répartis dans 9 villes françaises. Dans cette étude nous avons utilisé un cluster dans chacune de ces villes.

[1] Grid'5000 est une plate-forme de test financée par le ministère de la recherche français, l'INRIA, le CNRS et les régions. Voir https :// www.grid5000.fr

Les machines ayant servies aux tests sont toutes équipées de processeur Bi-Opteron et de 2Go de RAM. Les villes sont reliées entres elles par des liens dédiés en fibre optique. Des mesures répétées ont montré une latence de connexions *intra* cluster (i.e., entre les nœuds d'un même cluster) de l'ordre de 0,05 ms, ainsi qu'une la latence de connexions *inter* cluster variant de 6 à 20 ms suivant les villes (voir figure 3). Ces mesures pratiquement constantes font ressortir un rapport de l'ordre de 10^2 entre les latences *inter*-cluster et *intra*-cluster.

de \ à	orsay	grenoble	lyon	rennes	lille	nancy	toulouse	sophia	bordeaux
orsay	0.034	15.039	9.128	8.881	4.489	95.282	15.556	20.239	7.900
grenoble	14.976	0.066	3.293	15.269	12.954	13.246	10.582	9.904	16.288
lyon	9.136	3.309	0.026	12.672	10.377	10.634	7.956	7.289	10.078
rennes	8.913	15.258	12.617	0.059	11.269	11.654	19.911	19.224	8.114
lille	10.000	10.001	10.001	10.001	0.001	10.001	20.000	20.001	10.001
nancy	5.657	13.279	10.623	11.679	9.228	0.032	98.398	17.215	12.827
toulouse	15.547	10.586	7.934	19.888	19.102	17.886	0.043	14.540	3.131
sophia	20.332	9.889	7.254	19.215	16.811	17.238	14.529	0.051	10.629
bordeaux	7.925	16.338	10.043	8.129	10.845	12.795	3.150	10.640	0.045

FIG. 3 – Latences moyenne de RTT sur Grid'5000

Durant nos tests, nous avons fait tourner des processus qui ont exécuté 100 demandes de sections critiques. Chacune de ces sections critiques avaient une durée moyenne de 10 ms (ce qui est du même ordre de grandeur que le délai d'un envoi de paquet entre deux clusters). Chaque expérience a été répétée 10 fois, et nous présenterons ici une moyenne des résultats.
Vis-à-vis de l'exclusion mutuelle, une application se caractérise par :
- α : le temps moyen nécessaire pour exécuter une section critique (SC).
- β : le délai moyen entre la fin d'une SC et une nouvelle demande issue du même nœud.
- ρ : le ratio β/α qui exprime la fréquence d'accès à la section critique par un nœud.

Nous avons réalisé nos expériences avec des plusieurs types d'applications :
- **Parallélisme faible** : $\rho \leq N$: L'exclusion mutuelle est fortement sollicité. La majorité des nœuds applicatifs demandent la SC. Presque tous les coordinateurs sont en attente du jeton *inter*.
- **Parallélisme intermédiaire** : $N < \rho \leq 3N$: Seulement quelques nœuds applicatifs sont en compétition pour obtenir la SC. Par conséquent seulement une partie des coordinateurs ont fait une demande pour le jeton *inter*.
- **Parallélisme fort** : $3N \leq \rho$: Si le parallélisme est important dans l'application, les demandes de SC sont éparses. L'exclusion mutuelle est peu sollicitée. Le nombre total de requêtes sur l'ensemble des nœuds applicatifs est peu conséquent et le nombre de coordinateurs demandant le jeton *inter* est donc faible

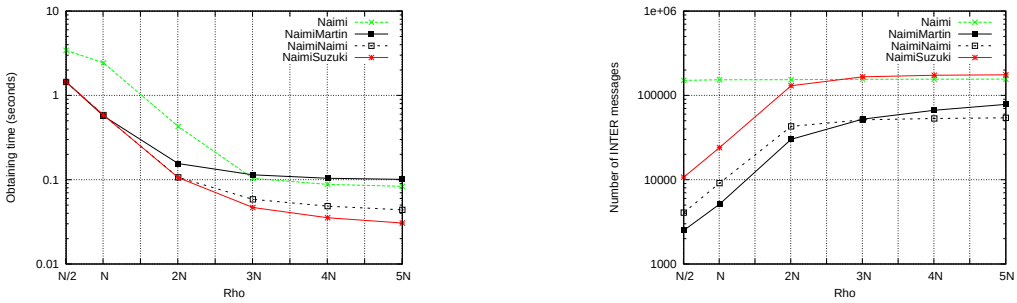
5. Impact de la composition

Le but de cette première partie de notre étude était de vérifier le gain apporté par la composition en fonction du type d'applications. Une partie de ces résultats ont été publiés à ICPP 2007 [16]. Le nombre de nœuds applicatifs N a été fixé à 180 répartis sur 180 processeurs distincts.
Tout d'abord, nous insistons sur le fait que le choix de l'algorithme *intra* a influence très faible sur les performances globales si les clusters restent de taille raisonnable. Pour cela, nous avons fixé l'algorithme *inter* et conduit différentes expériences qui ont toutes montrées un compor-

tement identique face aux différentes classes d'applications. En revanche, le choix des algorithmes *inter* change significativement le comportement global.
Nous avons choisi de composer 3 algorithmes utilisant 3 structures logiques différentes pour acheminer les requêtes.
Dans l'algorithme de Martin [**?**] les nœuds sont structurés en **anneau logique**. Les requêtes sont transmises autour de cet anneau jusqu'au possesseur du jeton. Le jeton suit le chemin inverse de la requête.
L'algorithme de Naimi-Tréhel [10] utilise un **arbre dynamique** pour acheminer les requêtes. Chaque requête étant acheminée jusqu'à la racine de l'arbre, le nombre moyen de requêtes transmise par SC est logarithmique. A la fin d'une SC, le jeton est directement transmis au premier nœud ayant fait une requête.
Enfin, comme dans l'exemple de la section [17], l'algorithme de Suzuki-Kasami diffuse chaque requête de SC à tous les nœuds, tandis que le jeton est directement transmis au prochain demandeur de SC.
Dans la suite de l'article nous noterons Naimi pour l'algorithme Naimi-Tréhel et Suzuki pour l'algorithme Suzuki-Kasami. Nous noterons la composition sous la forme "*Algorithme Intra-Algorithme Inter*" (ex : Naimi-Suzuki).
Pour évaluer la composition des algorithmes, nous avons mesuré :
– le temps moyen pour obtenir le jeton - i.e., le temps entre le moment de la demande par un nœud applicatif et le moment où le nœud reçoit le effectivement le jeton,
– le nombre de messages *inter* échangés.

5.1. Délai d'obtention

Pour expliquer les différences de *délai d'obtention* du jeton, nous utilisons les notations suivantes :
- T : le délai moyen de transmission d'un message entre 2 coordinateurs,
- T_{req} : le délai moyen pour acheminer la **demande de jeton** *inter* du coordinateur possédant le jeton au coordinateur qui va donner la permission à son nœud applicatif.
- T_{token} : le délai moyen pour acheminer le **jeton** *inter* du coordinateur possédant le jeton au coordinateur qui va donner la permission à son nœud applicatif.
- T_{pendCS} : le temps moyen pour satisfaire toutes les demandes de jeton *inter* avant de satisfaire la requête demandée. Ce temps augmente avec le nombre de requêtes en attente.



(a) Délai moyen d'obtention du jeton (b) Nombre total de messages *inter*

FIG. 4 – Performances des différentes compositions

Avec une application **faiblement parallèles** ($\rho \leq N$), il y a de nombreux accès à la SC, le coordinateur qui réclame le jeton doit attendre que toutes les requêtes de SC *inter* déjà en attente soient satisfaites. Le délai T_{pendCS} est, dans ce cas, plus grand que celui nécessaire pour de-

mander le jeton (T_{req}) et le recouvre complètement. Par conséquent le *délai d'obtention* est donc uniquement égal à T_{pendCS}+T_{token}.
Cela explique pourquoi le *délai d'obtention* est plus important pour des $\rho \leq N$, car il y a souvent de nombreux nœuds applicatifs (et donc de nombreux coordinateurs) en état REQ, alors qu'il y en a assez peu lorsque $\rho \geq 3N$. Ce comportement est illustré sur les courbes de la figure 4.(a). La file d'attente de la SC constitue un goulôt d'étranglement.
Nous observons donc peu de différences entre le temps moyens des différents algorithmes car le temps de demande du jeton (T_{req}) est recouvert par le temps nécessaires pour traiter les demandes de SC des nœuds applicatifs (T_{pendCS}). Les différences ne pourraient être dues qu'au temps de livraison du jeton (T_{token}). Mais on constate qu'il ne faut qu'un message de délai T pour Naimi et Suzuki. Pour Martin, c'est aussi presque toujours un seul message puisque la probabilité que le prédécesseur dans l'anneau du coordinateur possédant le jeton soit à l'état REQ est très proche de 1. Le prédécesseur va donc préempter le jeton avant de le faire suivre.
Enfin, comme la concurrence est forte entre coordinateurs pour accéder à la SC *inter*, le *délai d'obtention* reste très régulier et varie peu suivant la composition.
Concernant les applications avec un degré de **parallélisme moyen**, nous pouvons remarquer un comportement presque identique pour les compositions Naimi-Naimi et Naimi-Suzuki et un coût plus important pour Naimi-Martin. Ceci est dû au fait que pour Naimi-Martin, seuls certains coordinateurs sont à l'état REQ et que par conséquent il faut plusieurs transmissions successives de messages pour acheminer le jeton à un coordinateur à l'état REQ.
Dans le cas d'applications avec un degré de **parallélisme fort**, lorsqu'il y a peu de concurrence, le *délai d'obtention* du jeton *inter* ($\rho \geq 3N$) pour un coordinateur comprend le délai pour acheminer une demande de SC (T_{req}), et le délai pour récupérer le jeton (T_{token}) car la file de requêtes *inter* est pratiquement tout le temps vide et, par conséquent, T_{pendCS} est très souvent nul.
Avec Suzuki, T_{req} est une diffusion ce qui engendre un délai égal à T ; avec Naimi qui utilise un arbre, il est en $\mathcal{O}(\log(N)) * T$; avec Martin - qui est le moins efficace - il faut, en moyenne $N/2 * T$ à cause de $N/2$ messages successifs pour acheminer la requête sur l'anneau.
Pour Naimi-Naimi et Naimi-Suzuki, le délai d'acheminement du jeton (T_{token}) reste égal à celui des applications faiblement parallèle soit T. Par contre pour Naimi-Martin, il faut encore, en moyenne, un délai de $(N/2) * T$ pour atteindre le premier coordinateur demandeur du jeton.

Composition \ Parallélisme	Faible	Moyen	Fort
Naimi-Suzuki	$T_{pendCS} + T$	$T_{pendCS} + T$	$T + T$
Naimi-Martin	$T_{pendCS} + T$	$T_{pendCS} + (N/2) * T$	$(N/2) * T + (N/2) * T$
Naimi-Naimi	$T_{pendCS} + T$	$T_{pendCS} + T$	$\log(N) * T + T$

FIG. 5 – Délai moyen estimé d'obtention du jeton pour chaque composition

En terme d'efficacité concernant les *délais d'obtention* des algorithmes pour $\rho > 3N$, Suzuki est le plus performant. Il ne consomme qu'une diffusion (T) et qu'un envoi de message (T). Puis vient ensuite Naimi qui est en $(\log(N) + 1) * T$ et enfin Martin qui sera d'ordre $N * T$
La tableau 5.1 résume notre étude.

5.2. Nombre de messages *inter*-cluster

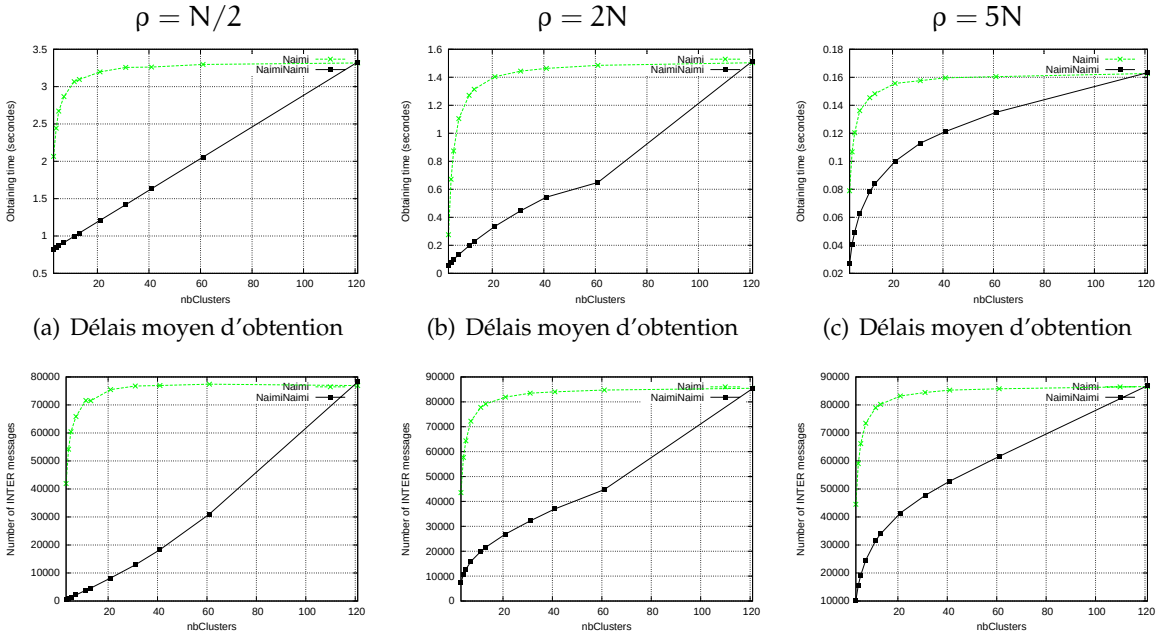
Lorsque ρ est petit, il y a beaucoup de concurrence d'accès à la SC de la part des nœuds applicatifs. Par conséquent, la propriété d'agrégation exposée dans la section 3 est très forte car toutes

les requêtes des nœuds applicatifs d'un même cluster ne vont générer qu'une seule requête du coordinateur du cluster.
Dans ce cas, il est très avantageux d'utiliser la composition par rapport aux algorithmes à plat qui génèrent un traffic plus intense et aléatoirement réparti entre les clusters. Ce traffic provient de la non correspondence entre les topologies logiques imposées par les algorithmes et les topologies physiques du réseau. Cependant, lorsque la concurrence d'accès à la SC diminue - avec l'augmentation de ρ, cet effet d'agrégation des requêtes *intra* au niveau du coordinateur diminue.
Pour les compositions avec Suzuki et Naimi en algorithme *inter*, le nombre de messages *inter* échangés entre les coordinateurs sont respectivement de N et $\mathcal{O}(\log(N))$ messages pour effectuer la requête et de 1 message pour acheminer le jeton. Donc Naimi reste le plus intéressant des deux. Pour la composition avec Martin, le nombre de messages dépend de N et du nombre de requêtes SC *intra* (et donc ρ). Pour des applications à faible parallélisme (ρ petit), la probabilité que tous les coordinateurs soient en état REQ est forte. Plus ρ augmente plus cette probabilité se réduit et plus statistiquement, le nombre de coordinateurs en état "NO_REQ" augmente. Le nombre de messages successifs nécessaires pour acheminer le jeton à traver l'anneau passe donc, en moyenne, de 1 à $N/2$.
Dans la figure 4(b), nous pouvons constater que indépendamment de ρ, l'algorithme Naimi à plat génère toujours un nombre important de messages *inter* cluster. Un message sera dont arbitrairement acheminé à travers de nœuds situées soit dans le même cluster soit hors de ce dernier. Par exemple, dans le cas d'un envoi de jeton entre deux éléments d'un même cluster, il se peut que le jeton passe par une série de nœuds extérieurs ce qui augmentera significativement le délai d'acheminement. A l'opposé, pour les compositions, l'effet d'agrégation génère un seul message pour plusieurs requêtes *intra* des nœuds applicatifs d'un même cluster d'où un nombre de messages *inter* bien inférieurs pour les compositions.

6. Impact de la structure de la grille

Après avoir étudié le choix de la composition en fonction du type d'applications, nous nous intéressons maintenant à l'impact du partitionnement de la grille sur les algorithmes d'exclusion mutuelle en général et sur notre algorithme de composition en particulier. Pour ce faire, nous allons réaliser des expériences en faisant varier le nombre de clusters de la grille tout en conservant le même nombre de processeurs. Nous avons fixé le nombre de processeurs à 120 et étudions des configurations de 2, 3, 4, 6, 8, 12, 20, 30, 40, 60 et 120 clusters.
La grille expérimentale Grid'5000, utilisée lors de l'étude de la composition étant limitée à 9 sites, nous avons émulé les différentes topologies réseaux et la structure de nos grilles sur un cluster possédant des nœuds routeurs "dummynet" utilisés pour injecter des latences entre les clusters virtuels. Le cluster est composé de 24 machines Bi-Xeon 2,8Ghz avec 2GO de RAM et d'une machine dummynet qui est une machine P4 2Ghz spécialement dédie à l'émulation des latences de connexions. L'ensemble de ces machines sont connectées par ethernet gigabit via un switch d'une capacité de 148 Gbps. Afin de valider la méthode, nous avons reproduit certaines expériences de la section 5. Les résultats obtenus se sont montrés quasiment identiques à ceux mesurés sur Grid'5000.
Pour se focaliser sur l'impact du partitionnement, nous avons choisi de n'étudier ici qu'une seule composition. Après les résultats obtenus dans la section 5, notre choix s'est porté sur la composition "Naimi-Naimi". Elle s'est en effet montrée la plus polyvalente, avec une latence optimum pour les applications faiblement et moyennement parallèles ainsi qu'une bonne latence pour les applications ayant un fort degré de parallélisme. De plus sa faible complexité en message permettra d'étudier les cas extrêmes (120 clusters).
La section 5, ayant montré le fort impact du niveau de parallélisme sur les performances, nous

avons choisi d'intégrer ce paramètre dans notre étude des effets du clustering. Les figures 6(a) et 6(d) correspondent à une application dont le niveau de parallélisme est relativement faible ($\rho = N/2$). Les figures 6(b) et 6(e) représentent les résultats pour une application moyennement parallèle ($\rho = 2N$). Tandis que les figures 6(c) et 6(f) ont été réalisées avec une application hautement parallèle ($\rho = 5N$).
Pour chaque expérience, nous avons mesuré : le temps moyen d'attente moyen (en seconde) avant d'obtenir la section critique (figures 6(a), 6(b) et 6(c)) et le nombre total de messages *inter* (figures 6(d), 6(e) et 6(f)).
Dans cette série d'expériences, chaque nœud exécute 100 sections critiques. Nous effectuons les mesures en régimes stationnaire ainsi les 1000 premiers accès et les 1000 derniers ne sont pas mesurés. Les délais entre deux requêtes et les temps passés en section critique suivent une distribution de Poisson dont les moyennes sont fixées respectivement par le paramètre ρ et à 10ms.



(a) Délais moyen d'obtention (b) Délais moyen d'obtention (c) Délais moyen d'obtention

(d) Nombre de messages *inter* (e) Nombre de messages *inter* (f) Nombre de messages *inter*

FIG. 6 – Impact du partitionnement de la grille sur la composition

6.1. Impact sur l'algorithme à plat

Commençons par étudier l'impact de la structure de la grille sur l'algorithme de Naimi-Tréhel à "plat".
En regardant globalement les 6 graphiques de la figure 6, on tout d'abord peut remarquer le comportement très similaire de l'ensemble des courbes correspondant à cet algorithme. En effet, la latence moyenne ainsi que le nombre de messages *inter* augmentent très rapidement lorsque le nombre de clusters varie de 2 à 12, et ce, quelle que soit la valeur de ρ. Ensuite ces augmentations diminuent fortement pour devenir négligeables passé 40 clusters.
Pour expliquer cette allure hyperbolique des courbes, il nous faut étudier la fréquence d'utilisation du réseau WAN par un algorithme d'exclusion mutuelle global. Soit $\mathcal{P}$ la probabilité que le destinataire d'un message n'appartienne pas au même cluster que l'émetteur. On considère

pour cela une grille de n nœuds se répartissant uniformément dans c clusters. Pour simplifier les calculs, on supposera qu'un site peut s'envoyer un message à lui même (modélise 2 accès successifs à la section critique). On obtient alors :

$$\mathcal{P} = \frac{n - \frac{n}{c}}{n} = 1 - \frac{1}{c}$$

Cette formule explique bien l'allure des courbes de la figure 6. Elle permet aussi de montrer que ce résultat ne dépend pas du nombre de machines si elles sont uniformément réparties dans la grille.
En comparant les mesures de l'algorithme de Naimi dans les figures 6(a), 6(b) et 6(c), on observe que le degré de parallélisme des applications a une incidence sur le délai moyen d'obtention de la section critique. A l'inverse, les courbes figures 6(d), 6(e) et 6(f) mesurant le nombre de messages *inter* envoyés, montre que le parallélisme n'a presque pas d'impact sur la complexité. On observe néanmoins une légère diminution du nombre de messages dans le cas d'un algorithme faiblement parallélisé.

6.2. Impact sur l'approche par composition

D'une manière générale, on peut voir sur l'ensemble courbes de la figure 6 que le partitionnement à aussi un impact sur l'algorithme de composition : la latence et le nombre de messages augmentent avec le nombre de clusters.
Mais si l'on exclut le cas particulier de 120 clusters constitués d'un seul d'un nœud applicatif, où il n'y a pas vraiment de composition, notre approche est toujours plus rapide et moins coûteuse en messages *inter* que l'algorithme de Naimi-Trehel à "plat". Ces gains restant encore significatifs pour une grille composée de 60 clusters de deux nœuds applicatifs.
D'autre part les augmentations sont beaucoup moins brutales que pour "Naimi" : l'approche par composition résiste mieux au partitionnement. Cette différence de croissance est très marquée pour les premières valeurs des courbes. On atteint alors, un écart maximal entre les deux courbes pour une grille de 12 clusters.
Comparons maintenant les résultats obtenus pour les différents types d'applications. Contrairement à ce que nous avions observé pour l'algorithme de Naimi-Trehel à "plat", on voit ici une nette différence entre l'allure des courbes suivant le niveau du parallélisme.
En effet la structure hiérarchique de composition a un effet concentrateur. Cet effet, lié au nombre de noeuds applicatifs par cluster, se démultiplie avec l'augmentation de la concurence d'accès à la section critique.
Ainsi, pour une application faiblement ou moyennement parallèle ($\rho = N/2$ et $\rho = 2N$) l'effet concentrateur de la composition est à son maximum. Et ce quelque soit le nombre de clusters. La latence et le nombre de messages deviennent alors proportionnels au taux de partitionnement, ce qui explique l'allure quasi linéaire des courbes des figures 6(a) et 6(d).
A contrario, pour une application fortement parallélisée ($\rho = 5N$), l'effet de concentration de la composition est plus faible et dépend directement du taux de partitionnement : plus le nombre de nœuds applicatifs est petit plus la probabilité d'avoir une autre requête dans son cluster local est faible.

7. Conclusions

Cet article présente un algorithme permettant de composer facilement différents algorithmes d'exclusion mutuelle afin de faciliter un passage à l'échelle dans les grilles de grappes de calculs. Cette approche permet de prendre en compte l'hétérogénéité des latences de communications : des communications locales de type LAN ayant une latence de l'ordre d'une dizaine de microsecondes et des communications de type WAN - pour les communications inter sites

sur de longues distances - de l'ordre de la dizaine de millisecondes. Une telle composition reste complètement transparente pour l'application et pour la plus part des algorithmes d'exclusion mutuelle classique.
L'aspect générique de notre mécanisme pourrait permettre sa mise en oeuvre sur de multiples niveaux. Il pourrait ainsi s'adapter à des grilles de calculs internationales incluant des latences intercontinentales de l'ordre de la centaine de millisecondes. Cela rend cette solution particulièrement adaptée aux systèmes très large échelle.
Du point de vue des performances, les résultats de nos expériences conduites d'abord sur une grille réelle puis émulées sur un cluster de calcul montrent que le degré du parallélisme de l'application a un impact significatif sur le choix de la composition.
Nous avons également étudié l'impact de la structure de la grille et montré que dans tous les cas la composition améliorait les performances par rapport à une approche non hiérarchique.

Bibliographie

1. Bertier (M.), Arantes (L.) et Sens (P.). – Distributed mutual exclusion algorithms for grid applications : A hierarchical approach. *JPDC*, vol. 66, 2006, pp. 128–144.
2. Chang (I.), Singhal (M.) et Liu (M.). – A hybrid approach to mutual exclusion for distributed system. *In : IEEE ICSAC*, pp. 289–294.
3. Erciyes (K.). – Distributed mutual exclusion algorithms on a ring of clusters. *In : ICCSA*, pp. 518–527.
4. Housni (A.) et Tréhel (M.). – Distributed mutual exclusion by groups based on token and permission. *In : ACM/IEEE ICCSA*, pp. 26–29.
5. Lamport (L.). – Time, clocks, and the ordering of events in a distributed system. *CACM*, vol. 21, n7, 1978, pp. 558–564.
6. Madhuram et Kumar. – A hybrid approach for mutual exclusion in distributed computing systems. *In : SPDP*.
7. Maekawa (M.). – A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM-TOCS*, vol. 3, n2, mai 1985, pp. 145–159.
8. Martin (Alain J.). – Distributed mutual exclusion on a ring of processes. *Sci. Comput. Program.*, vol. 5, n3, 1985, pp. 265–276.
9. Mueller (F.). – Prioritized token-based mutual exclusion for distributed systems. *In : IPPS/SPDP*, pp. 791–795.
10. Naimi (M.) et Trehel (M.). – An improvement of the log N distributed algorithm for mutual exclusion. pp. 371–377. – IEEE Computer Society Press.
11. Naimi (M.), Trehel (M.) et Arnold (A.). – A log (N) distributed mutual exclusion algorithm based on path reversal. *JPDC*, vol. 34, n1, 1996, pp. 1–13.
12. Omara (F.) et Nabil (M.). – A new hybrid algorithm for the mutual exclusion problem in the distributed systems. *IJICIS*, vol. 2, n2, 2002, pp. 94–105.
13. Raymond (K.). – A tree-based algorithm for distributed mutual exclusion. *ACM-TOCS*, vol. 7, n1, 1989, pp. 61–77.
14. Ricart (G.) et Agrawala (A.). – An optimal algorithm for mutual exclusion in computer networks. *CACM*, vol. 24, 1981.
15. Singhal (M.). – A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE TPDS*, vol. 3, n1, 1992, pp. 121–125.
16. Sopena (J.), Legond-Aubry (F.), Arantes (L.) et Sens (P.). – A composition approach to mutual exclusion algorithms for grid applications. *ICPP*, vol. 0, sep 2007, p. 65.
17. Suzuki (I.) et Kasami (T.). – A distributed mutual exclusion algorithm. *ACM-TOCS*, vol. 3, n4, 1985, pp. 344–349.