PROJET DE RECHERCHE

Nantes Université

Master Informatique
Parcours ALMA
Architecture Logicielle
1re année

COMPLEXITÉ DES ALGORITHMES WAIT-FREE DANS LES ARCHITECTURES MODERNES

Janvier - Mai 2025:

Erwann CUSSAGUET, Maxime POULALION, William REPPERT, Nathan SCOHIER

Encadré par : **Matthieu PERRIN**

Mots-clés. Programmation Répartie ; Mémoire Partagée ; Concurrence ; Construction Universelle ; Complexité ; Processus ; Threads ; Wait-Free ; Instructions Spéciales ;

1. Introduction

1.1 Contexte

La question de la gestion des accès concurrents aux ressources dans l'informatique existe depuis la création de celle-ci. Même avec une machine ne possédant qu'une unique unité d'exécution, comment réussit-on à répartir les accès à celle-ci sur plusieurs utilisateurs. Cette question à mené à la création des mécanismes d'ordonnancement et des verrous. Ces derniers ont pour objectif de bloquer une partie d'un programme, appelée « Section Critique », pour ne laisser qu'un seul processus ou utilisateur actif à la fois dans celle-ci.

L'objectif est d'assurer une propriété appelée « Linéarisabilité », qui permet de représenter l'exécution des processus concurrents dans un système distribué comme si elles étaient séquentielles. [1]

Les verrous sont des mécanismes relativement simples à implémenter, mais posent de gros problèmes de temps d'attente et de répartition des ressources. Les algorithmes utilisant des verrous font partie d'une catégorie d'algorithmes appelés algorithmes « Bloquants ». Un exemple d'implémentation des verrous est celui de l'algorithme de la boulangerie de Lamport [2], qui vient affecter un ordre de passage à chaque processus entrant.

Parmi ces problèmes on trouve celui de l'interblocage dans lequel deux processus attendent chacun indéfiniment une ressource possédée par l'autre, bloquant définitivement le programme. Un autre problème est celui de famine, qui apparaît lorsque des tâches à haute priorité prennent le dessus constamment sur la tâche de plus faible priorité. En général, l'utilisation de verrous aligne la vitesse d'exécution globale sur celle du thread le plus lent. Enfin, les verrous ne tolèrent pas les pannes. Si une tâche plante en possédant un verrou,

ce verrou ne sera jamais libéré, pouvant potentiellement entraîner un blocage du programme entier.

Afin de garantir un certain niveau de résilience face aux pannes franches, des algorithmes implémentant des structures de données de façon concurrente sans verrous - dits « Non-bloquants » ont été développés. Ces algorithmes garantissent un certain niveau de progression des différentes tâches et processus actifs.

Parmi eux, les algorithmes dits « Lock-Free » dont l'objectif est de garantir une progression globale du système, même si la complétion individuelle de toutes les tâches n'est pas garantie, une opération peut potentiellement terminer en un nombre infini d'étapes [1]. Des exemples d'algorithmes lock-free implémentant des structures de données sont la pile de Treiber [3] et la file de Michael-Scott [4].

Une solution pour garantir la complétion individuelle de toutes les tâches est l'utilisation d'algorithmes dits « Wait-Free », dont l'objectif est de faire terminer tous les processus entrants en un nombre fini d'étapes. [5] Un exemple d'algorithme wait-free implémentant une pile est la pile de Afek, Gafni et Morrison [6].

Cependant, certains algorithmes ne peuvent pas garantir qu'un processus seul soit capable de terminer son opération, ce qui pose problème lorsque l'on souhaite avoir un algorithme wait-free. Un autre mécanisme, appelé « helping » [7] peut alors être utilisé, permettant à tout processus d'aider les autres à terminer leurs opérations s'ils en ont la possibilité. Ce mécanisme permet donc de garantir qu'un algorithme concurrent soit wait-free s'il est utilisé correctement.

Les algorithmes cités ici ne sont cependant que des solutions appliquées à des structures de données spécifiques. Afin de simplifier et généraliser la conception de structures de données concurrentes, des algorithmes généraux appelés « Constructions Universelles » ont été créés. Ces algorithmes représentent des bases permettant d'adapter certaines structures de données de

façon concurrente au modèle concurrent choisi. [8] Ceux-ci ont pour objectif d'assurer la cohérence entre les différents processus essayant d'accéder à une structure de données, en plus d'assurer un certain niveau de progression (Lock-Free, Wait-Free, ...). Ces algorithmes peuvent être représentés par plusieurs sous-algorithmes et ont pour objectif d'ordonnancer et d'appliquer les opérations des processus actifs. Pour ce faire, les constructions universelles ainsi que les algorithmes spécifiques utilisent des instructions spéciales facilitant la gestion de la concurrence.

Ces instructions spéciales sont implémentées dans le matériel les utilisant. Des solutions logicielles existent pour les machines dont le matériel ne possède pas ces instructions, mais celles-ci ont le désavantage d'être lentes, ou de dépendre de plusieurs autres instructions, ce qui crée une couche d'abstraction supplémentaire pouvant rendre le débogage compliqué, ou altérer complètement leur fonctionnement. Ces instructions réalisent leur action de manière atomique. C'est-à-dire qu'un seul et unique processus est autorisé à exécuter l'instruction sur un espace mémoire donné. Parmi ces instructions, on trouve :

— L'instruction « Fetch-And-Add » qui vient incrémenter atomiquement un registre d'une certaine valeur tout en renvoyant l'ancienne valeur.
— L'instruction « Test-And-Set » qui viens mettre la valeur 1 dans un registre et renvoie l'ancienne valeur de ce registre. Cette opération n'agit que sur un mot, c'est-à-dire un octet.
— L'instruction « Compare-And-Set » (dorénavant abréviée « CAS ») qui compare la valeur stockée dans le registre ciblé avec le premier argument donné. Si elles sont égales, la valeur du registre est remplacée par le second argument. Enfin l'instruction renvoie un booléen indiquant si l'écriture à été effectuée ou non.

Ces instructions viennent aider les algorithmes concurrents à décider quels processus accèdent et lisent/modifient une ressource. Cela implique plusieurs choses :

— Les instructions ne garantissent pas que l'action est effectuée. Dans le cas du Compare-And-Set, un processus peut perdre son CAS car la valeur de l'espace mémoire a été changée par un CAS concurrent avant le sien.

— Les instructions ne laissent pas obligatoirement de trace de leur passage. Une instruction Fetch-And-Increment (dorénavant abréviée FAI), dérivée de Fetch-And-Add, qui incrémente de 1 un registre peut nous donner une indication du nombre d'utilisations de celle-ci. Un CAS en revanche viens remplacer le contenu d'un registre qui n'est pas forcément un entier. Le traçage devient alors compliqué.

1.2 Problématique

De par sa versatilité, l'instruction CAS est la plus utilisée dans la création de constructions universelles. Elle n'est cependant pas sans défauts. Comme dis précédemment, elle ne laisse en général pas de trace de son passage. Lorsqu'une construction universelle essaie de garder en mémoire les processus actifs afin de les appliquer un à un, cela entraîne le besoin d'inscrire ces opérations manuellement. Un exemple de construction universelle utilisant ce procédé est le Weak Log [9]. Celle-ci maintient une liste des processus ayant réussi leur CAS, et des sous-listes de ceux ne l'ayant pas réussi. Cela entraîne le besoin de conserver les opérations mêmes si celles-ci sont déjà terminées et appliquées.

On observe alors un problème de complexité spatiale lorsque CAS est l'unique instruction spéciale utilisée pour annoncer une opération dans une construction universelle. A cause de ce besoin de garder une trace constante des opérations dans la construction, la complexité en mémoire de la construction universelle augmente continuellement. A un moment donné, celle-ci dépend donc du nombre total d'opérations ayant utilisé l'algorithme, et ce même si aucun processus n'est activement en train de s'insérer.

La complexité spatiale d'un algorithme de construction universelle lorsque celui-ci ne possède pas de processus actifs l'utilisant est appelée « Complexité Quiescente ».

Dans un article de Denis Bédin, François Lépine, Achour Mostéfaoui, Damien Perez et Matthieu Perrin [10], il a été prouvé qu'un algorithme de construc-

tion universelle wait-free n'utilisant que l'instruction CAS ne peut pas avoir une complexité quiescente constante.

Dans une situation idéale, le nombre de processus présent ou utilisant un algorithme est toujours connu. En pratique ce n'est pas toujours le cas. Il nous faut alors prendre en compte les différents problèmes de décision posés par la concurrence dans la conception de nos algorithmes, tel que le problème du consensus dans lequel plusieurs processus se mettent d'accord sur une valeur commune lorsque ceux-ci en proposent plusieurs. [11]

Est-il alors possible de trouver un algorithme avec une complexité quiescente en utilisant les instructions spéciales disponibles dans les architectures modernes ?

1.3 Approche

Nous avons tout d'abord commencé par étudier les constructions universelles existantes, dont le weak log, ainsi qu'une construction universelle à base de CAS utilisant une liste d'annonce pour représenter les opérations à effectuer ainsi qu'un nœud de linéarisation permettant d'appliquer les opérations sur la structure de données.

Suite à cette étude, nous avons cherché à nous inspirer de ce second algorithme afin de ne plus dépendre entièrement de l'instruction CAS. Notre construction se base elle aussi sur une liste d'annonce ainsi que sur un nœud de linéarisation [10]. Chaque nœud d'annonce dans la liste d'annonce possède un indice correspondant à sa position par rapport aux autres, et peut posséder ou non un nœud d'opération. Ce nœud d'opération représente l'opération à effectuer par un processus, ainsi que son état et son résultat.

Le fonctionnement de l'algorithme de construction universelle est divisé en trois étapes ; l'insertion d'un processus dans la liste d'annonce, durant lequel le processus essaie de s'insérer dans la liste d'annonce à un indice spécifique. L'exécution de l'opération, faisant recours à un mécanisme d'aide qui consiste à exécuter les opérations des processus inscrits dans la liste d'annonce avant

lui afin de garantir la complétion de toutes les opérations. Enfin, le retrait de l'opération de la liste car il n'est plus nécessaire. Ces trois étapes nous permettent de ne pas utiliser de verrous pour gérer la concurrence. L'utilisation d'un mécanisme d'aide nous permet aussi de rendre nôtre construction wait-free.

Une fois tous les processus actifs terminés, les processus actifs seront tous retirés sauf un. La complexité quiescente dépend alors de l'élément unique de la liste ainsi que du nœud de linéarisation. Nous estimons qu'elle est donc constante.

1.4 Contributions

Nous proposons les contributions suivantes :

- Un algorithme de construction universelle basé sur les instructions FAI et CAS avec une complexité quiescente constante.
- Une implémentation en Java de cette construction. [1]
- Les preuves liés à l'algorithme :
  - La preuve que l'algorithme est wait-free.
  - La preuve que l'algorithme est correct.
  - La preuve que l'algorithme est linéarisable.
  - La preuve que l'algorithme possède une complexité quiescente constante.

1.5 Organisation de l'article

Nous commencerons par détailler le modèle utilisé lors de la création de la construction universelle. Nous détaillerons ensuite le fonctionnement de l'algorithme, avec l'insertion des processus dans la liste d'annonce, l'exécution de ceux-ci, le mécanisme d'aide permettant aux processus les plus rapides d'aider les plus lents, et enfin le retrait des opérations terminées de la construction.

1. Ajouter le lien vers dépot

Ensuite nous prouverons la condition de progression wait-free de l'algorithme, la linéarisabilité de celui-ci, sa preuve de correction, et enfin la preuve que sa complexité quiescente est bien constante.

Enfin, nous conclurons enfin sur les avantages et inconvénients de notre solution.

2. Modèle

Le modèle concurrent utilisé considère une machine contenant plusieurs unités d'exécution (cœurs). Cette machine possède plusieurs processus, chaque processus pouvant créer un nombre indéterminé de threads. Le système peut posséder plus de threads que de cœurs. Nous ne distinguons pas processus de threads car leur utilisation dans notre cas est très similaire. Chaque processus représente ici une séquence linéaire d'instructions à exécuter.

Vitesse d'exécution et nombre de processus

La vitesse d'exécution des processus est indéterminée, varie, et il n'existe pas d'horloge commune, le système est donc asynchrone durant sa durée d'activité. Les processus peuvent être victimes de plusieurs événements, tels que des pannes franches, des préemptions venant de l'ordonnanceur du système d'exploitation, ou peuvent être extrêmement lents à cause de leur tâche à réaliser. Dans cet article, nous ne distinguerons pas ces problèmes. Les processus affectés par ceux-ci seront indiqués comme étant « Infiniment Lent ».

Le modèle considère un nombre indéterminé, potentiellement infini de processus sur lesquels nous ne connaissons rien. Leur état courant et terminaison n'est pas connu, ni de nous, ni des autres processus. [12]

Modèle de mémoire

La machine contient un nombre indéterminé mais fixe d'espaces mémoire appelées « registres ». Ceux-ci sont de taille fixe, en bits. Chaque processus possède un accès illimité en écriture et en lecture sur ces registres.

Nous distinguons la mémoire locale à un processus, dans lequel il est libre d'effectuer n'importe quelle opération sur ses registres sans avoir à se préoccu-

per d'un potentiel accès concurrent, de la mémoire partagée, composée de registres pouvant être accédés simultanément par plusieurs processus. Il revient alors au développeur la tâche d'assurer la cohérence et la synchronisation de cette mémoire avec l'aide des instructions fournies et de l'état de l'art concernant la programmation distribuée.

Nous supposons aussi l'existence d'un garbage collector, venant réclamer les espaces mémoires non référencés pendant l'exécution de l'algorithme. Il n'est donc pas nécessaire de gérer manuellement les désallocations mémoire.

Communication et accès aux ressources

Enfin, le modèle nous donne accès aux instructions de lecture et d'écriture, ainsi qu'aux instructions spéciales Fetch-And-Increment ainsi que Compare-And-Set. Nous supposons ici que toutes les instructions s'exécutent de façon atomique.

Fetch-And-Increment est une instruction dérivée de l'instruction spéciale Fetch-And-Add, qui incrémente de 1 le registre ciblé au lieu d'y ajouter une valeur spécifiée.

Compare-And-Set cible un registre donné, compare sa valeur avec le premier argument, et si ceux-ci sont égaux, il remplace le contenu du registre par le second argument. L'instruction retourne un booléen indiquant si le CAS à été un succès ou un échec.

Ces instructions spéciales sont implémentées dans le matériel de la machine, nous n'utilisons pas de solution logicielle pour ces isntructions.

3. Algorithme

Cette section a pour but d'expliquer en détails les idées et le fonctionnement derrière la construction universelle proposée. Les algorithmes seront indiqués et accompagnés d'un schéma montrant la structure de données dans sa globalité.

3.1 Structures de données

La construction universelle est supportée par une structure principale composée de trois espaces mémoire. Ces espaces permettent à la structure de conserver une liste des opérations à effectuer et des opérations à venir, ainsi que d'appliquer ces opérations sur l'état du système et d'en récupérer les résultats.

Indice de positionnement

Le premier espace mémoire représente un registre nommé k contenant un entier non-signé sur n bits, n étant défini par la spécification de la machine. La valeur maximale de ce registre est donc $2^n - 1$. Ce nombre représente le prochain indice j qu'un processus utilisant l'algorithme va prendre afin d'introduire une nouvelle opération à exécuter. Ce mécanisme crée un ordre dans l'annonce des opérations pour pouvoir les placer indépendamment des autres, sans créer de concurrence sur leur position. Cet indice ne peut être incrémenté que de 1 par un seul processus à la fois, en utilisant l'instruction spéciale fetch-and-increment vue en Section 2, c'est donc un compteur.

Liste d'annonce et opérations

Le second espace mémoire représente un registre nommé *Head* contenant une adresse vers une sous-structure nommée « Nœud d'annonce ». Ce registre

représente la tête d'une « Liste d'annonce ». Chaque nœud représente un espace mémoire contenant trois informations :

— Un indice Ind_i, représentant son emplacement par rapport aux autres nœuds. Si le nœud est précédé par un autre nœud, celui-ci possède obligatoirement un indice Ind_i strictement plus petit que le nœud actuel. S'il est suivi d'autres nœuds, ceux-ci possèdent tous un indice Ind_i strictement plus grand que le nœud actuel.
— Une adresse *Next* représentant le prochain nœud d'annonce. Si aucune opération ne s'insère après, il peut être vide.
— Une adresse OP_i vers une autre sous-structure nommée « Nœud d'opération », contenant les informations nécessaires à l'exécution de l'opération. Cet emplacement mémoire peut être vide si une opération n'a pas encore été insérée.

Un nœud d'opération contient lui-même trois informations :

— Un champ OP_i représentant la nature de l'opération à effectuer sur la structure de données implémentée par la construction universelle. Elle contient donc les différents paramètres en plus des instructions à exécuter.
— Un champ *Res* contenant le résultat de l'opération une fois celle-ci exécutée et terminée.
— Un champ *Done* contenant un booléen indiquant si l'opération est terminée ou non.

Il est important de noter que l'ordre établi dans la liste d'annonce ne représente pas l'ordre d'exécution des opérations. Nous verrons par la suite que si l'algorithme détecte un nœud d'annonce dont le nœud d'opération est vide, il poursuit son parcours et essaie d'exécuter le premier nœud d'annonce possédant un nœud d'opération qu'il trouve. Le rôle de la liste et du compteur est de permettre à chaque opération déclarée (en prenant un indice j du compteur k) de réserver un emplacement représenté par un nœud d'annonce pour y placer son opération.

Nœud de linéarisation

Le dernier espace mémoire représente un registre nommé *LinNode* contenant une adresse vers une sous-structure appelée « Nœud de linéarisation ». Cette structure permet d'appliquer une opération à la fois sur l'état du système (qui peut être directement la structure de données à implémenter de façon concurrente). Elle valide la propriété de linéarisabilité de l'algorithme. Le nœud contient lui aussi trois champs :

— Un champ *OP* contenant un pointeur vers le nœud d'opération dont l'opération doit être appliquée.
— Un champ *State* contenant l'état global du système. Celui-ci peut être la structure de données à implémenter, ou toute partie de la mémoire étant affectée par une opération.
— Un champ *Res* qui va contenir le résultat de l'opération à appliquer. Ce résultat pourra ensuite être transmis au nœud d'opération.

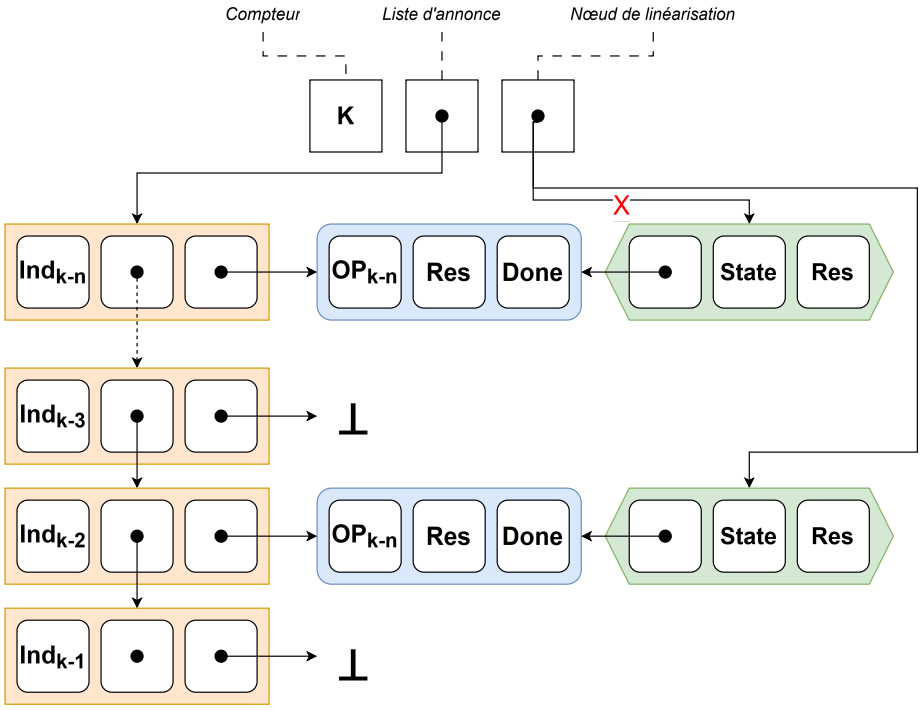

FIGURE 3.1 – Représentation de la structure de données utilisée par la construction universelle

Initialisation

La structure de données est toujours initialisée avec k égal à 1. La liste d'annonce $Head$ est initialisée avec un nœud d'annonce $ANode_{start}$ dont l'indice Ind_i est égal à 0, une opération OP_i vide elle-même constituée d'un un résultat Res vide, et un booléen $Done$ mis sur $True$. Enfin le suivant $Next$ de $ANode_{start}$ est vide. C'est un nœud d'annonce de base permettant d'avoir un début de liste, qui sera supprimé dès qu'une autre opération atteindra sa phase de retrait. Enfin, le nœud de linéarisation $LinNode$ est initialisé avec $OP = OP_i$, $State$ contient l'état courant du système, représenté ici par un registre, et un résultat Res vide.

Inspirations et différences

L'idée d'utiliser une liste de nœuds d'annonce, des nœuds d'opération et des nœuds de linéarisation est très largement inspirée de la construction universelle proposée dans l'article The Burden of the Past [10]. Celle-ci n'utilise que l'instruction CAS pour son fonctionnement et autorise le remplacement de l'adresse d'un nœud d'opération OP_i par d'autres nœuds d'opération dans un nœud d'annonce $ANode_i$, lui permettant d'accueillir plusieurs opérations dans un unique nœud afin de gérer plus efficacement la contention.

Ce n'est pas le cas dans notre construction universelle. Un nœud d'annonce $ANode_i$ ne peut posséder qu'un unique nœud d'opération $ONode_i$ ou une adresse vide. Chaque opération récupère effectivement un nœud d'annonce pour elle-même et n'essaie pas de partager cet espace avec d'autres opérations. Le problème de contention est réglé avec le compteur k établissant un ordre sur l'annonce des opérations.

3.2 Fonctionnement

Notre construction universelle se base sur un algorithme divisé en trois étapes utilisant la structure précédente comme support principal pour son exécution. Ces trois étapes sont l'insertion d'une opération dans la liste d'annonce, l'exécution et l'aide apportée aux autres opérations pour terminer, et enfin la

suppression de l'opération de la liste d'annonce afin de laisser le garbage collector réclamer l'espace mémoire utilisé.

Lors des prochaines explications, nous supposons la structure de données de la construction universelle comme étant initialisée.

3.2.1 Insertion

Afin de pouvoir insérer son opération dans la liste d'annonce, un processus commence par préparer localement son nœud d'opération $ONode_{local}$. Il est initialisé avec l'opération OP_i que le processus souhaite appliquer ainsi qu'un résultat *Res* vide et un booléen de terminaison *Done* mis sur *False*.

Le processus récupère ensuite l'indice de positionnement j de son opération dans la liste d'annonce en utilisant l'instruction Fetch-And-Increment sur k. Cet indice j est sauvegardé localement par le processus, et k est incrémenté.

Le processus continue en effectuant une lecture sur *Head* afin de récupérer localement le premier nœud d'annonce $ANode_i$. Il entre ensuite dans une boucle qu'il ne quitte qu'une fois que l'indice Ind_i du nœud d'annonce lu localement correspond à sa valeur locale de j.

Tant que Ind_i est strictement inférieur à j, le processus va continuer son parcours de la liste d'annonce. Deux cas sont alors possibles :

— Le champ *Next* de $ANode_i$ n'est pas vide. Il existe donc un nœud d'annonce succédant à $ANode_i$. Le processus remplace alors $Anode_i$ localement par *Next* et répète la boucle sur le nouveau nœud d'annonce.

— Le champ *next* de $ANode_i$ est vide. $ANode_i$ représente la fin de la liste d'annonce. Le processus initialise alors localement un nouveau nœud d'annonce $Anode_{i+1}$, dont le champ OP_{i+1} est vide, avec un champ *Next* vide, et avec un indice ind_{i+1} correspondant à celui du $ANode_i$ incrémenté de 1. Le processus essaie ensuite de placer l'adresse de $ANode_{i+1}$ dans le champ *Next* de $ANode_i$. Pour ce faire, il essaie d'effectuer un CAS en utilisant une valeur vide comme valeur courante de *Next*, et l'adresse de $Anode_{i+1}$ comme valeur de remplacement. Si ce CAS réussi, c'est que le $ANode_{i+1}$ à bien été placé. Si ce CAS échoue, c'est qu'un

autre processus aura alloué $ANode_{i+1}$ à sa place. Enfin, Le processus effectue une lecture sur $Next$ une nouvelle fois et remplace $ANode_i$ localement par cette valeur et répète la boucle avec le nouveau noeud d'annonce obtenu.

Étant donné que les indices Ind_i de chaque nœud d'annonce $ANode_i$ successifs sont strictement croissants, on fini éventuellement par arriver sur un nœud d'annonce dont l'indice correspond à j.

Une fois arrivé sur le nœud d'annonce réservé $Anode_j$, le processus peut simplement écrire son nœud d'opération $ONode_{local}$ dans le champ OP_j du nœud. Cette étape met à disposition l'opération à appliquer et termine la première phase de l'algorithme.

3.2.2 Exécution et Aide

La seconde phase de l'algorithme consiste pour le processus à parcourir une nouvelle fois la liste d'annonce à partir d'une nouvelle lecture locale de $Head$. Cette étape s'assure que toutes les opérations OP_i disponibles jusqu'à OP_j inclus soient terminées. Le processus conserve à tout moment deux références ; celle du nœud d'annonce courant $ANode_i$, et celle du précédent de ce nœud (s'il existe) $ANode_{i-1}$. Il entre ensuite dans une boucle qu'il ne quitte que si le rang de $ANode_i$ est plus grand que le sien (j).

Pour chaque exécution de la boucle, le processus lit localement le nœud d'annonce $ANode_i$. Il met ensuite à jour le booléen vérifiant la condition de la boucle pour savoir si le rang de $ANode_i$ est supérieur à j. Il vérifie ensuite la valeur de OP_i dans $ANode_i$ avec une lecture. Si le champ est vide, l'opération n'est pas encore disponible. Le processus va alors remplacer $ANode_{i-1}$ par $Anode_i$, et $Anode_i$ par $Anode_{i+1}$ et effectue une nouvelle exécution de la boucle. Cette étape vérifie essentiellement que le processus arrive à la première opération disponible pour lui. Si le champ OP_i n'est pas vide, c'est qu'une opération est disponible et qu'il faut l'appliquer afin d'aider le processus qui la possède à terminer.

L'algorithme d'aide consiste en trois sous étapes simples effectuées dans une boucle infinie :

1. Le processus effectue une lecture locale sur la valeur de *LinNode* et essaie de confirmer les résultats de la dernière opération ayant été appliquée. Il remplace la valeur du champ *Res* de *OP* par celle du champ *Res* de *LinNode*, pour que le processus possédant l'opération puisse en récupérer le résultat. Il change la valeur du champ *Done* de *OP* à *True* afin d'indiquer que *OP* est appliquée et terminée. Ainsi tout processus sait que *OP* doit être retirée de la liste d'annonce.
2. Le processus vérifie que l'opération OP_i de $ANode_i$ est terminée ($Done = True$). Si c'est le cas, le processus sort de la boucle. Sinon elle continue vers la troisième étape.
3. Localement, le processus exécute l'opération de $ANode_i$ sur une copie locale de $LinNode.State$ afin d'en récupérer un nouvel état *NewState* et un résultat local *LocalRes* correspondant au résultat de l'opération exécutée. Il alloue ensuite un nouveau nœud de linéarisation *NewLinNode* avec $OP = OP_i$, $State = NewState$ et $Res = LocalRes$. Ce nœud de linéarisation correspond à un nouvel état possible du système après application de l'opération de $ANode_i$. Enfin le processus essaie d'effectuer un CAS avec l'ancienne valeur de *LinNode* pour la remplacer avec *NewLinNode*. Si le CAS réussi, l'état du système est confirmé. Si le CAS échoue, c'est qu'un autre processus plus rapide à modifié l'état du système avant lui en appliquant une opération (autre ou OP_i). Le processus boucle ensuite à nouveau.

Ce mécanisme d'aide et d'application des opérations est à quelques détails près le même que celui utilisé dans la construction universelle de The Burden of the Past [10]. L'algorithme utilisé étant indépendant de la structure de données et utilisant la même structure de nœud de linéarisation, elle peut être réutilisée.

Une fois cette boucle terminée car l'opération OP_i a été appliquée, le processus essaie de retirer l'opération de la liste d'annonce.

3.2.3 Retrait

Cette dernière phase de retrait consiste à manipuler la liste d'annonce à partir des deux références locales $ANode_{i-1}$ et $ANode_i$. Si ce dernier est le dernier nœud de la liste d'annonce ($Next$ vide), le processus n'essaie pas de retirer, car la liste requiert au minimum un nœud d'annonce pour assurer la validité du compteur k. Sans un nœud d'annonce, les processus ne savent plus se positionner, la liste ne peut donc pas être vide, et c'est pour cela que l'initialisation créer un nœud d'annonce factice. Si $ANode_i.Next$ n'est pas vide, le processus vérifie que $ANode_{i-1}$ ne soit pas vide. S'il l'est, $ANode_i = Head$, il faut alors le conserver pour les raisons expliquées ci-dessus. Si $Anode_{i-1}$ n'est pas vide, le processus effectue une lecture locale sur $ANode_{i-1}.Next$, et essaie ensuite d'effectuer un CAS sur $ANode_{i-1}.Next$ avec la valeur lue et comme valeur de remplacement $ANode_i.Next$. Cette opération à pour but de faire oublier le nœud d'annonce $ANode_i$ lié à OP_i pour retirer définitivement l'opération de la construction universelle. Si le CAS échoue, c'est qu'un autre processus aura réussi à retirer $ANode_i$ avant notre processus. Enfin, le processus récupère la nouvelle valeur de $ANode_i$ en effectuant une lecture sur $ANode_i.Next$ et continue la boucle.

à la fin de cette étape, $ANode_i$ et OP_i auront été retirés de la liste d'annonce. Sans référencement direct, ces espaces mémoire devraient être réclamés par le garbage collector de la machine.

3.2.4 Résumé

L'algorithme divisé en trois parties distinctes assure qu'une fois une opération disponible est appliquée sur l'état du système, celle-ci est marquée comme étant terminée et supprimable. Tous les processus vont alors essayer de la supprimer en retirant le nœud d'annonce lié à l'opération dans la liste d'annonce. Ces opérations garantissent qu'à tout moment, les seuls nœuds d'annonce visibles sont ceux liés à une opération non-disponible car pas encore insérée dans son nœud d'annonce, et ceux liés à une opération disponible non-appliquée. Lorsque aucun processus actif essaie d'utiliser la structure, on estime alors que seul un unique nœud d'annonce est présent dans la liste d'annonce. Son opération est appliquée et terminée, mais il sert de repère pour les futurs processus

voulant utiliser la construction universelle. Il en va de même pour le nœud de linéarisation. La complexité quiescente de la construction universelle est donc supposée constante.

3.3 Algorithmes

Algorithm 1: Nœud d'annonce

```
1 Struct {
2 |   ONode OP_i;
3 |   ANode Next;
4 |   Int Ind_i;
5 } ANode;
```

Algorithm 2: Nœud d'opération

```
1 Struct {
2 |   Operation OP_i;
3 |   Register Res;
4 |   Boolean Done;
5 } ONode;
```

Algorithm 3: Nœud de linéarisation

```
1 Struct {
2 |   ONode OP;
3 |   Register Res;
4 |   Register State;
5 } LNode;
```

Algorithm 4: Structure principale

```
1  Struct {
2      ANode Head;
3      LNode LinNode;
4      Int k;
5  } UniversalConstruction;
```

Algorithm 5: Initialisation

```
1  Function initialize(Register initialState) -> UniversalConstruction :
2      defaultOp ← new ONode {OP_i ← ⊥, Res ← ⊥, Done ← True};
3      Head ← new ANode {OP_i ← defaultOp, Next ← ⊥, Ind_i ← 0};
4      LinNode ← new LNode {OP ← defaultOp, Res ← ⊥, State ←
         initialState};
5      k ← 1;
6      return new UniversalConstruction(k ← k, Head ← Head, LinNode ←
         LinNode)
```

Algorithm 6: Exécution et aide

```
1  Function help(ANode announce, UniversalConstruction construction) :
2      localOperation ← announce.OP_i.get();
3      while True do
4          localLin ← construction.LinNode.get();
5          localLin.OP.Res.set(localLin.Res);
6          localLin.OP.Done.set(True);
7          if localOperation.Done.get() then
8              return ;
9          ⟨localState, localRes⟩ ← execute(localLin.State.get(),
             localOperation.OP_i);
10         newLin ← new LNode {State ← localState, Res ← localRes, OP
             ← localOperation};
11         construction.LinNode.CAS(localLin, newLin);
```

Algorithm 7: Appel de l'opération

```
1  Function invoke(Operation op, UniversalConstruction construction) ->
     Register :
2      newOp ← new ONode {OP_i ← op, Res ← ⊥, Done ← false};
3      rank ← construction.k.fetchAndIncrement();
4      currentNode ← construction.Head.get();
5      while currentNode.Ind_i < rank do
6          if currentNode.Next.get() = ⊥ then
7              newNext ← new ANode {OP_i ← ⊥, Next ← ⊥, Ind_i
                 ← currentNode.rank + 1};
8              currentNode.Next.CAS(⊥, newNext);
9          currentNode ← currentNode.Next.get();
10     currentNode.OP_i.set(op);
11     start ← construction.Head.get();
12     prev ← ⊥;
13     selfDeleted ← False;
14     while ¬(selfDeleted) do
15         currentNode ← start;
16         selfDeleted ← (currentNode.Ind_i ≥ rank);
17         next ← currentNode.Next.get();
18         currentOp ← currentNode.OP_i.get();
19         if currentOp = ⊥ then
20             start ← next;
21             prev ← currentNode;
22         else
23             help(currentNode, construction);
24             if next ≠ ⊥ then
25                 if prev ≠ ⊥ then
26                     prev.Next.CAS(currentNode, next);
27                 else
28                     prev ← currentNode;
29                 start ← next;
30     return New Register(newOp.Res.get());
```

4. Preuves

4.1 L'algorithme est Wait-Free

Lemme 1 *Aucun appel à $help()$ dans l'algorithme de construction universelle ne prend un nombre infini d'étapes.*

Preuve 1 *Supposons par l'absurde qu'un appel à $help(ANode_i)$ effectué par un processus p_i prenne un nombre infini d'étapes. Soit Ind_i le rang du nœud d'annonce $ANode_i$. Sans pert de généralité, on peut supposer que Ind_i soit minimal parmi tous les appels infinis de $help(ANode_j)$. Soit OP_i l'opération attachée à $ANode_i$.*

D'après l'algorithme, $help(ANode_i)$ boucle à partir de $localLin \lrcorner LinNode.get$ (ligne 3). S'il n'y a pas eu de linéarisation avec OP_i, alors $OP_i.Done = False$ en ligne 7, et ce pour toute exécution. Un CAS ne peut être effectué avec OP_i dans un nœud de linéarisation que si $OP_i.Done = False$. Mais toute réussite d'un CAS (ligne 11) est suivie d'une écriture du résultat présent dans $LinNode.Res$ dans $OP_i.Res$, et de la mise à vrai du booléen de terminaison dans $OP_i.Done$. Donc après un CAS réussi, $OP_i.Done = True$. Or si $help(ANode_i)$ est infini, le CAS n'est jamais réussi par aucun processus et $OP_i.Done$ reste faux.

Maintenant, regardons les processus qui pourraient précéder p_i. Lors de l'appel à $invoke()$, un processus p_j récupère son indice Ind_j (ligne 3) tel que $Ind_j < Ind_i$ dans un nombre fini de cas, car Ind_i est minimal parmi les appels à $help()$. Donc, un nombre fini k de processus p_j ont pu avoir un $ANode_j$ avec $Ind_j < Ind_i$, et tous terminent, par minimalité de Ind_i. Donc, après un certain point, tous les nouveaux $ANode_j$ créés auront $Ind_j > Ind_i$.

On remarque que plus personne n'effectue de CAS dans $ANode_j.Next$ avec $Ind_j < Ind_i$. Donc la liste d'annonce se stabilise avant $ANode_i$. Dans tous les appels à $help(ANode_j)$

pour $Ind_j < Ind_i$, $ANode_j.Next$ finit par pointer vers $ANode_i$, donc le prochain appel à $help()$ s'effectue sur $ANode_i$. Or, chaque appel à $help()$ essaye un CAS avec OP_i. Comme OP_i est toujours la même instance, seul un CAS avec OP_i dans $LinNode.OP$ peut réussir. Donc un processus finit obligatoirement par réussir un CAS avec OP_i, et met le booléen de terminaison $OP_i.Done$ à vrai, ce qui est en contradiction avec le fait que $OP_i.Done = False$ pour toujours.

Par l'absurde, il n'existe pas d'exécution de $help()$ possédant un nombre infini d'étapes.

Lemme 2 *L'algorithme de construction universelle est wait-free.*

Preuve 2 *Considérons un appel à invoke() par un processus p_i.*

Ce processus étend si nécessaire une liste de nœuds d'annonce un nombre fini de fois. Les rangs Ind_i de ces nœuds sont strictement croissants, et la boucle s'arrête lorsque le nœud courant possède un rang égal à l'indice i récupéré par p_i sur k (ligne 5).

Il exécute ensuite une autre boucle jusqu'à ce que $self.Deleted = True$ (ligne 14). A chaque itération, il essaie de faire avancer la complétion des opérations disponibles avec des appels à $help()$ (ligne 23). Selon le lemme 1, tout appel à $help()$ termine.

L'opération linéarisée et terminée dans le $help()$ permet de mettre la valeur de $OP_j.Done$ à vrai. Or comme les appels successifs à $help()$ finissent par éventuellement l'appeler sur OP_i, l'opération sera alors aussi linéarisée et terminée avec $OP_i.Done = True$. A la prochaine itération de la boucle (ligne 14), p_i sortira de celle-ci.

Donc Aucune boucle dans l'algorithme ne peut tourner indéfiniment : L'insertion d'une opération dans la liste d'annonce s'effectue en un nombre fini d'étapes. Les appels à $help()$ terminent selon le lemme 1. Chaque opération termine, que ce soit grâce à l'activité des autres processus ou grâce à l'activité de son propre processus.

Chaque appel à invoke() se termine en un nombre fini d'étapes, l'algorithme de construction universelle proposé est wait-free.

4.2 L'algorithme est Linéarisable

Lemme 3 *La construction universelle proposée est linéarisable.*

Preuve 3 *Soit α une exécution admissible par l'algorithme. Observons d'abord que, pour toute opération $invoke(op_i)$ invoquée par un processus p_i, il existe au plus un seul nœud de linéarisation l_i tel que $l_i.OP.OP_i = op_i$ et qu'une invocation de CAS sur l_i retourne vrai. Il est important de noter que tous les nœuds de linéarisation créés sont uniques. De plus, il n'existe qu'un seul nœud d'opération OP_i construit par p_i tel que $operation(OP_i) = op_i$.*

Supposons par contradiction deux nœuds de linéarisation l_j et l_k. $l_j.OP = l_k.OP$, et les deux sont écrits avec succès dans $LinNode$ par p_j et p_k. Considérons, deux nœuds de linéarisation l_i et l_m, tel que l_m soit le nœud succédant à l_i grâce à un appel du processus p_m à CAS(l_i,l_m) réussi. Le processus p_j lis d'abord localement le nœud de linéarisation courant $LinNode$, puis vérifie que l'opération qu'il souhaite appliqué ne soit pas terminée.p_m écrit quand à lui True dans OP_i.Done avant d'invoquer un CAS proposant l_i et l_m comme arguments. Le processus est donc au moins aussi ancien que l_i. Or il Il est impossible que $LinNode = l_i$ car cela signifierait que $p_j = p_m$ et qu'ils aient exécutés les opérations dans l'ordre inverse.

Définissons maintenant le point de linéarisation de toute opération $invoke(op_i)$ comme étant, s'il existe, l'unique invocation réussie de $LinNode.CAS(l'_i, l_i)$. Comme p_i a terminé, cela signifie qu'un processus p_j a marqué son opération comme terminée après avoir lu l_i. Cette lecture ne peut se produire qu'après qu'un processus p_k ait écrit l_i. Cela constitue donc un point de linéarisation pour $invoke(op_i)$. Comme nous l'avons vu, ce point de linéarisation a lieu avant la terminaison de p_i. Il s'est également produit après l'invocation de p_i, car op_i ne peut être créé que par ce processus.

Enfin,les états et valeurs de résultat atteints dans une exécution séquentielle E, définie par l'ordre de linéarisation, sont les mêmes que ceux écrits dans les nœuds de linéarisation. Si le processus p_i retourne Res_i à la fin de l'exécution de $invoke(op_i)$, c'est qu'il l'a lu après avoir vu que l'opération était terminée. Or, cela ne peut se produire que si un processus l'a inscrit comme terminé, après avoir écrit son résultat dans le nœud d'opération à partir du nœud de linéarisation. Par conséquent, p_i retourne la même valeur que dans E. Donc α est linéarisable.

4.3 L'algorithme possède une complexité quiescente constante

Lemme 4 *L'algorithme de construction universelle possède une complexité en mémoire constante lorsque aucun processus n'est activement en train de l'utiliser. Cette notion est appelée complexité quiescente.*

Prédicat 1 *D'après l'algorithme, chaque processus p_i peut insérer des nœuds d'annonce dans la liste d'annonce pour l'étendre si nécessaire. A chaque itération de sa boucle d'aide, tout processus p_i aide les opérations précédentes à terminer. Il essaie ensuite de supprimer les nœuds d'annonces dont les opérations sont marqués comme étant terminés, et ce jusqu'à atteindre sa propre opération OP_i dans le nœud d'annonce $ANode_i$. Un nœud d'annonce est effectivement retiré de la liste lorsqu'un CAS sur le suivant du nœud d'annonce précédent celui actuellement lu par p_i remplace le pointeur vers ce nœud lu par le pointeur vers son propre successeur.*

Donc chaque processus dont l'opération termine voit son nœud d'annonce supprimé de la liste d'annonce si celui-ci n'est pas le dernier ou seul élément de la liste d'annonce.

Preuve 4 *D'après le prédicat 1, Chaque processus p_i ayant invoqué une opération o_i avec un ordre de rang Ind_i supprime toutes les opérations précédentes terminées avec un rang inférieur à Ind_i, donc les nœuds d'annonce associés à des opérations o_j, $j < i$. Cette suppression s'effectue avant que le processus p_i ne se supprime lui-même de la liste d'annonce s'il en a la possibilité (il n'est pas le dernier ou l'unique nœud de la liste) Donc tout processus p_j ayant invoqué une opération o_j avec $j < i$ et donc $Ind_j < Ind_j$, et qui n'a pas terminé son opération avant p_i, sera supprimé, au plus tard, lorsque p_j aura terminé son opération.*

Supposons qu'il y ait n opérations distinctes, déclarées avec la capture d'un rang sur k. Il existe un ordre total sur les rangs des opérations. Une fois que tous les processus ont terminés leurs opérations, le processus ayant le rang capturé le plus élevé sera le dernier à rester dans la liste d'annonce à cause des contraintes sur l'algorithme (voir Section 3.2.3).

Une fois toutes les opérations terminées, tous les nœuds correspondant aux opérations ayant un rang Ind_j strictement inférieur à l'opération de plus haut rang $Ind_i = k - 1$ seront supprimés. Il ne restera que le dernier nœud d'annonce lié à Ind_i

Le nombre de nœuds dans la liste de nœuds d'annonce est donc limité à un seul nœud en régime quiescent, nécessaire pour le bon fonctionnement de l'algorithme. L'occupation en mémoire de la liste est donc constante sous ce régime. A tout moment, le compteur k reste présent et conserve un espace mémoire constant (n bits). Il en va de même pour le nœud de linéarisation, composé de trois champs dont la complexité en mémoire est constante, et qui possède donc elle-même une complexité constante. La complexité quiescente de l'algorithme est donc une constante définie par $\theta(taille(ANode_{k-1}) + taille(k) + taille(LinNode))$

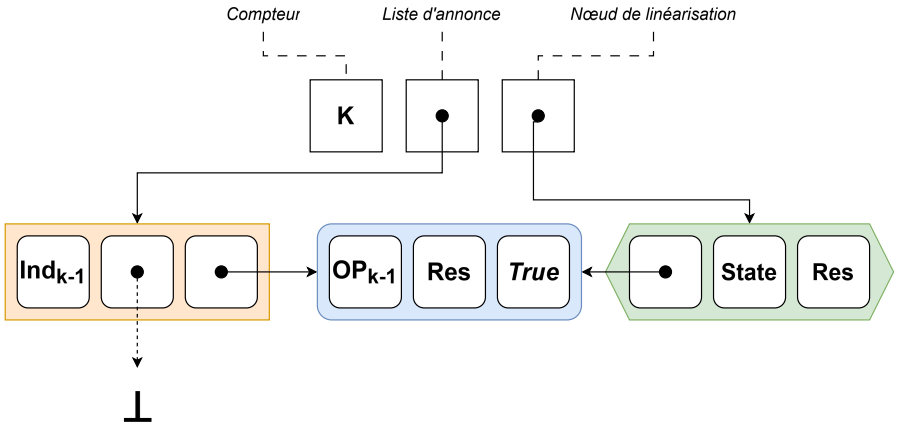

FIGURE 4.1 – État de la structure de données de l'algorithme sous régime quiescent

5. Conclusion

Récapitulatif

Dans cet article nous avons étudié les problèmes de la complexité en mémoire de certains algorithmes de construction universelles wait-free dans le modèle d'arrivé infini. Ceux ne se basant que sur l'instruction compare-and-set doivent garder une trace du passage de tous les processus utilisant l'algorithme, ce qui résulte en une complexité strictement croissante.

Nous avons proposé un nouvel algorithme de construction universelle se basant sur les instructions compare-and-set ainsi que fetch-and-increment afin de résoudre ce problème. Cet algorithme utilise une liste chaînée de nœuds d'annonce, contenant les informations nécessaires à l'exécution des opérations. Cette liste établit un ordre total sur les opérations selon un rang que chaque processus récupère via un compteur partagé, incrémenté de 1 pour chaque processus souhaitant déclarer une opération. Les nœuds d'annonce successifs possèdent donc un indice j strictement croissant.

Grâce à un mécanisme d'aide et à des nœuds de linéarisation, nous avons prouvés que l'algorithme est à la fois wait-free en assurant la terminaison de toute opération disponible, et il est linéarisable grâce au nœud de linéarisation unique, ne pouvant être remplacé que par un compare-and-set qui ne peut être gagné que par un unique processus à la fois. Cette décision représente le point de linéarisation de l'algorithme.

Enfin nous avons prouvé que l'algorithme possède une complexité quiescente constante. Lorsque aucun processus n'est actif dans l'algorithme, la complexité en mémoire de l'algorithme ne compte qu'un unique nœud d'annonce dont l'opération est déjà terminée, nécessaire au fonctionnement de l'algo-

rithme, le compteur k représenté par un registre sur n bits, et le nœud de linéarisation.

Problèmes de l'algorithme

Bien que l'algorithme soit wait-free, la complexité temporelle de celui-ci peut être importante dans le cas ou des opérations lourdes sont effectuées. On peut imaginer un scénario dans lequel la liste d'annonce ne fait que croître, forçant chaque nouveau processus à venir s'insérer toujours plus loin dans celle-ci. En pratique ce scénario n'est pas la norme et l'algorithme convient à la plupart des structures de données à implémenter de façon concurrente.

Un autre problème est celui de la valeur maximum du compteur k. Celui-ci, responsable du bon fonctionnement de l'ordonnancement des nœuds d'annonce est un entier représenté par un registre unique sur n bits. On estime donc que pour une machine fonctionnant sur des registres à 32-bits, le nombre d'opérations maximum pouvant être insérés est de $2^{32} - 1$ opérations. Pour une machine fonctionnant sur des registres à 64-bits, ce nombre passe à $2^{64} - 1$ opérations. k pourrait aussi être représenté par un entier dont le nombre de bits est supérieur à 64. Des structures de données spécifiques existent pour ce besoin. Dans ce cas, la complexité en mémoire de l'algorithme dépend de l'espace alloué dynamiquement durant l'exécution pour obtenir le nombre souhaité. On estime alors que la complexité quiescente n'est plus constante, mais de l'ordre de $\theta(log(n))$ selon l'espace alloué pour k.

Lien avec l'instruction Memory-To-Memory-Swap

L'objectif initial de nos recherches était de trouver une implémentation concrète de l'instruction Memory-To-Memory-Swap, qui consiste à échanger de façon atomique le contenu de deux registres partagés. La difficulté de la tâche nous a menés à concevoir une construction universelle résolvant les mêmes problèmes que cette instruction. Il est donc techniquement possible d'utiliser cette construction universelle afin d'implémenter une version logicielle de l'instruction Memory-To-Memory-Swap. L'implémentation matérielle de cet instruction reste cependant toujours un défi.

Bibliographie

[1] M. P. Herlihy and J. M. Wing, "Linearizability : A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[2] L. Lamport, "A new solution of dijkstra's concurrent programming problem," in *Concurrency : the works of leslie lamport*, 2019, pp. 171–178.

[3] R. K. Treiber *et al.*, *Systems programming : Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research ..., 1986.

[4] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 267–275.

[5] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[6] Y. Afek, E. Gafni, and A. Morrison, "Common2 extended to stacks and unbounded concurrency," in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, 2006, pp. 218–227.

[7] K. Censor-Hillel, E. Petrank, and S. Timnat, "Help!" in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, 2015, pp. 241–250.

[8] M. Raynal, "Distributed universal constructions : a guided tour," Ph.D. dissertation, IRISA, 2016.

[9] M. Perrin, A. Mostefaoui, and G. Bonin, "Extending the wait-free hierarchy to multi-threaded systems," in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 21–30.

[10] D. Bédin, F. Lépine, A. Mostéfaoui, D. Perez, and M. Perrin, "Wait-free cas-based algorithms : The burden of the past," in *35th International Sym-*

posium on Distributed Computing (DISC 2021). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, pp. 11–1.

[11] M. Merritt and G. Taubenfeld, "Computing with infinitely many processes : Under assumptions on concurrency and participation," in *International Symposium on Distributed Computing*. Springer, 2000, pp. 164–178.

[12] T. G. Merritt, Michael, "Resilient consensus for infinitely many processes," in *Distributed Computing : 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003. Proceedings 17*. Springer, 2003, pp. 1–15.