RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Politiques de gestion de protections pour l'implémentation de sections critiques.

Emmanuel Sifakis et Laurent Mounier

Emmanuel.Sifakis@imag.fr Laurent.Mounier@imag.fr
VERIMAG
Centre Equation, 2 av. de Vignate, 38610, Gières, France

Résumé

L'exploitation des architectures multicœurs repose sur une bonne gestion du parallélisme à l'intérieur des applications. Une telle gestion nécessite notamment d'éviter les conflits d'accès (ou *race conditions*) aux variables partagées entre plusieurs processus. Ceci peut se faire par l'introduction de *sections critiques* destinées à protéger les accès à ces variables. L'exécution atomique de ces sections est alors garantie soit par des mémoires transactionnelles, soit par des primitives de synchronisation de type *mutex*. Il apparait toutefois que, dans ce deuxième cas, ces verrous de synchronisation ne sont pas toujours utilisés de manière optimale par les applications. Nous proposons dans ce travail différentes politiques de gestion des verrous qui ont pour objectif d'améliorer les temps d'exécution. Nous présentons également une bibliothèque de gestion de verrous dédiée, qui permet l'acquisition simultanée de plusieurs objets de synchronisation, et qui offre de meilleures performances que des implémentations bien établies comme les *pthreads*. Ces résultats sont validés expérimentalement.

Mots-clés : programmation concurrente, section critique, mutex, inter-blocage

1. Introduction

L'introduction des architectures multiprocesseurs et multicœurs n'est pas sans influence sur la structure des applications logicielles. Dans un passé récent, l'amélioration des performances en temps d'exécution reposait pour une grande part sur l'avancé du matériel (processeurs plus rapides, accès à la mémoire plus efficace, etc.). Une partie importante du code des applications pouvait donc être conçu de manière séquentielle, les gains de performance étant assurés par ailleurs. Aujourd'hui, pour exploiter au mieux le parallélisme intrinsèque aux nouvelles plateformes matérielles, le modèle d'excution des applications doit être celui d'un ensemble d'activités asynchrones, s'exécutant de manière concurrente, et partageant ponctuellement des informations communes. Dans la pratique, un tel modèle d'exécution est encore bien souvent transposé directement au niveau des langages de programmation à travers les notions de processus légers (ou *threads*) et de variables partagées. Ce mode de programmation est notoirement difficile, en particulier en ce qui concerne le contrôle de l'accès au variables partagées afin d'éviter deux principaux écueils :

– les conflits d'accès (ou *race conditions*), lorsque plusieurs threads accèdent simultanément à une même variable ;
– les inter-blocages (*deadlocks*), lorsque plusieurs threads attendent mutuellement de se délivrer une autorisation pour poursuivre leur exécution.

Pour faciliter cette tâche, la communauté des langages de programmation a introduit le concept de *section atomique*, issu du domaine des bases de données. Leur objectif est de garantir que l'ensemble des instructions incluses dans une section critique apparaisse comme une instruction unique, exécutée de manière atomique par le processeur. Ceci implique notamment que la séquence des modifications intermédiaires apportées aux variables partagées lors de l'exécution d'une section atomique ne soit pas visible par les autres threads de l'application.

Différentes solutions ont été proposées pour implémenter ce concept de sections atomiques. Une première approche, dite "optimiste" (*optimistic concurrency*)[1, 2, 3] consiste à supposer *a priori* qu'il n'y aura pas de conflits : le thread qui souhaite exécuter une section critique l'exécute librement, et un journal des accès est tenu lors de cette exécution. S'il s'avère que des accès concurrents conflictuels ont eu lieu, alors l'ensemble des opérations effectuées dans la section critique est annulé, et celle-ci doit être ré-exécutée à nouveau (*roll-back*). Ce type d'approche peut être implémentée soit en utilisant des mécanismes matériels dédiés [4], soit des *mémoires transactionnelles logicielles (STM)* qui en simule l'effet [5, 6].

Dans les deux cas cette approche n'est pas satisfaisante lorsque le nombre de conflits potentiels est élevé, ou lorsque les opérations mises en jeu sont difficiles à annuler (comme des entrées-sorties).

Une seconde approche, dite "pessimiste" (*pessimistic concurrency*), consiste à protéger par défaut toute exécution d'une section critique en utilisant des *mécanismes de protection* (comme les *mutex*). Il s'agit de primitives de synchronisation qui permettent de conditionner l'exécution d'un thread à l'obtention de certaines ressources de contrôle, dans le cas contraire celui-ci est suspendu en attendant que ces ressources soient disponibles. Lorsqu'il est bien utilisé ce mécanisme assure donc une protection efficace d'une section critique, en garantissant notamment que tous les accès aux variables partagées inclus dans cette section se feront de manière exclusive par le thread autorisé à s'exécuter. Toutefois, cette approche demande également un certain nombre de précautions de la part du programmeur :

– les *mutex* doivent être introduits en nombre suffisant pour protéger l'ensemble des variables partagées ;
– ils ne doivent pas produire d'inter-blocages, ni limiter trop drastiquement le parallélisme entre threads.

Ainsi, l'utilisation d'un petit nombre de protection à gros grains (protégeant plusieurs variables partagées) facilite le raisonnement mais réduit le parallélisme, alors qu'une granularité plus fine permet théoriquement de meilleures performances au risque d'introduire des inter-blocages.

Un certain nombre de travaux ont été proposés pour aider les programmeurs dans ces choix. On peut citer par exemple [7], qui propose une approche permettant de déduire automatiquement des *mutex* de différente granularité pour des sections atomiques dans des programmes C. Cette approche cherche notamment à identifier le niveau de granularité le plus fin tout en garantissant l'absence d'inter-blocage (en se basant sur le protocole proposé par Gray *et al.* [8]). Une autre approche, proposée par Hicks *et al.* [9], se base sur une analyse des sections atomiques en utilisant un calcul dédié (*L-calculus*). Enfin, on peut également citer l'approche proposée par McCloskey *et al.* [10], qui repose sur un ensemble d'annotations fournies par le programmeur.

La majorité des travaux qui étudient l'implémentation des sections atomiques selon l'approche "pessimiste" se focalisent ainsi essentiellement sur la recherche des variables et portions de code qui doivent être protégées. Les méthodes les plus utilisées reposent sur des analyses statiques, et notamment des analyses d'alias. On peut noter que, malgré la complexité et la finesse de ces analyses, les résultats obtenus sont souvent utilisés de manière assez grossière. En effet, dans la plupart des cas, l'ensemble des protections nécessaires pour protéger le contenu d'une section critique est acquis *globalement* en début de section, puis relâché *globalement* à la fin de la section.

Nous proposons dans ce travail différentes directions permettant d'améliorer cet aspect de la gestion des mécanismes de protection :
– La définition et l'étude comparative de différentes *politiques* d'allocation et désallocation des mécanismes de protection. Nous montrons ainsi que la politique "globale" la plus souvent mise en œuvre n'est pas toujours la plus efficace, et que d'autres politiques, qui permettent de limiter les durées de possession de ces mécanismes de protection, sont parfois plus performantes.
– La réalisation d'une *bibliothèque* de protections dédiée, dont l'interface permet l'allocation/désallocation d'un ensemble de protections, réduisant ainsi les temps d'exécution par rapport à l'utilisation d'implémentations classiques comme les *pthreads*.

Notons que toutes les politiques que nous proposons respectent le principe du vérouillage à deux phases (*two-phase locking* [11]), qui garantit la séquentialité (*serializability*) des sections atomiques. Par ailleurs, l'absence d'inter-blocage est assurée par le respect d'un *ordre total* [12, 13] sur les variables partagées lors de l'acquisition de protection, solution aussi utilisée dans [9, 10].

La suite de cet article est organisée de la façon suivante : la section 2 définit à la fois les différents types de protection que nous avons considéré et les politiques d'allocation/désallocation que nous proposons ; la section 3 décrit l'implémentation de notre bibliothèque de protection et les résulats expérimentaux obtenus ; enfin, la section 4 conclut ce travail.

2. Politiques de gestion des protections

Dans cette section nous précisons tout d'abord la notion de mécanisme de protection utilisée dans le cas d'une approche pessimiste d'implémentation de sections critiques. Nous proposons ensuite différentes politiques de gestion de ces protections.

2.1. Mécanismes de protection

Comme indiqué en introduction, les *mécanismes de protections* ont pour objectif de gérer l'accès à des variables partagées. Plus précisément, une *protection* permet/restreint différents types d'accès, pouvant être soit exclusifs, soit partagés entre plusieurs threads. De plus, une protection peut être de granularité variable (protéger une seule variable *vs* protéger un ensemble de variables). Dans la suite de cet article nous adopterons les notations suivantes :

1. V est un ensemble de variables partagées, $x \in V$.
2. $SC = [s_1, s_2, \ldots, s_n]$ est une section critique (sans branchements), où s_i est une instruction effectuant soit une lecture (read(x)) soit une écriture (write(x)), de la variable partagée x.
3. P_i, un ensemble de *protections* nécessaires pour exécuter l'instruction s_i en protégeant l'accès aux variables partagées :
   - lorsque s_i est une lecture, au moins une protection non- exclusive doit être obtenue,
   - lorsque s_i est une écriture, une protection exclusive doit être obtenue.

Nous présentons ci-dessous à titre d'exemple deux types de protections.

Mutex : Le type de protection *mutex* donne accès exclusif (écriture/lecture) au thread qui l'a obtenu.

$$P_i = \{m_x \mid read(x) \in s_i \vee write(x) \in s_i\}, \quad \text{où } m_x \text{ est une protection de type } \textit{mutex} \text{ sur la variable } x$$

Read/Write (RW) : Ce type de protection distingue le mode d'accès à une variable, l'accès en lecture est non exclusif, l'accès en écriture est exclusif.

$$\begin{aligned} P_i \quad = \quad & \{r_x \mid read(x) \in s_i\} \bigcup \\ & \{r_x, w_x \mid read(x) \in s_i \wedge write(x) \in s_j, j > i\} \bigcup \\ & \{r_x, w_x \mid write(x) \in s_i\}, \end{aligned}$$

où r_x est une protection sur la lecture de la variable x et w_x est une protection sur l'écriture de la variable x.

2.2. Garantir l'absence d'inter-blocage

L'utilisation de *protections* pour résoudre des problèmes de synchronisation peut introduire des inter-blocages. Une solution souvent utilisée est de définir un ordre sur les variables partagées [12, 13] et d'imposer que tous les threads respectent cet ordre lors de l'acquisition de protections.

Supposons que $V = \{x, z, y\}$ soit un ensemble de variables partagées. On définit alors un ordre total $<$ sur les variables, par exemple $x < y < z$. Soit π un thread devant exécuter une section critique $SC = \{s_1, ..., s_k, ..., s_m, ..., s_n\}$. Si l'exécution de s_k nécessite une protection sur y tandis que s_m nécessite une protection sur x, le thread π sera obligé d'obtenir la *protection* sur x *avant* celle sur y, même si l'accès à x ne se fera que plus tard (car $x < y$).

Dans la suite, pour faciliter l'identification des *protections* qui doivent être obtenues en avance, on définit la fonction **Prefix(P)** qui, pour un ensemble de protections P donné, renvoie l'ensemble des protections compatibles avec P par rapport à la relation d'ordre considérée $<$ sur les variables partagées du programme.

La fonction **Prefix(P)** est définie de la manière suivante :

– Pour des protections de type **Mutex :** $Prefix(P) = \{m_x \mid \exists y \,.\, x \leq y \wedge m_y \in P\}$
– Pour des protections de type **Read/Write (RW) :** $Prefix(P) = \{r_x, w_x \mid \exists y \,.\, x \leq y \wedge (r_y \in P \vee w_y \in P)\}$

2.3. Algorithme général de gestion des protections

Le problème de la gestion des protections pour l'implémentation de sections critiques dans le cas d'une approche "pessimiste" de la concurrence se pose de la manière suivante :
– assurer que les protections P_i prévues pour exécuter chaque instruction s_i sont bien acquises (afin de garantir l'atomicité des sections critiques) ;
– éviter les inter-blocages ;
– autoriser un parallélisme le plus maximal possible entre les différents threads.
Nous formalisons tout d'abord ce problème en donnant des règles permettant de synthétiser un ensemble minimal de protections étendu H_i qu'il est nécessaire d'acquérir avant chaque instruction s_i. Un algorithme général de gestion des protections consiste alors à exécuter chaque section critique en

acquièrant avant chaque instruction les nouvelles protections nécessaires ($H_i \setminus H_{i-1}$) puis en relachant celles qui ne sont plus utiles ($H_i \setminus H_{i+1}$). Pour la première instruction s_0 on doit obtenir H_0 tandis que pour la dernière s_n on doit relâcher H_n. Les politiques de gestion que nous présenterons en section 2.4 sont des instances de cet algorithme général.

1. Toute protection p_x nécessaire pour l'exécution de l'instruction s_i doit être acquise.

$$\frac{p_x \in P_i}{p_x \in H_i}$$

2. Absence d'inter-blocage : les protections sont obtenues par tous les threads en respectant l'ordre total défini sur les variables partagées.

$$\frac{p_y \in H_i \quad p_x \in H_j \quad i < j \ \wedge \ x < y}{p_x \in H_i}$$

3. Vérouillage à deux phases, pour garantir la séquentialité des sections critiques. La première phase permet d'obtenir les protections, la seconde permet de les relâcher. La règle 3.a garantit qu'une protection est obtenue/relâchée une seule fois par section critique, tandis que 3.b garantit qu'il existe une position dans la section critique où toutes les protections sont obtenues.

$$3.a \ \frac{p_x \in H_i \quad p_x \in H_k \quad i < j < k}{p_x \in H_j} \qquad 3.b \ \frac{p_x \in H_i \quad p_z \in H_k \quad i < k}{\exists j \in [i,k] \ . \ (p_x, p_z) \in H_j}$$

2.4. Différentes politiques de gestion des protections

Dans cette section nous présentons cinq politiques pour obtenir/relâcher des *protections*. Chaque politique est d'abord présentée de manière intuitive, puis le calcul des protections (H_i) associées à chaque instruction s_i selon cette politique est formalisé. Toutes ces politiques respectent les règles décrites précédemment(2.3).

La figure 1 illustre l'exécution d'une section atomique dans laquelle trois variables partagées (x,y,z) sont accédées. Pour simplifier l'exemple on suppose que chaque variable est protégée par une *protection* de type *mutex* qui lui est associée. Sur la figure le code de la section atomique est répliqué pour chacune des politiques, celles-ci étant séparées par des lignes pointillées. Les zones hachurées indiquent les portions du code sur lesquelles chaque variable doit être protégée. Nous supposons ici que les variables sont utilisées dans un ordre qui garantit l'absence de blocages (x < y < z). Les lignes verticales qui traversent ces zones hachurées indiquent la durée de possession du *mutex* attribué à chaque variable selon la politique considérée.

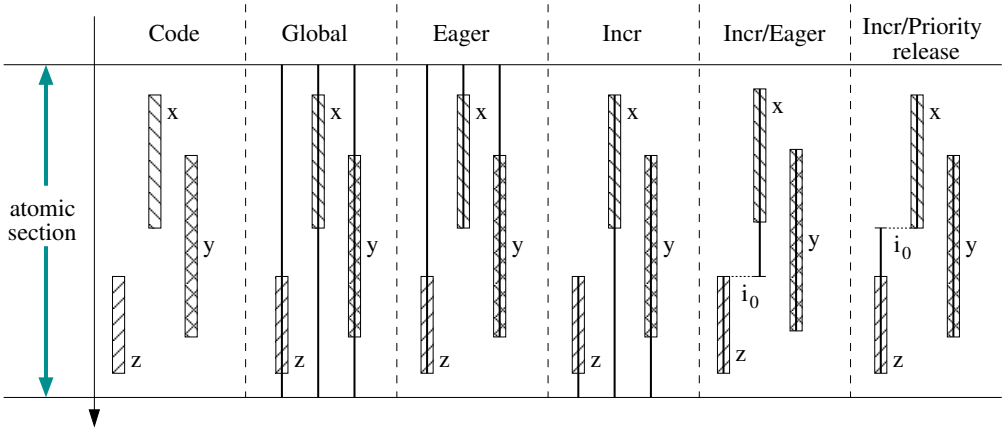

FIGURE 1 – Cinq politiques de gestion des protections

Plus formellement, chacune de ces politiques peut être décrite de la manière suivante :

Global : C'est la solution classique utilisée dans la majorité des applications. Toutes les *protections* sont obtenues dès le début et conservées jusqu'à la fin de la section atomique.

- $H_i = \bigcup_{j=1}^{n} P_j \ , \quad \forall i = 1, n$

Eager : Les *protections* sont relâchées dès qu'elles ne sont plus utilisées. A l'entrée d'une section critique, toutes les protections sont obtenues. Ensuite, dès qu'une variable n'est plus utilisée dans la section critique, on peut la relâcher. Ainsi d'autres threads qui sont potentiellement bloqués en attendant cette protection pourront progresser.

- $H_i = \bigcup_{j=i}^{n} P_j \ , \quad \forall i = 1, n$

Incr : Les *protections* sont acquises incrémentalement. Les threads essayent ainsi d'obtenir les protections le plus tard possible (mais avant d'accéder à la variable), puis elles sont relâchées globalement à la sortie de la section critique. Pour éviter les inter-blocages quelques protections doivent être acquises plus tôt.

- Règle 2 : $H_i^d = \bigcup_{j=i}^{n} P_j \bigcap Prefix(P_i) \ , \quad \forall i = 1, n$
- Règle 3.a : $H_i^a = \bigcup_{j<i} P_j \bigcap \bigcup_{j>i} P_j \ , \quad \forall i = 1, n$
- Règle 3.b : $H'_i = H_i^d \bigcup H_i^a \quad \forall i = 1, n$
- $H_i = \bigcup_{j \leq i} H'_j, \forall i[1, n]$

Incr/Eager : Les *protections* sont obtenues de nouveau incrémentalement, mais elles sont relâchées dès que possible. Toutefois, Pour respecter le vérouillage à deux phases, on ne relâche des protections que après avoir obtenu toutes les protections utilisées dans la section critique (noté i_0 ci-dessous). On observe sur la figure 1 que la possession du *mutex* pour la variable x a été prolongée jusqu'au point (i_0) où le *mutex* de la variable z a été obtenu.

- Règle 2 : $H_i^d = \bigcup_{j=i}^{n} P_j \bigcap Prefix(P_i) \ , \quad \forall i = 1, n$
- Règle 3.a : $H_i^a = \bigcup_{j<i} P_j \bigcap \bigcup_{j>i} P_j \ , \quad \forall i = 1, n$
- Règle 3.b : $H'_i = H_i^d \bigcup H_i^a \quad \forall i = 1, n$
- $H_i = \bigcup_{j \leq i} H'_j \ , \quad \forall i \leq i_0 \ , H_{i_0} \setminus H_{i_0 - 1} \neq \emptyset \ , H_k \setminus H_{k-1} = \emptyset \ \forall k \geq i_0$
  $H_i = H'_i \ , \quad \forall i > i_0$

Incr/Priority release : Les *protections* sont obtenues incrémentalement jusqu'à ce qu'on arrive à la dernière occurence d'une variable protégée. A ce point là, noté i_0 ci-dessous, on peut relâcher la protection, mais on doit être sûr d'avoir obtenu auparavant toutes les protections utilisées dans la section critique. Sur la figure 1 on remarque que le *mutex* sur la variable z a été anticipé jusque au point (i_0) ou le mutex sur la variable x peut être relâché.

- Règle 2 : $H_i^d = \bigcup_{j=i}^{n} P_j \bigcap Prefix(P_i) \ , \quad \forall i = 1, n$
- Règle 3.a : $H_i^a = \bigcup_{j<i} P_j \bigcap \bigcup_{j>i} P_j \ , \quad \forall i = 1, n$
- Règle 3.b : $H'_i = H_i^d \bigcup H_i^a \quad \forall i = 1, n$
- $H_i = \bigcup_{j \geq i} H'_j \ , \quad \forall i \geq i_0 \ , H_{i_0} \setminus H_{i_0 + 1} \neq \emptyset \ , H_k \setminus H_{k+1} = \emptyset \ \forall k < i_0$
  $H_i = H'_i \ , \quad \forall i < i_0$

3. Implémentation et expérimentations

Dans cette section nous présentons notre implémentation des *protections* et quelques expérimentations qui comparent les performances obtenues en utilisant les différentes politiques présentées dans la section précédente.

3.1. Implémentation des protections

Les protections sont implémentées dans une bibliothèque *C++* non optimisée, utilisant des structures de données de *STL* comme *Linked-List* et *Set*. Nous avons implémenté les *mutex* et *read/write*. Les deux principales caractéristiques de notre implémentation sont :

– Les acquisition des protections sont *ré-entrantes*. Cela signifie que si un thread réclame une protection qu'il possède déjà il ne restera pas bloqué.
– Un thread peut réclamer un ensemble de protections atomiquement. Cela signifie que soit toutes les protections de l'ensemble vont être obtenues par le thread (si toutes celles-ci sont disponibles), soit aucune (dans le cas contraire, le thread restant alors bloqué). Cette propriété est importante quand on ne peut pas définir un ordre sur les variables (par exemple en cas d'allocation dynamique de mémoires). Dans ce cas là, on peux utiliser les politiques *Global et Eager Release* sans avoir d'inter-blocages.

Dans notre implémentation, pour protéger les structures de données concernant les protections elles-mêmes on utilise un *mutex* global (basé sur l'utilisation des *pthreads*).

3.2. Expérimentations

Nos résultats expérimentaux sont basés sur trois applications. Pour chaque application nous avons comparé les quatre premières politiques mises en œuvre en utilisant des *pthread fast mutex*, et des *mutex* implémentés par notre bibliothèque. Dans nos exemples la politique *Incr/Priority release* était équivalente à *Incr/Eager*. Ceci est dû à la structure des exemples pour lesquels le verrouillage à deux phases est respecté par construction. Ce qui distingue les deux politiques, est la définition du point i_0. Comme présenté dans la Figure 2 le point i_0 de *Incr/ Eager* se trouve avant le point i_0 de *Incr/Priority release* et donc les deux politiques donnent exactement le même résultat. Toutes les expérimentations ont été effectuées sur un *Intel Xeon W3520* à quatre cœurs avec *6GB de mémoire* et système d'exploitation *Debian GNU/Linux 5.0.8 (lenny)*. Le code a été compilé avec *GCC 4.3.2* .

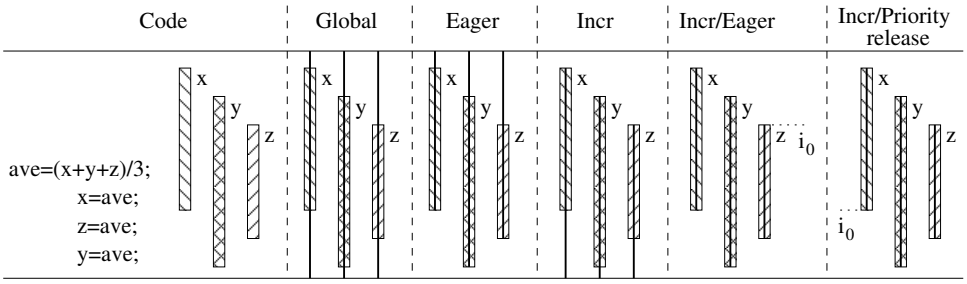

FIGURE 2 – Politiques Incr/Eager et Incr/Priority release dans le cas de l'application 1

3.2.1. Description des applications

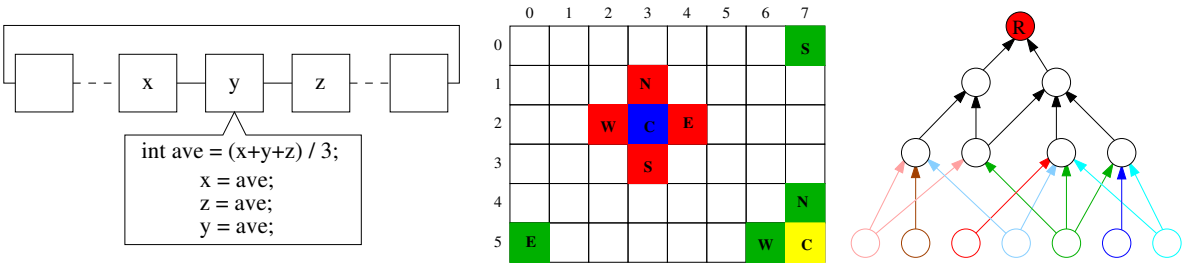

FIGURE 3 – Architecture des expérimentations.

Application 1 et 2 : Calcul de moyennes.

Il s'agit du calcul de la moyenne d'un ensemble d'éléments (ici des entiers), organisé selon deux architectures (Figure 3, parties gauche et centrale) :

1. une liste circulaire, dans laquelle chaque élément a deux voisins ;
2. un tore de dimension 2, dans lequel chaque élément a quatre voisins.

A chaque élément de l'ensemble est associé un thread. Toutes les valeurs des éléments sont des variables partagées, mais chaque thread ne peut accéder qu'aux valeurs des éléments qui lui sont voisins. L'algorithme exécuté par chaque thread consiste alors à remplacer continûment la valeur de la variable qui lui est associée, ainsi que celle de ses voisins, par la moyenne des valeurs des variables associées à ses voisins, et la sienne. L'algorithme se termine lorsque tous les éléments ont la même valeur. Toutefois, pour les besoins de notre expérimentation (reproductibilité des résultats, observation de temps de calcul siginificatifs entre deux accès mémoire) nous avons modifié cet algorithme d'une part en fixant une borne au nombre d'itérations effectuées, d'autre part en ajoutant un calcul de fonction d'*Ackermann* (avec deux arguments de valeurs 3 et 10) en plus du calcul de moyenne.

Application 3 : Communication dans un réseau.

Cet exemple est inspiré d'algorithmes de diffusion dans un réseau sans fil. Nous considérons un ensemble de nœuds qui doivent propager des informations vers un nœud racine. La topologie (arborescente) considérée est celle présentée sur la figure 3 (partie droite). Comme on peut le constater, chaque nœud peut émettre vers plusieurs parents. Un thread est attaché à chaque nœud, et une variable partagée est associée à chaque thread. Une émission est alors représentée par une écriture du nœud émetteur de la variable partagée asociée au nœud récepteur. Deux modes de communication sont possibles : un mode

de type *unicast* dans lequel un noeud ne communique qu'avec un seul parent, et un mode *multicast* dans lequel un noeud communique *de façon atomique* avec l'ensemble de ses parents. On introduit en outre une probabilité d'échec pour chaque communication.

3.2.2. Résultats

Pour l'application 1 (liste circulaire), nous avons considéré un ensemble de 20 nœuds effectuant chacun 20 itérations (accès au voisins, calcul de moyenne, Ackermann). Les accès à la mémoire ont été simulé avec un délai de 1 seconde pour amplifier l'effet des synchronisations.

Pour l'application 2 (tore de dimension 2), nous avons considéré un ensemble de 100 nœuds (10×10), effectuant chacun 5 itérations de l'algorithme. Les accès mémoire durent également une seconde.

Pour l'application 3, nous avons considéré une topologie arborescente de 70 nœuds, qui comportait 6 niveaux, et dans laquelle chaque nœud avait au plus 4 parents. La racine devait recevoir 50 messages. Chaque transmission de message dure 1 seconde, et son taux de réussite est de 80%. En cas d'échec le message est re-transmis avec succès. Cette probabilité permet de faire varier les comportements de l'algorithme d'une itération sur l'autre.

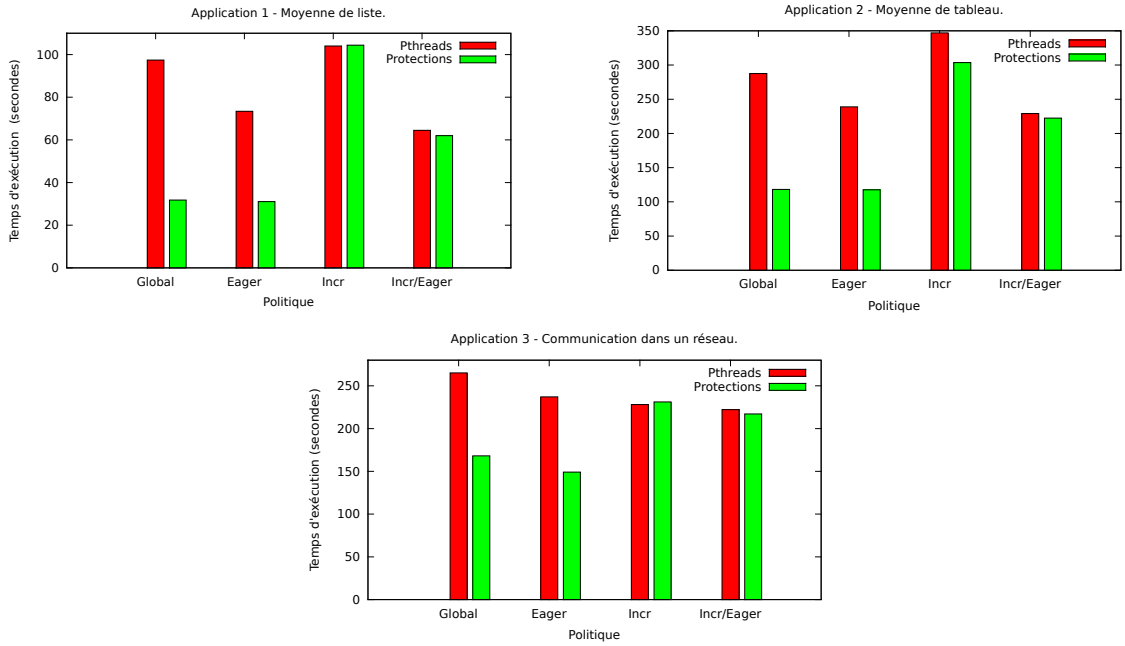

FIGURE 4 – Comparaison des politiques et des mécanismes de protection utilisés

La figure 4 fournit les histogrammes indiquant les temps d'exécution obtenus pour chaque application, en considérant d'une part chacune des politiques introduites dans la section 2.4, et, d'autre part, deux implémentations possibles des *mutex*, celle fournie par la bibliothèque standard *pthread* (fast mutex), et celle que nous avons implémenté (section 3.1). On constate que, dans tous les cas, les politiques de type *eager release* donnent de meilleures performances que la politique *global*. Ceci est dû au fait que, en relâchant les protections plus tôt, on donne la possibilité à d'autres thread de progresser. La politique *Incr* n'est pas très efficace dans le cas des applications 1 et 2. La raison est que le parallélisme est réduit car chaque thread obtient un sous-ensemble de protections qui ne lui permet pas de terminer sa section critique, il n'y a donc qu'un petit nombre de threads qui peuvent progresser en parallèle. La figure 5 illustre le cas pour l'application 1. Les flèches solides dénotent que le nœud d'origine a obtenu la protection du nœud destinataire. Le nœud *n-1* est le seul qui puisse progresser.

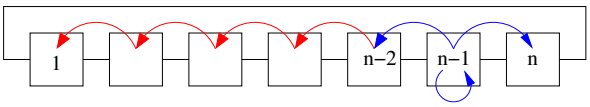

FIGURE 5 – Problème d'acquisition incrémentale dans l'application 1

Par ailleurs, notre implémentation des *mutex* réduit fortement les temps d'exécution par rapport à ceux obtenus avec la bibliothèque *pthread* standard. Cela est dû à l'utilisation de primitives d'acquisition

atomique des protections : soit l'ensemble des mutex nécessaire à un thread est obtenu en une seule opération, soit cet ensemble est laissé *entièrement* disponible pour d'autres threads. Notons, que même dans le cas des politiques d'acquisition incrémentales, pour lesquelles les protections doivent nécessairement être obtenues une à une, notre implémentation reste la plus efficace.

4. Conclusion

Ce travail avait pour objectif l'amélioration des temps d'exécution de sections critiques dans le cadre d'une approche de type "pessimiste", basée sur l'utilisation de protections. Nous avons proposé cinq politiques de gestion (allocation/désallocation) de protections qui garantissent l'atomicité des sections critiques et l'absence d'inter-blocage. Nous avons montré à travers des expérimentations que l'utilisation de ces politiques peut influer nettement sur les temps d'exécution, la politique *eager release* étant, pour les exemples considérés, bien meilleure que la politique *global* utilisée classiquement. De plus, nous avons implémenté une bibliothèque qui permet une acquisition atomique d'un ensemble de protections. Ceci a amélioré considérablement les performances.

Plusieurs perspectives sont envisageables pour ce travail. Tout d'abord les expérimentations effectuées mériteraient d'être étendues à des exemples réels comme par exemple le *benchmark PARSEC* [14]. Ensuite, il serait intéressant de définir plus précisément quelles sont les "patrons de synchronisation" (*synchronization patterns*) sur lesquels telle politique est plus avantageuse que telle autre. On pourrait ainsi obtenir des heuristiques permettant d'automatiser le choix de la politique en fonction de la structure du code. Des travaux dans ce sens, sur les calculs itératifs, ont été proposés dans [15]. De plus, nous avons constaté que les performances des politiques incrémentales était influencées par l'ordre total défini sur les variables partagées. Déterminer un ordre "optimal" (s'il existe) vis-à-vis de ces politiques serait donc un plus.

Finalement, du point de vue de l'implémentation des protections on pourrait utiliser des *mutex* de plus fine granularité pour améliorer la performance de gestion des protections. En outre, on pourrait introduire des mécanismes d'ordonnancement pour éviter des phénomènes de *famine*. Cela peut arriver quand un thread essaie d'obtenir un grand ensemble de protections, atomiquement, qui ne sont jamais disponibles simultanément, due à la compétition avec d'autres threads. Ainsi, on pourrait envisager l'utilisation d'ordonnancement de type *FIFO*, même si cela risque de réduire fortement le parallélisme. Notons qu'une solution simple qui pourrait éviter le besoin d'ordonnancement est de définir un nombre maximum de protections pouvant être réclamées atomiquement, mais cela ne garantirait pas toujours l'absence de *famine*.

Bibliographie

1. Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11), 2003.
2. M. Herlihy, V. Luchangco, M. Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. *Proceedings of PODC '03*, 2003.
3. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. *Communications of the ACM*, 51(8), 2008.
4. Tom Knight. An architecture for mostly functional languages. *Proceedings of ACM LFP 1986*, 1986.
5. Nir Shavit and Dan Touitou. Software Transactional Memory. *PODC*, 1995.
6. Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. *Network*.
7. S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. *PLDI*, 43(6), 2008.
8. JN Gray, RA Lorie, and GR Putzolu. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases*, Framingham, Massachusetts, 1975. ACM.
9. Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Lock Inference for Atomic Sections. 2006.
10. Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker : synchronization inference for atomic sections. *POPL*, 2006.
11. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), 1976.
12. J.W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal1*, 1968.
13. E.G. Coffman, M.J. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2), 1971.
14. Christian Bienia. Benchmarking Modern Multiprocessors. 2011.
15. Pierre-Nicolas Clauss and Jens Gustedt. Iterative computations with ordered read-write locks. *Parallel Distributed Computing*, 5, 2010.