Les objets ajustés : Une approche bien fondée et efficace pour la programmation concurrente

Boubacar Kane

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2025IPPAS001

Thèse de doctorat

Les objets ajustés: Une approche bien fondée et efficace de la programmation concurrente

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 Institut Polytechnique de Paris (IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 10 Janvier 2025, par

Boubacar KANE

Composition du Jury :

Vania Marangozova Professeur, Université de Grenoble-Alpes (LIG, ERODS)	Rapportrice
Davide Frey Professeur, (INRIA, WIDE)	Rapporteur
Julien Sopena Maître de Conférences, Sorbonne Université (LIP6, DELYS)	Examinateur
Petr Kuznetsov Professeur, Télécom Paris (INFRES, ACES)	Examinateur
François Trahay Professeur, Télécom SudParis (SAMOVAR/BENAGIL)	Examinateur
Pierre Sutra Maître de Conférences, Télécom SudParis (SAMOVAR/BENAGIL)	Directeur de thèse

Remerciements

Je souhaite commencer par exprimer ma gratitude envers toutes les personnes qui ont lu ou liront cette thèse, y compris celles qui ne parcourront que la section des remerciements.

Je remercie sincèrement les rapporteurs, Vania Marangozova et Davide Frey, pour la lecture attentive de ce manuscrit et la qualité de leur rapport, particulièrement détaillé et enrichissant.

Je tiens également à exprimer ma reconnaissance envers Petr Kuznetsov et Julien Sopena, qui ont suivi mon travail non seulement en tant qu'examinateurs, mais aussi tout au long de la thèse, depuis la soutenance de mi-parcours jusqu'au comité de suivi individuel.

Un immense merci à Denis, sans qui cette thèse n'aurait pas vu le jour. Tu as été un soutien précieux, tant pour le travail scientifique que pour tes conseils avisés concernant la rédaction des rapports et la préparation des présentations.

Je souhaite adresser une reconnaissance toute particulière à Pierre. Non seulement l'idée de ce sujet de recherche vient de toi, mais tu m'as surtout consacré d'innombrables heures de rendez-vous, qui, j'en suis certain, ont joué un rôle essentiel dans ma motivation. J'espère ne pas avoir épuisé toute ta patience et que de nombreux autres doctorants pourront bénéficier de ta disponibilité exemplaire.

Cela en surprendra peut-être certains, mais je n'ai pas été très souvent présent au laboratoire durant ces SIX (!) années. Cependant, je tiens à remercier chaleureusement les membres du laboratoire pour avoir égayé ces moments passés sur place. Un merci particulier à Gaël pour sa bonne humeur constante, à François pour tous les conseils qu'il a pu distiller, ainsi qu'à Mathieu, Élisabeth, Sophie, Amel B., Amel M., Julien, Mohamed et Chantal.

Je souhaite également remercier les autres doctorants, car partager des soupirs

de découragement après la question : "Alors, comment avancent les recherches ?" nous a permis de nous sentir moins seuls.

Je commencerai par ceux qui ont déjà soutenu et qui, pour certains, ont été des modèles : Alexis C., Alexis L., Nabila, Pedro et Anatole.

Un remerciement particulier à Tuanir, avec qui j'ai partagé le même bureau et le même directeur de thèse. Je me souviens encore de ces moments où nous tentions de deviner l'humeur de Pierre avant une réunion !

Je remercie également Yohan, qui m'a toujours semblé être le doctorant modèle, capable d'apporter un avis constructif sur tous les sujets. Au-delà des discussions scientifiques qui m'ont aidé à progresser dans mes expériences, j'ai particulièrement apprécié nos échanges plus généraux lors de mes trop rares visites au laboratoire.

Un grand merci à Suba. Je ne sais pas si c'est toi qui as influencé Gaël ou l'inverse, mais votre duo de bonne humeur a largement contribué à rendre mes passages au laboratoire plus agréables.

Je souhaite également encourager et remercier les futurs docteurs : Marie, Mickaël, Jean-François, Catherine, Jules, Jana, Nevena et Eric. Si cette thèse a pu être menée à terme, vous pouvez être confiants dans votre propre réussite.

Je ne peux pas clore cette section sans remercier Adam Chader. J'insiste sur le nom, car, à en croire les faits, j'ai pratiquement tout appris à ce jeune chercheur. Je vous invite à vérifier cette affirmation dans le paragraphe qui me sera sans doute dédié dans sa propre thèse. Plus sérieusement, nos discussions, souvent éloignées de nos sujets de recherche respectifs, ont été précieuses. Continue à apporter de la bonne humeur là où tu passes : le prochain laboratoire qui t'accueillera en post-doc ou en tant que Maître de conférences aura bien de la chance.

Je remercie également Marie Degli-Esposti pour m'avoir grandement facilité la gestion administrative de cette thèse.

Pour conclure, je souhaite exprimer toute ma gratitude à mes proches.

À ma femme Marie, qui, lassée d'attendre la fin de cette thèse, m'a offert une petite fille - Imany - lors de ma dernière année, me forçant ainsi à avoir la motivation ultime pour terminer.

À ses enfants, Nathanaël et Camille, qui doivent probablement me voir comme le plus grand flemmard de la planète.

À nos amis Marie-Claire et John, pour leurs invitations estivales à la piscine et

aux soirées jeux de société, m'aidant à mieux reprendre le travail chaque septembre.

À mes frères et sœurs Fatou, Abdoulaye, Awa et Djibril, ainsi qu'à mes amis d'enfance Karim et Khadim, qui ont sans doute cru pendant six ans que j'étais au chômage.

Enfin, à mes parents Amady et Sokhna, qui ont su serré la vis suffisamment longtemps pour me rendre autonome et me permettre d'en arriver là aujourd'hui.

Une dernière fois, merci à tous.

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Au cours des deux dernières décennies, la fréquence d'horloge des processeurs a cessé de croître de manière significative. Depuis que la barre du 1 GHz a été franchie au début des années 2000, les processeurs modernes ont atteint une vitesse de seulement quelques GHz. En revanche, la loi de Moore reste valable : l'industrie continue d'augmenter le nombre de transistors intégrés sur une même puce. Cela a permis l'émergence et la généralisation des systèmes multicœurs, qui peuvent désormais prendre en charge des dizaines, voire des centaines de threads matériels. Ces systèmes sont omniprésents, des grands serveurs aux dispositifs embarqués les plus compacts.

Exploiter pleinement cette puissance de calcul exige des développeurs qu'ils conçoivent des programmes parallèles. Idéalement, les performances doivent évoluer de manière linéaire avec le nombre de cœurs disponibles ; on parlera de scalabilité d'une application. Or, maîtriser le parallélisme reste un défi majeur pour tout développeur, ceci notamment pour deux raisons. D'une part, il est difficile d'utiliser pleinement les ressources matérielles disponibles. Cette difficulté est notamment induite par la différence d'abstraction (voir son inadéquation) entre le matériel et le logiciel. D'autre part, les raisonnements liés à la concurrence sont complexes. En particulier, les entrelacements des opérations augmentent de manière exponentielle avec le nombre de cœurs. Maîtriser le parallélisme révèle donc

de l'art de trouver un juste équilibre entre correction et performance [28].

1.2 Motivation

Pour maximiser les performances, il est souvent préférable de permettre un haut degré de parallélisme sans imposer des synchronisations excessives, bien que cela augmente le risque d'interférences entre threads. Identifier les goulots d'étranglement dans un programme est une tâche ardue, car les causes peuvent être multiples, allant d'implémentations logicielles inefficaces à des contraintes matérielles telles que la vitesse d'accès à la mémoire ou une mauvaise répartition des données dans un système distribué.

Pour aider le développeur dans cette tâche, il est essentiel d'avoir un support logiciel pour la programmation parallèle. Á cette fin, des bibliothèques existent dans de nombreux langages de programmation. Ainsi, des bibliothèques, comme `Boost.LockFree` en C++ ou `java.util.concurrent` pour Java, fournissent des outils pour exprimer le parallélisme et la concurrence de manière élégante et efficace. Ces bibliothèques incluent des mécanismes de gestion de la concurrence comme des objets partagés linéarisables, des verrous et barrières, ou encore des primitives de synchronization telles que *compare-and-swap*.

Cependant, ces bibliothèques sont génériques. En effet, les abstractions qu'elles proposent ont une interface étendue afin de satisfaire un grand nombre de développeurs et de couvrir de nombreux cas d'usage. Or, ceci peut nuire aux performances, en particulier dans les scénarios où la nécessité de passer à l'échelle est cruciale. Le point de départ de cette thèse est la question de savoir s'il est intéressant de spécialiser ces objets afin de les rendre plus performants.

1.3 Contributions

Afin de répondre à cette question, nous avons conduit une analyse de l'utilisation des objets partagés à travers de nombreux projets open source. Notre étude révèle des tendances significatives et des axes d'améliorations potentiels. Ainsi, bien que les objets partagés soient relativement rares dans le code source global,

notre étude montre qu'ils sont essentiels pour tirer pleinement parti des architectures multicœurs modernes. L'analyse des appels de méthodes a mis en évidence que seules certaines fonctionnalités de ces objets sont fréquemment utilisées, ce qui ouvre la voie à des optimisations ciblées.

Pour répondre à ce constat, nous avons introduit le concept d'**objets ajustés**, des objets partagés spécialement conçus pour offrir des performances optimales dans des contextes d'exécution spécifiques. Nous avons formalisé ce concept à l'aide du **graphe d'indistinguabilité**, un outil théorique qui mesure la capacité d'un objet à évoluer avec le nombre de threads. Ce graphe permet de comparer différents objets partagés pour choisir celui qui offre les meilleures performances selon le scénario.

Nous avons également développé `DEGO`, une bibliothèque d'objets ajustés qui applique ces principes. `DEGO` propose des collections partagées et d'autres objets courants, dont l'interface et les comportements sont ajustés pour maximiser l'efficacité dans des environnements parallèles. Ces objets exploitent des ajustements spécifiques pour améliorer la scalabilité, tout en réduisant les coûts de synchronisation.

Pour valider l'efficacité de `DEGO`, nous avons effectué des évaluations rigoureuses. Nous avons utilisé des micro-benchmarks, inspirés de *Synchrobench* [21], ainsi qu'une application de type réseau social, *Retwis* [69], qui simule un environnement proche de celui de Twitter. Ces expériences ont démontré l'avantage significatif des objets ajustés dans des situations où la gestion efficace des ressources concurrentes est cruciale, confirmant ainsi la pertinence de notre approche pour l'optimisation des programmes parallèles.

1.4 Plan

Le reste de la thèse est structuré comme suit :

— Le Chapitre 2 aborde les défis relatifs à la conception de programmes parallèles sur des architectures multicœurs modernes et explore des stratégies de communication optimisées pour améliorer les performances. Nous y présentons divers outils permettant de générer et de synchroniser le parallélisme,

illustrés à travers des exemples en Java. Plus loin, nous rappelons les principaux critères de cohérence et les conditions de progrès nécessaires pour garantir un comportement correct des objets partagés et assurer la bonne exécution d'un programme. Enfin, nous discutons de solutions permettant d'adapter ces critères de cohérence en fonction des besoins applicatifs et présentons des outils destinés à aider les développeurs dans la création de programmes parallèles.

— Le Chapitre 3 présente le principe d'ajustement, qui vise à concevoir des objets partagés optimisés pour des usages spécifiques afin d'améliorer les performances des programmes. Nous débutons par une analyse approfondie de l'utilisation des objets partagés dans plusieurs bases de code Java. Ensuite, nous introduisons le graphe d'indistinguabilité, un outil théorique permettant d'évaluer la scalabilité d'un objet partagé. Nous formalisons ensuite le principe d'ajustement et illustrons ses applications avec plusieurs exemples.

— Le Chapitre 4 décrit la bibliothèque d'objets ajustés `DEGO`. Ce chapitre évalue les objets de `DEGO` à travers plusieurs microbenchmarks ainsi qu'au travers d'une application de type réseau social.

— Le Chapitre 5 conclut la thèse en résumant nos travaux, en exposant les difficultés rencontrées, et en proposant des pistes pour de futures avancées.

Chapitre 2

Contexte et Motivation

Face à l'augmentation constante des besoins en puissance de calcul, l'usage des programmes parallèles s'est considérablement intensifié pour répondre aux exigences de performance. Les avancées dans le design des processeurs et des architectures distribuées incarnent des solutions innovantes pour relever ces défis, en proposant des structures matérielles et logicielles adaptées à l'exécution simultanée de multiples tâches. Cette évolution vers le parallélisme constitue une étape cruciale pour optimiser l'exploitation des ressources disponibles et accélérer le traitement des données.

Au cœur de ces programmes parallèles, les objets partagés jouent un rôle fondamental. Ils offrent aux développeurs un ensemble d'outils simples et variés pour concevoir des programmes capables d'exécuter des opérations simultanées de manière cohérente.

Dans cette thèse, nous illustrons nos analyses et expérimentations en nous appuyant sur les objets partagés de la bibliothèque Java. En tant que l'un des langages les plus utilisés dans le monde, Java offre une infrastructure mature pour développer des applications parallèles et concurrentes. Sa vaste bibliothèque `java.util.concurrent`, incluant des structures de données variées et des mécanismes de synchronisation performants, permet d'explorer de manière approfondie les différentes stratégies de gestion du parallélisme.

Ce chapitre explore en profondeur l'architecture des processeurs et leur évolution pour comprendre comment les choix de conception et les techniques d'optimi-

sation permettent d'accroître les performances des programmes modernes. Dans la Section 2.1 Nous nous intéressons particulièrement au passage des architectures monocœur aux architectures multicœurs et distribuées.

Pour comprendre comment le parallélisme est mis en œuvre, nous étudions dans la Section 2.2 les deux stratégies de communication principales dans les systèmes distribués : la mémoire partagée et le passage de messages, tant dans leurs implémentations logicielles que matérielles.

La discussion se poursuit en Section 2.3 avec une exploration des mécanismes de parallélisation, abordant divers outils de gestion des tâches concurrentes, comme les objets promesses, les pools de threads ou encore les threads virtuels. À cela s'ajoutent des techniques de synchronisation essentielles, notamment les verrous et les sémaphores, pour garantir la sécurité des accès en situation de concurrence.

Enfin, nous abordons en Section 2.4 les conditions de progrès qui garantissent la terminaison des programmes ainsi que les propriétés de correction essentielles à la fiabilité des programmes concurrents, telles que la *linéarisabilité*, la cohérence séquentielle ou encore la cohérence causale.

Le chapitre se conclut en Section 2.5 par l'exploration de modèles de cohérence flexibles, permettant d'ajuster les critères de cohérence en fonction des cas d'utilisation pour maximiser les performances. Nous présentons également des outils qui aident les développeurs à évaluer la scalabilité de leurs applications ou à tester les performances de leurs objets partagés, offrant ainsi des moyens de mieux anticiper le comportement de leurs applications dans un contexte parallèle.

2.1 Les bases du parallélisme

Fonctionnement d'un ordinateur

Les informations au sein d'un processeur sont représentées par une série de nombres binaires. Pour exprimer ces nombres, on fait usage de **transistors**. Le **codage NRZ** est employé pour désigner un 0 ou un 1, en fonction de la tension électrique qui traverse le transistor, cette tension étant soit inférieure à un certain seuil, soit supérieure à un autre seuil. On dit alors que le transistor se ferme ou s'ouvre, et que cette action constitue un **cycle**. Un processeur cadencé à 3 GHz

est capable d'exécuter 3 milliards de cycles en une seconde. Ces transistors sont ensuite regroupés au sein de **circuits**.

Tous ces circuits peuvent fonctionner à des vitesses différentes, en fonction du temps nécessaire à la propagation de la tension électrique. Pour synchroniser ces circuits et prévenir tout problème d'interaction susceptible de provoquer des dysfonctionnements, la durée entre deux mises à jour des circuits est maintenue constante et ajustée en fonction du circuit le plus lent. Ainsi, tous les circuits reçoivent simultanément un signal d'écriture, que l'on appelle le **signal d'horloge**.

Différents circuits sont regroupés pour former des sous-unités qui constituent la mémoire ou le processeur, comme par exemple l'**unité de contrôle** ou encore l'**unité d'exécution**. La première a pour rôle de gérer les instructions, tandis que la seconde exécute les opérations de calcul et traite les données.

Un processeur exécute des requêtes qui sont décomposées en calculs simples, appelés **instructions**. L'exécution d'une instruction est généralement divisée en plusieurs étapes, chacune durant un cycle d'horloge. Ces étapes peuvent être résumées ainsi : l'**extraction**, qui consiste à récupérer l'instruction depuis la mémoire ; le **décodage**, qui permet d'identifier l'opération ainsi que les opérandes ; l'**exécution** de l'instruction ; et enfin l'**écriture des résultats** en mémoire, laquelle sera définie plus en détail dans cette section. La rapidité avec laquelle un processeur exécute une instruction dépend donc du nombre de cycles qu'il peut effectuer par seconde et de la durée de chaque cycle.

Augmenter le nombre de transistors a longtemps été la solution privilégiée pour améliorer les performances des processeurs. En 1965, Gordon E. Moore, cofondateur de la société *Intel*, a fait une observation qui sera plus tard connue sous le nom de "**Loi de Moore**". Cette loi est une observation empirique stipulant que le nombre de transistors sur une puce double environ tous les deux ans. Par la suite, cette loi a été étendue à la "vitesse" ou à la fréquence d'horloge des processeurs. Bien que le nombre de transistors sur une puce continue encore aujourd'hui de doubler, la fréquence d'horloge a atteint un plateau depuis le milieu des années 2000. Il s'agit d'une limitation physique qui empêche les fabricants de maintenir une croissance exponentielle de la fréquence des processeurs. En effet, augmenter le nombre de cycles par seconde entraîne une hausse de la chaleur produite, pour deux raisons principales : d'une part, l'augmentation de la puissance électrique fournie

au processeur, et d'autre part, la réduction du temps disponible pour dissiper cette chaleur.

Parallélisme matériel

Étant donné que la fréquence ne peut plus être augmentée rapidement, d'autres solutions ont dû être trouvées pour continuer à améliorer les performances des processeurs. Pour rappel, les instructions sont décomposées en différentes étapes, et ces étapes sont souvent effectuées par des sous-unités distinctes, ce qui permet de les exécuter en parallèle. Cette technique est appelée **pipelining**.

Imaginons quatre instructions, chacune décomposée en quatre étapes : e_1, e_2, e_3, e_4. Examinons combien de cycles sont nécessaires pour traiter ces instructions avec et sans *pipelining*. Supposons qu'il existe quatre sous-unités spécifiques, notées su_1, su_2, su_3, su_4, capables de traiter respectivement les étapes e_1, e_2, e_3 et e_4. Lorsque les instructions sont exécutées les unes après les autres, il est nécessaire d'attendre que la première instruction soit entièrement traitée avant de commencer la seconde. Par conséquent, si chaque étape d'une instruction est traitée en un cycle, il faut 16 cycles pour traiter les quatre instructions sans *pipelining*. Avec *pipelining*, dès que l'étape e_1 de la première instruction est terminée par la sous-unité su_1, l'étape e_1 de la deuxième instruction peut immédiatement être traitée par cette même sous-unité, tandis que l'étape e_2 de la première instruction est traitée simultanément par la sous-unité su_2. Ainsi, le traitement des quatre instructions avec *pipelining* nécessite seulement 7 cycles.

Le *pipelining* a incité les fabricants de processeurs à adapter leurs puces, en isolant les sous-unités et en insérant de la mémoire entre elles afin de les séparer. On qualifie ces processeurs de **superscalaires** lorsque les instructions sont distribuées de manière dynamique entre les sous-unités, avec une gestion des dépendances des données directement assurée par le processeur. À l'inverse, les processeurs dits **VLIW** (*Very Long Instruction Word*) se caractérisent par un ordre d'exécution des instructions fixé à l'avance, sans prise en charge dynamique des dépendances de données [12, 23]. Le nombre d'instructions qu'un processeur peut traiter simultanément définit ce que l'on appelle la **profondeur du pipeline**. Toutefois, le parallélisme des instructions rencontre certaines limites, en particulier une sur-

charge architecturale liée à l'ajout de nouvelles sous-unités, ainsi qu'un rendement décroissant, car le nombre d'instructions indépendantes pouvant être exécutées simultanément n'augmente pas toujours proportionnellement.

Des limites physiques apparaissent lorsque l'on se limite à un seul processeur sur une puce, notamment en raison des contraintes liées à sa taille, qui restreignent le nombre de connexions physiques pouvant être établies avec la mémoire. Cette limitation se traduit par une contrainte sur la bande passante maximale accessible entre le processeur et la mémoire, capant ainsi les performances globales du système. En réponse à ces contraintes, et grâce à l'augmentation continue du nombre de transistors pouvant être intégrés sur une seule puce, les concepteurs ont opté pour l'intégration de plusieurs unités de calcul indépendantes au sein d'une même puce. Cela peut se faire soit en intégrant plusieurs unités de traitement indépendantes (ou **cœurs**) dans un seul processeur physique, on parle alors de **processeur multi-cœurs** [36], soit en intégrant plusieurs processeurs physiques dans un ordinateur, auquel cas on parle de **multiprocesseur**. Dans la Figure 2.1, on peut voir un exemple de micro-architecture multicœur *Skylake*, développée par *Intel*.

Ces deux types d'approche présentent l'avantage de ne pas nécessiter une architecture plus complexe pour chaque cœur, tout en offrant une gestion plus fine de la consommation d'énergie. En effet, lorsqu'un cœur n'est pas sollicité pour effectuer des calculs, il peut être désactivé afin d'économiser de l'énergie.

Certains multiprocesseurs possèdent une partie de leurs puces qui est répliquée, en particulier celle responsable de conserver l'état d'un processeur. Ainsi, chaque processeur apparaît comme étant un ensemble de **processeur logique** pouvant effectuer des calculs simultanément. On parle alors de **multithreading simultané**, ou, pour certains processeurs *Intel*, d'**hyperthreading**. Dans ce cas, au cours d'un même cycle, les différents processeurs logiques peuvent accéder à des ressources physiques distinctes. Généralement, il n'y a que deux processeurs logiques par processeur.

Avec ces nouvelles puces, les différents cœurs doivent communiquer pour résoudre une tâche. Le modèle de calcul principalement utilisé aujourd'hui, classifié par Flynn [18], est le modèle *Multiple-Instruction, Multiple-Data* (*MIMD*). Ce modèle comprend plusieurs unités de calcul, chacune ayant ses propres instructions et appliquant ces instructions à des données différentes.

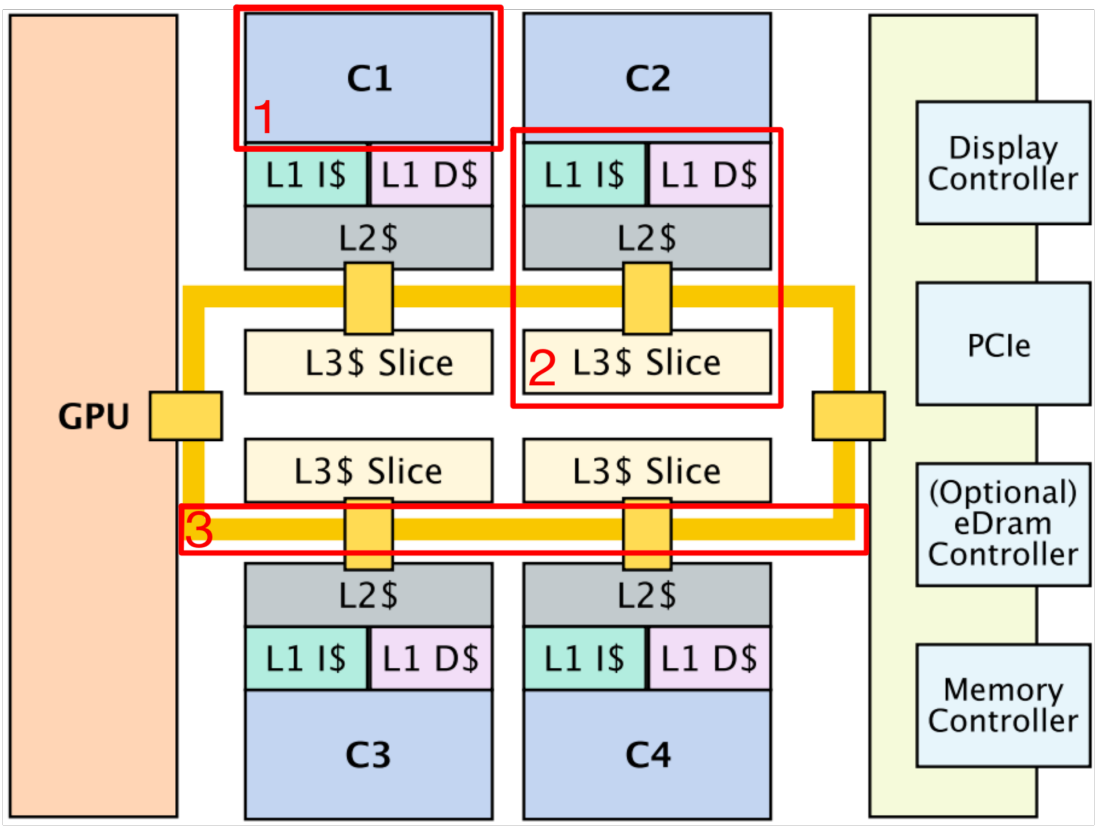

FIGURE 2.1 – Diagramme d'un processeur *Intel* 4 cœurs avec une micro-architecture Skylake (1. cœur, 2. niveaux de cache, 3. bus). À partir de Forsell et al. [19].

Un autre modèle également utilisé est le modèle *Single-Instruction, Multiple-Data* (*SIMD*), où une unité de calcul spéciale se charge de récupérer les instructions et de les dispatcher ensuite aux autres unités de calcul. Chaque unité de calcul reçoit les mêmes instructions et charge les données dans des emplacements différents de la mémoire. Ce modèle est utilisé, par exemple, dans certaines unités de traitement graphique (GPU).

Parallélisme logiciel

L'architecture matérielle multi-cœur et les techniques de pipelining ont permis de maximiser le traitement parallèle au niveau matériel. Cependant, pour exploiter pleinement ces capacités matérielles, il est essentiel que le logiciel, et en particulier

le **système d'exploitation**, coordonne et gère efficacement les ressources pour optimiser les performances globales du système.

Un système d'exploitation est un logiciel essentiel qui agit comme une interface entre le matériel informatique et les programmes utilisateurs. Il gère les ressources matérielles de l'ordinateur, telles que le processeur, la mémoire, les périphériques, et permet l'exécution des programmes de manière coordonnée et efficace.

Lorsqu'un programme est lancé, le système d'exploitation crée un **processus**. Ce processus est une instance du programme en cours d'exécution qui gère plusieurs éléments comme la pile d'exécution, la mémoire utilisée ou encore les descripteurs de fichiers. Il existe deux types de processus : les **processus lourds** et les **processus légers**. Les processus lourds sont généralement simplement appelés processus. Chaque processus lourd est créé avec son propre espace mémoire, ce qui les rend isolés les uns des autres. La création d'un processus lourd peut s'avérer coûteuse en termes de ressources. Les processus légers, souvent appelés ***threads***, coexistent au sein d'un processus lourd et peuvent donc facilement communiquer entre eux puisqu'ils partagent le même espace mémoire.

L'accès à la mémoire étant une opération coûteuse en termes de temps, certains threads peuvent se retrouver bloqués pendant la récupération des données nécessaires. Pour atténuer ce décalage, plusieurs threads peuvent s'exécuter simultanément, une technique connue sous le nom de **multithreading**. Ainsi, tandis qu'un thread attend l'accès à la mémoire, un autre peut initier ou poursuivre une opération, optimisant ainsi l'utilisation des ressources processeur.

Il existe plusieurs approches pour passer d'un thread à un autre, telles que le **multithreading à grain fin** et le **multithreading à grain grossier**. Dans le cas du multithreading à grain fin, on passe d'un thread à un autre après chaque instruction, tandis que dans le multithreading à grain grossier, on change de thread uniquement lors de longues attentes, comme par exemple lors d'une lecture dans la mémoire vive.

Bien que le parallélisme logiciel améliore la gestion des processus et des threads, l'efficacité de ces exécutions parallèles dépend également d'un accès rapide aux données. Une organisation hiérarchique de la mémoire devient alors cruciale pour supporter plusieurs unités de calcul en parallèle, permettant ainsi une coordination efficace entre le processeur et la mémoire à travers des structures de caches et de

registres.

Architecture mémoire

Pour que ces cœurs multiples ou processeurs puissent fonctionner efficacement ensemble, ils doivent avoir accès à des données rapidement accessibles. C'est ici qu'intervient le rôle crucial de la mémoire. Pour effectuer des calculs, les étapes intermédiaires ainsi que les résultats de ces calculs peuvent être stockés dans la mémoire principale. Ce composant sert donc à mémoriser des informations et des données. En principe, la mémoire est composée de plusieurs bits qui, regroupés par 8, forment des octets. La mémoire n'étant pas infinie, le nombre d'octets qui la constitue détermine sa capacité. Historiquement, un ordinateur était composé d'un processeur, chargé d'effectuer les calculs, et d'une mémoire principale, destinée à stocker les informations. L'ordinateur moderne, quant à lui, comporte différents types de mémoire : des mémoires très rapides mais de faible capacité, ainsi que des mémoires de grande capacité mais plus lentes. Pour optimiser l'accès aux données, ces mémoires sont organisées en plusieurs niveaux, formant ce que l'on appelle la **hiérarchie mémoire**. Ces niveaux sont les suivants :

— **Les registres** : C'est la mémoire la plus proche du processeur et celle qui peut être lue le plus rapidement. Le temps d'accès aux registres est d'un cycle d'horloge, donc leur vitesse dépend de celle du processeur. Les registres ont généralement une taille de 32 ou 64 bits. Cependant, leur taille et leur nombre dépendent principalement de l'architecture du processeur. Il existe plusieurs types de registres pour stocker des informations différentes : registres généraux, registres de pointeurs, registres de flottants, registres d'entiers, etc.

— **Le cache** : C'est une mémoire intermédiaire entre le processeur et la mémoire principale (On peut le voir dans le bloc 2 de la Figure 2.1). Les données fréquemment utilisées y sont stockées pour permettre un accès plus rapide que depuis la mémoire principale. Cependant, tout comme les registres, sa taille est également limitée. La taille du cache peut varier de quelques kilobits à plusieurs mégabits, et il est souvent divisé en plusieurs sous-niveaux : L1, L2, L3, et parfois L4. Ces sous-niveaux ont un temps d'accès de plus en plus long, mais une capacité de stockage de plus en plus grande.

Le niveau L1 est le plus proche du processeur et est divisé en deux sous-parties : le $L1_i$, qui stocke les instructions d'un programme en cours d'exécution, et le $L1_d$, qui stocke les données. Sa taille varie de 32 à 64 Kb par cœur. Le niveau L2 est également propre à chaque cœur, mais il peut stocker plusieurs centaines de Kb par cœur. Le niveau L3, quant à lui, est partagé entre plusieurs cœurs et peut atteindre plusieurs dizaines de Mb selon les gammes de processeurs. Certains processeurs disposent d'un niveau L4, mais cela reste moins courant. Le temps d'accès au cache peut varier de 1 à 20 cycles pour un processeur cadencé à 3 GHz, en fonction du niveau de cache accédé.

— **La mémoire vive** : On l'appelle aussi *RAM* pour *Random Access Memory*, et elle peut être de type dynamique (*DRAM*) ou statique (*SRAM*) [62]. La principale différence entre ces deux types de mémoire réside dans la structure des cellules de mémoire. La DRAM est plus lente mais permet de stocker une plus grande quantité de données, tandis que la SRAM est plus rapide mais avec une densité de stockage plus faible.

La RAM peut être *volatile* ou *non-volatile*. Dans le premier cas, elle perd son contenu lorsqu'elle n'est plus alimentée, alors que dans le second cas, les données sont conservées même en l'absence d'alimentation. Le temps d'accès à la RAM peut varier de 60 à 180 cycles pour un processeur cadencé à 3 GHz, en fonction du type de mémoire, de l'architecture de l'ordinateur, de la charge du système, etc.

— **La mémoire de masse** : Ce type de mémoire est appelé disque dur. Il est le plus lent mais permet de stocker le plus grand volume de données. Les disques durs peuvent offrir une capacité de stockage allant jusqu'à plusieurs téraoctets. Le temps d'accès au disque dur peut varier de 300 à 60 000 cycles pour un processeur cadencé à 3 GHz, en fonction du type de disque dur et de ses caractéristiques spécifiques.

L'organisation mémoire d'un ordinateur parallèle se divise en deux catégories : soit les unités de calcul partagent une mémoire physique, soit la mémoire est physiquement distribuée entre les unités de calcul. Dans le premier cas, on parle de *Shared Memory Machine* ou **SMM**, tandis que dans le second cas, on parle d'un

modèle à **mémoire distribuée** également connu sous le nom de *Distributed Memory Machine* ou **DMM**. Le temps d'accès à la mémoire peut-être uniforme (architecture *SMP*) ou uniforme (architecture *NUMA* [8]). La communication entre les différents processeurs peut s'effectuer de deux manières principales : en écrivant sur des sections communes de la mémoire, ce qui correspond à un système à **mémoire partagée**, ou en envoyant directement les données d'un processeur à un autre, ce qui correspond à un système à **passage de messages**. Plus de détails seront donnés en Section 2.2.

Les différents processeurs partagent donc une même mémoire. Cette mémoire est composée d'un ensemble de variables partagées, qui peuvent être lues et modifiées par tous les processeurs, en lecture comme en écriture. Comme mentionné précédemment, cette mémoire est divisée en plusieurs sous-parties afin d'optimiser la vitesse d'accès, tant en lecture qu'en écriture.

Chaque cœur de processeur disposant de sa propre mémoire cache, qui stocke une copie locale de la mémoire globale et qui n'est pas partagée, il est crucial que les différents cœurs se synchronisent pour maintenir une vue cohérente de la mémoire globale. En effet, si deux processeurs, p et q, utilisent une même variable partagée et que p modifie cette variable dans son cache sans que q ne mette à jour le sien, q ne verra pas la modification effectuée par p. Ce phénomène est souvent désigné sous le nom de **problème de cohérence de cache**.

Pour résoudre ce problème, un **protocole de cohérence de cache** est mis en place, comme détaillé en Section 2.2.2 .

Virtualisation

Pour exécuter plusieurs applications en parallèle de manière efficace, il est souhaitable qu'elles soient stockées dans un niveau de mémoire offrant des accès rapides pour tous les processus. Cependant, plus la mémoire est rapide, plus elle est limitée en espace. Il est donc impossible de charger entièrement plusieurs applications en parallèle dans la *RAM*.

Pour surmonter cette limite, le système d'exploitation fournit aux applications une vue virtuelle unifiée de la mémoire. Cette "virtualisation de la mémoire" repose sur un mappage entre adresses virtuelles (utilisées par les applications) et

adresses physiques (réelles en mémoire). La *MMU* (*Memory Management Unit*) dans le processeur est responsable de la traduction de ces adresses virtuelles en adresses physiques, permettant ainsi la répartition des applications en fragments qui peuvent être présents ou non en *RAM*, selon leur utilisation.

La mémoire virtuelle peut être organisée en pagination ou en segmentation. La pagination divise la mémoire en blocs de taille fixe, alors que la segmentation la divise en segments de taille variable correspondant aux structures logiques du programme. Il est aussi possible de combiner ces approches : chaque segment peut être constitué de pages, où une adresse virtuelle inclut un numéro de segment et un numéro de page. Bien que la segmentation soit utile pour certaines applications, la pagination reste le mécanisme de partitionnement le plus courant.

Lors de l'exécution, l'application accède à différentes pages de la mémoire virtuelle. Si une page demandée n'est pas en *RAM*, cela génère un *page fault* : le système d'exploitation doit alors récupérer cette page sur le disque pour la charger en RAM. Ce processus est appelé *demand paging*. Dans certaines situations, plusieurs pages sont transférées vers le disque lorsque le processus est inactif, une opération appelée *swapping*.

Un bon algorithme de remplacement des pages est nécessaire pour éviter un phénomène de *thrashing*, où le système passe plus de temps à déplacer des pages qu'à exécuter les instructions, entraînant une dégradation des performances.

Communication matérielle

On peut regrouper les différentes composantes d'une puce de processeur en trois grands groupes : d'abord, les unités d'entrée et de sortie, qui permettent d'interagir avec l'ordinateur ; ensuite, le processeur lui-même, qui interprète le programme fourni en entrée et effectue les calculs nécessaires ; et enfin, la mémoire, qui enregistre les données au fur et à mesure, permettant ainsi au processeur de progresser dans l'exécution du programme.

Ces trois groupes de composants communiquent entre eux via des **bus** [77], des ensembles de fils électriques reliant les différentes parties du système (Ils sont représentés dans le bloc 3 de la Figure 2.1). Ces bus se divisent en trois sous-catégories : *(i)* le **bus de données**, qui, comme son nom l'indique, permet l'échange de don-

nées entre les sous-unités ; *(ii)* le **bus de commande**, qui permet au processeur de configurer la mémoire, de gérer les sorties, et de recevoir les instructions en entrée ; et enfin, *(iii)* le **bus d'adresse**, qui permet au processeur de sélectionner avec quelle sous-unité il souhaite échanger des données.

Résumé

Dans cette section nous abordons le fonctionnement interne des processeurs en se focalisant sur la représentation des données et la gestion des instructions à travers des cycles d'horloge. Nous explorons l'organisation des processeurs, des cycles d'instructions, ainsi que des techniques d'optimisation comme le *pipelining*. Nous abordons également la hiérarchie mémoire, qui inclut les registres, les caches, la RAM et la mémoire de masse, permettant d'optimiser les accès aux données en fonction de leur proximité et vitesse d'accès. L'évolution des architectures, allant des processeurs monocœurs aux processeurs multicœurs, est discutée, avec un accent sur l'importance du parallélisme pour améliorer les performances.

2.2 Stratégies de communication

Dans la section précédente, nous avons étudié l'architecture des processeurs, les limitations des processeurs monocœurs, ainsi que les différentes stratégies pour concevoir des architectures parallèles. Ces dernières permettent la création de programmes capables de traiter des tâches de complexité variable.

Pour simplifier et unifier la terminologie dans les sections suivantes, nous utiliserons ici le terme *thread* pour désigner une entité indépendante de calcul.

Dans cette section, nous examinons deux stratégies majeures de communication dans les systèmes parallèles. La première, **la mémoire partagée**, repose sur une vision globale de la mémoire accessible par tous les threads. La seconde, **le passage de message**, est caractérisée par l'échange direct de données entre threads via des opérations explicites d'envoi et de réception. Nous discuterons également de la manière dont ces deux approches sont implémentées au niveau logiciel et matériel.

2.2.1 Passage de message

Implémentation physique

Dans un modèle de mémoire distribuée, chaque thread possède sa propre mémoire locale et ne peut accéder directement à celle d'un autre thread. Les échanges de données s'effectuent exclusivement par des opérations explicites d'envoi et de réception de messages. Ces messages transitent souvent via un réseau reliant les différents nœuds (ou machines) entre eux. Ce type de réseau est appelé **réseau d'interconnexion**. Ce dernier relie non seulement les threads, mais également les modules mémoire. Ainsi, un message circulant sur ce réseau peut contenir aussi bien des données que des requêtes mémoire. Le destinataire d'un message peut donc être soit un autre thread, soit un module mémoire.

Dans les modèles à passage de message, les accès mémoire et les transferts de messages constituent une part significative des opérations effectuées. Par conséquent, le choix du réseau d'interconnexion est crucial pour garantir des performances optimales.

Un réseau peut être de deux types : direct ou indirect. Dans le premier cas, les nœuds sont directement reliés entre eux par un lien physique, formant ainsi un réseau point à point. Dans le second cas, les nœuds communiquent par l'intermédiaire d'autres composants, tels que des bus ou des commutateurs (switches), formant un réseau d'interconnexion dynamique.

Les communications point à point sont plus simples à gérer puisque la communication s'effectue directement entre deux nœuds. Le choix de la topologie du réseau est un facteur déterminant pour les communications entre nœuds. La topologie définit la structure géométrique sous-jacente du réseau, c'est-à-dire l'organisation des liens entre les différents threads. Elle peut prendre diverses formes, chacune présentant ses propres avantages et inconvénients en fonction des critères à considérer, tels que le nombre de connexions physiques et la résilience aux pannes (exemples : graphe complet, réseau en anneau, en étoile, hypercube, etc.).

Lorsque les nœuds communiquent de manière indirecte, la gestion du transit des messages joue un rôle central dans l'implémentation du réseau. Le choix du chemin emprunté par un message pour aller d'un nœud A à un nœud B est réalisé par un algorithme de routage. L'objectif de cet algorithme est de minimiser le

"coût" de la communication entre A et B. Ce coût dépend du chemin choisi mais également de la quantité de messages en transit sur ce chemin. Trois éléments influencent la sélection du chemin : la topologie du réseau, la contention (qui survient lorsque plusieurs messages tentent d'emprunter simultanément le même lien) et la congestion (qui se produit lorsque trop de messages sont envoyés à travers une ressource limitée, entraînant des pertes de messages).

Il existe plusieurs types d'algorithmes de routage, qui peuvent être déterministes ou non déterministes. Dans le premier cas, le chemin emprunté par un message dépend uniquement du nœud émetteur et du nœud récepteur. Par exemple, l'algorithme *Deterministic graph contraction* est déterministe, tandis que l'algorithme *Random mate graph contraction* est non déterministe [7].

Le choix de l'algorithme de routage détermine le chemin que prendra un message, mais non la manière dont celui-ci transitera physiquement à travers le réseau. Cette dernière partie est régie par les stratégies de commutation (*switching strategies*). Les switches sont des éléments matériels par lesquels transitent les messages entre les nœuds. Les stratégies de commutation déterminent si, et comment, un message est fragmenté en plus petits messages appelés paquets, comment ces paquets sont transmis, et comment les liens entre les différents switches sont alloués.

Il existe deux principales stratégies de commutation : la commutation de circuits (*circuit switching*) et la commutation de paquets (*packet switching*) [23]. Dans la commutation de circuits, le chemin complet entre le nœud émetteur et le nœud récepteur est d'abord établi, puis réservé pour toute la durée de la transmission. En revanche, dans la commutation de paquets, le message est fragmenté en plusieurs paquets, chacun étant acheminé de manière indépendante de l'émetteur vers le récepteur.

Implémentation logiciel

En général, des bibliothèques de communication sont utilisées pour fournir une variété d'opérations différentes, adaptées aux besoins spécifiques du programme. Parmi ces bibliothèques, certaines reposent sur un concept appelé le **modèle acteurs** [30]. Dans ce modèle, les différentes entités indépendantes qui composent le système, appelées acteurs, interagissent exclusivement par l'échange de messages.

Chaque acteur peut accomplir trois types d'actions fondamentales : créer de nouveaux acteurs, envoyer des messages à d'autres acteurs, ou modifier son propre état interne.

Un exemple notable de l'utilisation du modèle des acteurs est l'Interface de Passage de Messages, ou *Message Passing Interface* [20] (*MPI*).

MPI établit les règles, la syntaxe, et la sémantique que les développeurs doivent suivre lorsqu'ils utilisent des fonctions pour effectuer des communications standardisées entre différents threads dans un système de mémoire distribuée. Deux versions principales de la norme *MPI* existent : *MPI-1*, qui définit les normes de communication dans un système avec des threads statiques, c'est-à-dire lorsque le nombre de threads reste constant du début à la fin du programme ; et *MPI-2*, une extension de *MPI-1* qui prend en charge les systèmes avec des threads dynamiques, les communications unilatérales, et les entrées/sorties parallèles.

MPI est essentiellement une spécification qui définit la syntaxe et la sémantique des opérations de communication, laissant aux développeurs la tâche de les implémenter. En normalisant la syntaxe des opérations, *MPI* garantit la portabilité des programmes sur différentes architectures matérielles.

Un programme basé sur *MPI-1* consiste en un groupe statique de threads qui s'échangent des messages, avec un seul thread responsable des entrées/sorties. Bien que le nombre de threads utilisés dans le programme reste fixe, le programme doit être conçu de manière à pouvoir fonctionner avec un nombre arbitraire de threads, assurant ainsi sa portabilité. Le nombre exact de threads est spécifié lors du démarrage du programme.

Les opérations d'échange de données entre différents threads peuvent présenter plusieurs caractéristiques distinctes. Certaines opérations peuvent être **bloquantes**, c'est-à-dire que tous les changements d'état induits par l'appel à cette opération sont effectués avant que le thread ayant fait l'appel puisse continuer son exécution. À l'inverse, d'autres opérations peuvent être **non bloquantes**, ce qui signifie que l'appel à cette opération peut se terminer avant que les changements d'état soient effectivement observés. Dans ce cas, le thread continue l'exécution du programme immédiatement après l'appel.

Les opérations de communication peuvent également être **synchrones** ou **asynchrones**. Dans le cas des opérations synchrones, les opérations d'envoi et de ré-

ception de deux threads ne peuvent pas être complétées tant que chacune d'elles n'a pas commencé. Cela signifie que l'opération d'envoi ne se termine pas immédiatement après avoir été déclenchée, mais attend que le thread récepteur soit prêt à recevoir les données. En revanche, dans le cas des opérations asynchrones, le thread qui exécute l'opération d'envoi ne se synchronise pas avec le thread exécutant l'opération de réception, permettant ainsi une exécution plus souple mais potentiellement plus complexe à gérer.

Une propriété importante garantie par tous les programmes *MPI* est l'ordre de réception des messages. En effet, lorsque un thread A envoie plusieurs messages à un autre thread B, ces messages sont reçus par le thread B dans l'ordre exact dans lequel ils ont été envoyés. Cette garantie d'ordre est essentielle pour maintenir la cohérence et la prévisibilité des communications dans les systèmes parallèles. Elle permet aux développeurs de concevoir des programmes qui reposent sur l'ordre des messages, assurant ainsi le bon fonctionnement de leurs applications.

L'utilisation des opérations d'envoi et de réception doit être effectuée avec une grande prudence, car elle peut conduire à des situations de ***deadlock*** où le programme se retrouve bloqué et incapable de progresser. Un *deadlock* peut survenir, par exemple, lorsque deux threads tentent de s'envoyer des messages, mais que le programme est conçu de telle manière que chaque thread attend d'abord la réception du message de l'autre. Dans ce scénario, les deux threads se retrouvent dans une situation de blocage mutuel : chacun attend indéfiniment le message de l'autre, et aucun des threads ne peut continuer son exécution.

Dans un système *MPI*, l'envoi de messages se déroule en trois étapes distinctes. Tout d'abord, les données à envoyer sont copiées depuis le *buffer* d'envoi vers le système de *runtime MPI*. Durant cette étape, les données sont associées à un en-tête qui contient des informations essentielles, telles que l'identifiant du thread expéditeur, l'identifiant du thread destinataire, un *tag* pour distinguer différents types de messages, et le *communicateur* utilisé pour la communication.

Ensuite, le message, incluant les données et l'en-tête, est transmis via le réseau au thread destinataire. Cette étape implique le transfert des données à travers l'infrastructure réseau qui relie les différents threads du système.

Enfin, une fois que le message a atteint le thread destinataire, les données sont copiées depuis le *buffer* du système *MPI* vers le *buffer* de réception du message.

Ce thread assure que les données sont correctement transférées et prêtes à être utilisées par le thread récepteur.

Dans un programme *MPI*, les opérations de communication sont réalisées à travers un **communicateur**. Les communicateurs définissent un groupe de threads capables de communiquer entre eux en échangeant des messages au sein d'un même espace de communication. Le communicateur permet donc d'avoir un cadre qui délimite les interactions entre les threads au sein du système.

Attiya et al. [4] proposent un émulateur pour simuler un système à mémoire partagée au sein d'un système à passage de messages, sous réserve qu'une majorité de threads et de liens ne soit pas défaillant. Grâce à cette approche, il devient possible de résoudre un problème dans un environnement à mémoire partagée et de transposer directement la solution dans un système à passage de messages, simplifiant ainsi la conception de programmes tolérants aux pannes.

2.2.2 Mémoire partagée

Implémentation physique

Dans le modèle à mémoire partagée (ou *SMM*, pour *Shared Memory Machine*), les différentes unités de calcul possèdent une vision globale et cohérente de la mémoire, contrairement au modèle à passage de messages, où les threads doivent échanger des données via un réseau pour obtenir une vue complète de l'état du système. Cet espace mémoire partagé est accessible en lecture et en écriture par l'ensemble des threads du système, offrant ainsi une approche centralisée de la gestion des données.

Lorsqu'un accès uniforme à la mémoire est assuré pour tous les threads, on parle de **multiprocesseur symétrique** (ou *SMP*, pour *Symmetric MultiProcessor*). Ce type d'architecture se retrouve principalement dans les systèmes où la mémoire est physiquement partagée entre les différents threads, qui disposent ainsi d'un espace d'adressage commun. Les *SMP* utilisent souvent un bus central (Section 2.1) pour garantir un temps d'accès égal pour tous les threads. Toutefois, en raison des limitations inhérentes à l'utilisation d'un bus central, le nombre de threads dans une architecture *SMP* est restreint. De ce fait, les systèmes reposant sur un seul *SMP* sont également désignés comme des systèmes à **accès uniforme**

à la mémoire (ou *UMA*, pour *Uniform Memory Access*). Les processeurs de bureau *Intel Core* ou *AMD Ryzen* ont typiquement des architectures SMP car ils ne nécessitent pas souvent une gestion complexe de la mémoire.

À l'opposé, dans les systèmes où la mémoire partagée est distribuée, le temps d'accès à la mémoire varie en fonction de la localisation de la mémoire à laquelle on souhaite accéder.

Il est possible de rencontrer des configurations où, bien que l'architecture soit de type mémoire distribuée, un espace d'adressage commun est partagé entre tous les threads. Ce sont des systèmes avec un **accès non-uniform à la mémoire** (ou ***NUMA*** [8] pour *Non-Uniform Memory Access*). Dans une architecture *NUMA*, les adresses utilisées par les différents threads appartiennent à un espace d'adressage global. Lorsqu'un thread accède à une partie de la mémoire qui est localement adressée, l'accès est direct et rapide. En revanche, si la mémoire souhaitée est localisée sur un autre processeur, une communication sur le réseau est initiée. Les serveurs avec plusieurs processeurs ou *multi-socket* utilisent souvent une architecture NUMA. Ça peut être le cas d'un serveur équipé de deux processeurs *Intel Xeon Platinum* par exemple.

Pour tirer le meilleur parti de cette architecture, il est crucial d'optimiser la répartition des données afin de maximiser les accès locaux, car la communication réseau entraîne des latences significatives et peut devenir un goulot d'étranglement. Optimiser l'affinité entre les threads et la mémoire, en veillant à placer les deux de manière à privilégier des accès mémoire rapides, peut améliorer les performances d'un programme jusqu'à 70% [63, 64, 65]. En effet, un accès mémoire distant peut être $1,5$ à $5\times$ plus lent qu'un accès à la mémoire locale.

La mémoire dans les systèmes à mémoire partagée est organisée en hiérarchie, avec pour objectif d'optimiser l'accès aux données fréquemment utilisées en les rendant accessibles rapidement. Cette optimisation repose notamment sur l'utilisation de caches, qui jouent un rôle clé dans la réduction du temps d'accès moyen aux données (Section 2.1).

Cette amélioration de la performance est généralement garantie grâce à deux propriétés fondamentales : la **localité temporelle** et la **localité spatiale**.

La localité temporelle repose sur l'hypothèse que les futurs accès mémoire sont plus susceptibles de se produire au même emplacement que les accès mémoire

récents. En d'autres termes, si une donnée a été récemment utilisée, il est probable qu'elle soit utilisée à nouveau dans un avenir proche. La localité spatiale suggère que les futurs accès mémoire sont plus susceptibles de se produire à proximité des accès mémoire récents. Cela signifie que si un thread accède à une certaine donnée, il est probable que les données adjacentes soient également nécessaires peu de temps après.

Le cache est décomposé en sous-unités appelées **blocs de cache** ou **lignes de cache**. Ce sont ces blocs qui sont transférés depuis la mémoire principale vers le cache. Chaque ligne de cache est accessible via une **adresse mémoire**, qui joue un rôle crucial dans l'organisation et l'efficacité de l'accès aux données stockées.

L'adresse mémoire est généralement composée de trois parties essentielles : le **tag**, qui permet de vérifier si les données stockées dans le cache correspondent bien à celles demandées par le thread ; l'**index**, qui identifie la ligne de cache spécifique où la donnée recherchée pourrait se trouver ; et l'**offset**, qui localise précisément où, dans la ligne de cache, la donnée est stockée. Lorsque le thread émet une requête mémoire, l'adresse est décomposée pour retrouver l'**entrée de cache** contenant le tag et les données stockées dans la ligne de cache correspondante.

L'organisation des entrées de cache joue un rôle déterminant dans les performances d'un programme. Ces entrées sont regroupées en ensembles, et la manière dont elles sont organisées influence directement l'efficacité de l'accès aux données. On parle alors d'**organisation logique du cache**, et il existe trois principaux types d'organisations : l'**associativité directe**, **complète**, et **par ensemble**.

Dans une organisation à associativité complète, toutes les entrées de cache sont stockées dans un seul ensemble. Lorsqu'une requête mémoire est émise, tous les tags de chaque entrée doivent être vérifiés pour trouver une correspondance. Bien que cela maximise la flexibilité en termes de placement des données dans le cache, l'évaluation de chaque tag est inefficace et coûteuse en termes de temps de calcul. À l'opposé, dans une organisation à associativité directe, chaque ensemble contient une seule entrée de cache. Ainsi, il y a autant d'ensembles que d'entrées de cache. L'index dans l'adresse mémoire permet de localiser directement l'entrée de cache contenant la donnée demandée. Cette organisation a l'avantage d'être extrêmement rapide, car l'accès se fait directement. Cependant, comme il n'y a qu'une entrée par ensemble, cela peut entraîner des conflits lorsque plusieurs données, qui devraient

être stockées dans la même entrée, sont fréquemment modifiées. Ce phénomène, peut être atténué en augmentant la taille du cache [31], mais cela a des limites pratiques. L'organisation à associativité par ensemble est un compromis entre les deux précédentes. Ici, chaque ensemble contient plusieurs entrées de cache. L'index dans l'adresse mémoire identifie d'abord un ensemble, puis les tags des entrées au sein de cet ensemble sont vérifiés pour trouver la correspondance. Une fois que le tag correspondant est trouvé, l'offset est utilisé pour localiser précisément la donnée dans la ligne de cache. Cette organisation permet de réduire les accès concurrents sur une même entrée de cache tout en évitant l'évaluation exhaustive de tous les tags, ce qui en fait un compromis efficace entre rapidité et flexibilité.

Les différentes organisations logiques du cache jouent un rôle fondamental dans la gestion efficace de l'accès aux données. Cependant, dans les systèmes multiprocesseurs, un défi supplémentaire et majeur se présente. Lorsqu'une donnée est répliquée dans les caches de plusieurs threads et que chacun d'eux la modifie indépendamment, il peut en résulter des incohérences : chaque thread pourrait alors travailler avec une version différente de la donnée. Sans une synchronisation adéquate avec la mémoire principale, cela pourrait conduire à des résultats incorrects, où l'exécution parallèle du programme ne correspondrait à aucune séquence logique d'exécution séquentielle.

La stratégie d'écriture dans la mémoire principale joue également un rôle crucial dans la cohérence des données au sein du système. Deux stratégies principales sont couramment utilisées. Dans la première, le thread écrit dans la mémoire principale en même temps qu'il modifie sa ligne de cache, une approche connue sous le nom d'**écriture immédiate** (ou *write-through*). Dans la seconde, l'écriture dans la mémoire principale est différée jusqu'au moment où la ligne de cache est effacée du cache, une stratégie appelée **écriture différée** (ou *write-back*).

L'écriture immédiate a l'avantage de simplifier le maintien d'une vue cohérente des données. Cependant, cette méthode est moins efficace, car elle implique une grande quantité d'écritures, même lorsque ces écritures ne sont pas strictement nécessaires. À l'inverse, l'écriture différée permet d'éviter des écritures inutiles dans la mémoire principale. Par exemple, si un thread modifie plusieurs fois une donnée sans qu'elle soit utilisée par un autre thread, seules les modifications finales doivent être répercutées dans la mémoire principale, ce qui réduit la charge d'écriture.

Toutefois, cette stratégie exige une communication accrue entre les threads, et est donc plus complexe à implémenter.

L'organisation des données dans le cache est ainsi gérée par des heuristiques définies par le **contrôleur de cache**, une interface implantée entre le processeur et le cache. Ce contrôleur est chargé de gérer la communication et de prendre des décisions sur la manière dont les données doivent être manipulées.

Les heuristiques de **gestion du contenu** déterminent quels éléments doivent être présents dans le cache pour anticiper et répondre efficacement aux besoins du processeur. Par exemple, la **prélecture** (ou *prefetching*) permet de charger une donnée dans le cache avant même que la requête la concernant ne soit émise, optimisant ainsi la performance du système. De plus, les politiques de remplacement décident quelles données doivent être évacuées du cache lorsque celui-ci atteint sa capacité limite, et ce pour garantir que les données les plus pertinentes restent accessibles rapidement.

Parallèlement, les heuristiques de **gestion de la cohérence** sont conçues pour garantir que l'image des données demeure cohérente à travers la mémoire partagée. Pour ce faire, le contrôleur de cache applique un **protocole de cohérence de cache** [39, 58].

Dans les processeurs modernes, le protocole de cohérence le plus couramment utilisé est le protocole ***MESI***. Ce protocole attribue à chaque ligne de cache un état spécifique basé sur les opérations de lecture et d'écriture récentes. Ces états, au nombre de quatre, forment l'acronyme "*MESI*". Une ligne de cache est en état ***M****odified* lorsqu'elle n'est présente que dans le cache d'un seul thread et que sa valeur diffère de celle stockée dans la mémoire principale. Si la ligne de cache correspond à la valeur dans la mémoire principale et n'est présente que dans un seul cache, elle est en état ***E****xclusive*. Lorsque plusieurs threads partagent une ligne de cache et que sa valeur est identique à celle de la mémoire principale, elle est en état ***S****hared*. Enfin, l'état ***I****nvalid* indique qu'une copie de la ligne de cache a été modifiée dans le cache d'un autre thread, rendant ainsi la version actuelle obsolète.

L'état d'une ligne de cache est représenté par un ou deux bits, qui sont lus lorsqu'un thread tente d'accéder à une donnée présente dans cette ligne. Ainsi, lorsque le thread essaie de lire un registre situé dans une ligne de cache invalide, on parle de **cache *miss***. En revanche, si le registre se trouve dans une ligne de

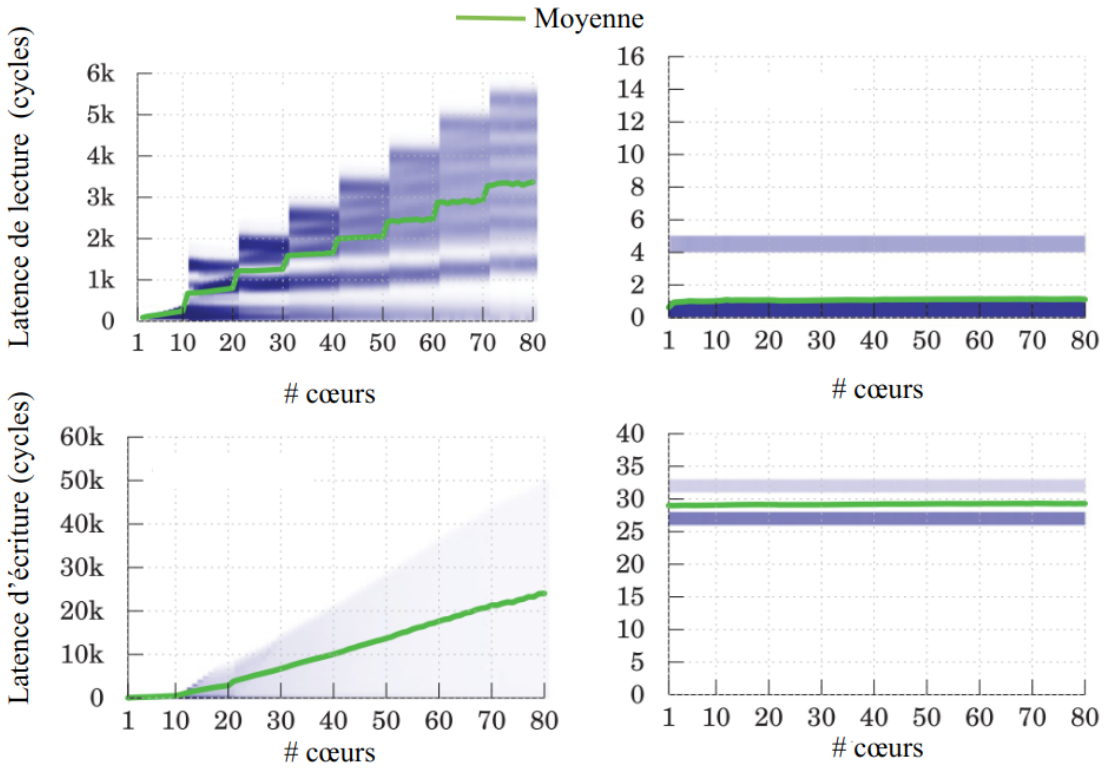

FIGURE 2.2 – Chaque graphique montre la distribution du nombre de cycle nécessaire pour effectuer une lecture ou une écriture pour N cœur(s). Plus la ligne est sombre, plus le nombre de cycle est fréquent pour N cœur(s). À partir de Clements et al. [11]

cache valide, on parle alors de **cache *hit***. Plus un programme génère de cache hits, meilleures sont ses performances.

Ainsi, lorsque les différents threads accèdent à des lignes de cache distinctes, les performances du programme augmentent de manière linéaire en fonction du nombre de cœurs, un phénomène désigné par le terme de **scalabilité**.

La Figure 2.2 illustre le nombre de cycles nécessaires pour effectuer une lecture ou une écriture en fonction du nombre de cœurs, que ce soit sur une même ligne de cache pour les lectures ou sur une ligne de cache dédiée à chaque thread pour les écritures. On constate que, lorsque la ligne de cache n'est jamais mise à jour (en haut à droite de la figure), les performances s'améliorent lors des accès exclusivement en lecture. En effet, dans ce cas, la ligne de cache demeure dans un état *Shared*, ce qui permet aux threads d'éviter de mettre à jour la ligne de cache en

allant lire dans la mémoire principale.

En revanche, si la ligne de cache est mise à jour entre les lectures (en haut à gauche de la figure), le temps requis pour lire les données augmente avec le nombre de cœurs, car l'état de la ligne passe régulièrement de *Shared* à *Modified*/*Exclusive*, obligeant ainsi les threads à mettre régulièrement à jour la ligne de cache en lisant dans la mémoire principale.

Lorsque les threads effectuent des mises à jour sur des lignes de cache distinctes (en bas à droite de la figure), le temps nécessaire pour écrire dans le cache reste constant, car la ligne de cache demeure dans l'état *Modified*. Cela s'explique par le fait qu'il n'est pas nécessaire de vider la ligne de cache dans la mémoire principale, ce qui permet une scalabilité des performances.

En revanche, lorsque les mises à jour se font sur une même ligne de cache (en bas à gauche de la figure), le temps requis pour écrire dans le cache augmente proportionnellement au nombre de cœurs. Cela est dû au fait que la ligne de cache passe par l'état *Invalid*, ce qui a pour conséquence qu'elle doit être vidée dans la mémoire principale lorsque un thread écrase les données d'un autre thread, entraînant ainsi une augmentation du nombre de cycles nécessaires pour effectuer une écriture.

Il est intéressant de noter que les performances d'un programme augmentent lorsque les accès mémoire sont disjoint (ou commutatifs). En effet, lorsque le programme n'effectue pas d'opérations de mise à jour, les lectures peuvent être réordonnées sans affecter le résultat final. De même, lorsque les processeurs effectuent des écritures dans des lignes de cache distinctes, l'ordre d'exécution de ces mises à jour n'a aucun impact sur les effets de bord de ces écritures. On peut donc conclure que, si les accès mémoire d'un programme sont commutatifs, le nombre d'accès peut évoluer proportionnellement avec le nombre de cœurs effectuant des opérations.

Cela met en évidence l'importance des opérations commutatives dans la scalabilité des systèmes parallèles.

Si la gestion des registres et des accès à la mémoire via des opérations de lecture et d'écriture est essentielle pour optimiser les performances d'un programme, il est important de se rappeler que les programmes modernes ne se limitent pas à ces opérations de bas niveau. Ils manipulent des objets plus complexes, construits à

partir de ces registres, qui nécessitent des opérations bien plus sophistiquées que de simples lectures et écritures.

Implémentation logiciel

Pour développer des programmes parallèles sur une architecture à mémoire partagée, on utilise des langages de programmation capables de créer des threads exécutés de manière concurrente. Ces langages adhèrent à des normes spécifiques pour en faciliter leur compréhension et leurs utilisation.

Les normes POSIX sont des standards visant à assurer une compatibilité et une interopérabilité entre différents systèmes d'exploitation, en standardisant plusieurs interfaces, telles que les interfaces de programmation d'applications (API) au niveau système et utilisateur, ainsi que les commandes de shell. Ces normes permettent de garantir qu'un programme écrit pour un système d'exploitation respectant POSIX fonctionne sur un autre système compatible avec peu ou pas de modifications.

Grâce à ces normes, la bibliothèque POSIX C, qui spécifie les fonctionnalités de la bibliothèque standard C pour les systèmes POSIX, a introduit la possibilité de créer des pthread (POSIX threads), facilitant ainsi le calcul parallèle. Les différents pthreads d'un même processus partagent un espace d'adressage mémoire commun. Cependant, chaque thread dispose d'une pile d'exécution (ou runtime stack) distincte, qui permet de suivre les appels de fonctions de manière individuelle. Ainsi, des variables globales, accessibles à tous les pthreads, coexistent avec des variables locales, qui ne sont visibles qu'au sein du thread concerné.

Pour utiliser les types de données, interfaces, et macros associées aux pthreads, il est nécessaire d'inclure l'en-tête `<pthread.h>` au début du programme. Cela permet ensuite d'utiliser les types de données tels que `pthread_<objet>_t`, où `<objet>` représente un type spécifique, comme un mutex (le concept est détaillé en Section 2.3.4). L'implémentation interne des types de données pthread est opaque pour le développeur.

Les pthreads créés par le développeur sont mappés aux threads du système d'exploitation via l'ordonnanceur de la bibliothèque pthread, puis exécutés par l'ordonnanceur du système d'exploitation. En conséquence, le développeur ne peut

pas directement choisir le mappage des pthreads sur les processeurs pour optimiser les performances. Cependant, il est intéressant de noter l'existence d'extensions permettant d'attacher un pthread à un processeur spécifique, par exemple via la bibliothèque *sched*.

Lors de l'exécution d'un programme parallèle en C, un premier thread, appelé *main*, est généré et exécute la fonction principale `main()` du programme. Ce thread peut ensuite créer d'autres threads à l'aide de la fonction `pthread_create()`. Lors de la création d'un thread, plusieurs arguments peuvent être spécifiés, tels qu'un pointeur vers un objet de type `pthread_t`, qui sert également d'identifiant de thread (ou TID). Ce TID peut être utilisé ultérieurement par d'autres pthreads pour identifier le thread créé. Un thread peut également récupérer son propre TID à l'aide de la fonction `pthread_self()` et comparer des TIDs avec la fonction `pthread_equal()`. Étant donné que l'implémentation des pthreads est opaque, il est important d'utiliser `pthread_equal()` pour effectuer de telles comparaisons.

Le nombre maximal de pthreads que l'on peut créer dépend généralement de l'architecture du processeur. Un pthread se termine lorsque la fonction associée à sa création se termine, ou lorsqu'il appelle lui-même la fonction `pthread_exit()`. À la fin d'un thread, sa pile d'exécution et sa mémoire locale sont libérées, ce qui explique qu'il est déconseillé d'utiliser une variable locale comme valeur de retour d'un pthread. Un pthread peut attendre la fin d'un autre pthread en appelant la fonction bloquante `pthread_join`. Si un thread se termine sans qu'un autre appelle `pthread_join`, sa mémoire locale n'est pas libérée, permettant ainsi à d'autres threads d'y accéder ultérieurement. Cependant, dans des programmes de grande envergure, cela peut poser problème, car la mémoire disponible est limitée. Il est donc possible de libérer immédiatement la mémoire d'un thread à l'aide de la fonction `pthread_detach()`, qui permet de récupérer les ressources dès que le thread a terminé son exécution.

Il existe divers mécanismes permettant d'assurer la synchronisation entre les threads et de prévenir les problèmes liés à la concurrence, tels que les *race conditions* ou les *deadlocks*. Dans la section suivante, nous explorerons plusieurs de ces techniques spécifiques au langage de programmation JAVA.

Résumé

Dans cette section, nous avons examiné les deux principales stratégies de communication utilisées dans les systèmes parallèles : la **mémoire partagée** et le **passage de messages**. Nous avons souligné que la mémoire partagée permet à plusieurs threads d'accéder à un espace mémoire global et cohérent tandis que le modèle à passage de messages consiste en un échange explicite de données par des opérations d'envoi et de réception.

Nous avons noté que l'implémentation du passage de messages repose fréquemment sur un **réseau d'interconnexion** et des **algorithmes de routage**, visant à optimiser la communication entre les threads puis nous avons introduit **MPI**, une bibliothèque de communication qui repose sur le **modèle acteur**.

Nous expliquons comment la mémoire partagée repose fréquemment sur des mécanismes de **cache** pour accélérer l'accès aux données, en adoptant des stratégies telles que le ***write-back*** ou le protocole **MESI** afin d'assurer la cohérence des caches.

Enfin, nous présentons la bibliothèque **POSIX C** ainsi que différentes fonctions, qui permettent la création et la gestion de **pthreads** disposant à la fois d'une mémoire locale propre à chaque thread et d'une mémoire partagée entre tous les threads d'un même processeur.

2.3 Parallélisme dans Java

Dans la section précédente, nous avons étudié les différents moyens utilisés pour assurer la communication entre les threads dans un système comportant plusieurs unités de calcul.

Dans cette section, nous commencerons par rappeler deux paradigmes de programmation essentiels, qui offrent des cadres conceptuels pour structurer et développer des programmes de manière cohérente : la **programmation procédurale** [34] et la **programmation orientée objet** [41].

Nous proposerons ensuite une brève introduction au Runtime Java.

Enfin, nous explorerons en détail les diverses méthodes permettant d'intro-

duire du parallélisme dans les programmes Java, tout en analysant de manière approfondie les mécanismes de synchronisation nécessaires à leur gestion efficace.

2.3.1 Rappel sur les paradigmes de programmation

Un paradigme de programmation constitue une abstraction de haut niveau qui aide à organiser et à comprendre la manière dont un programme fonctionne. Les paradigmes de programmation se classent généralement en deux grandes catégories : la **programmation impérative** et la **programmation déclarative**. La programmation impérative se caractérise par une série d'instructions successives modifiant l'état du système. À l'inverse, la programmation déclarative privilégie la spécification des opérations réalisables ainsi que les conditions sous lesquelles elles peuvent être effectuées, sans détailler les étapes intermédiaires.

Bien que la programmation déclarative puisse offrir une meilleure lisibilité et une plus grande simplicité, elle ne permet pas un contrôle précis de l'exécution du programme. Cela peut entraîner des performances sous-optimales, notamment dans des contextes où une gestion minutieuse des ressources et des performances est essentielle.

C'est pourquoi, dans le cadre de cette thèse, nous nous concentrerons sur la programmation impérative.

La programmation impérative, contrairement à la programmation déclarative, permet de contrôler l'ordre d'exécution des instructions, offrant ainsi la possibilité d'optimiser les performances et de mieux gérer les ressources. Ce paradigme est principalement divisé en deux catégories populaires : la **programmation procédurale** [34] et la **programmation orientée objet** [41].

Programmation procédurale

La programmation procédurale se concentre sur l'utilisation de **fonctions**, qui sont des blocs d'instructions réutilisables, conçus pour accomplir des tâches spécifiques. L'un des principaux avantages de cette approche est qu'elle favorise une structuration claire du code, offrant ainsi plusieurs bénéfices. Grâce aux fonctions, le code devient plus compréhensible et plus facile à déboguer.

En décomposant un problème complexe en plusieurs sous-problèmes plus simples, les fonctions permettent une résolution plus efficace. Au sein de ces fonctions, il est possible d'utiliser des variables locales, visibles uniquement pendant l'exécution de la fonction, ce qui facilite le contrôle sur la modification des variables dans le programme.

Ces fonctions peuvent accepter des données en entrée, effectuer des calculs, puis renvoyer un résultat. Cela permet de réutiliser le même ensemble d'instructions pour traiter des valeurs différentes sans avoir à réécrire le code. Cette caractéristique permet également l'utilisation de techniques telles que les appels **récursifs**, où une fonction s'appelle elle-même avec des paramètres modifiés. La récursivité est souvent utilisée dans les programmes qui nécessitent des opérations répétitives, comme la recherche d'une valeur dans une structure de données telle qu'un tableau ou un arbre.

Programmation orientée objet

La **programmation orientée objet** [41], quant à elle, structure le programme autour de structures de données appelées objets. Ces objets sont constitués de champs, appelés **attributs**, qui stockent l'état de l'objet, ainsi que de **méthodes**, qui définissent les comportements ou les opérations que l'objet peut effectuer.

En général, la définition des attributs et des méthodes propres à un objet s'effectue au sein d'une **classe**. Une classe sert ainsi de modèle à partir duquel les objets individuels, appelés instances de la classe, sont créés. Chaque instance de la classe hérite des attributs et des méthodes définis dans la classe, mais peut également posséder des valeurs d'attributs spécifiques qui la différencient des autres instances. Ainsi, il est possible d'avoir plusieurs instances d'une même classe lorsque différents objets partagent un comportement commun tout en ayant des états distincts.

Les attributs, qui sont des variables contenant des données, caractérisent chaque instance. Ils peuvent être spécifiques à chaque instance, on parle alors d'**attributs d'instance**, ou bien être partagés par toutes les instances d'une même classe, auquel cas il s'agit d'**attributs de classe**. Cette distinction permet de gérer à la fois les propriétés individuelles et les informations communes à tous les objets d'une classe.

Les méthodes, quant à elles, sont des ensembles d'instructions conçus pour manipuler les attributs d'instance ou pour effectuer des opérations utilisant ces mêmes attributs. Comme les fonctions en programmation procédurale, les méthodes permettent de structurer le code de manière modulaire, facilitant ainsi sa réutilisation et sa maintenance.

Au cours d'un programme, on appel **objet** une instance de classe. Il est caractérisé par son **type** — déterminé par la classe à partir de laquelle il a été instancié — et encapsule une donnée ou un ensemble de données, accessibles via un ensemble de méthodes définies par le type de l'objet. Ces méthodes permettent d'interagir avec l'objet, en modifiant son état ou en accédant aux données qu'il contient. Un objet peut se trouver dans un ensemble d'**états** bien définis, correspondant aux valeurs actuelles des données qu'il encapsule. Les méthodes associées à l'objet sont conçues pour transformer l'état de l'objet ou pour fournir un accès aux données qu'il contient. Lorsqu'une méthode est invoquée sur un objet, l'état de l'objet juste avant l'invocation de la méthode est appelé **précondition**. La précondition définit les exigences initiales que l'état de l'objet doit satisfaire pour que la méthode soit exécutée correctement. Une fois la méthode terminée, l'état de l'objet et la valeur de retour résultant de la méthode sont désignés sous le terme de **postcondition**. La postcondition décrit l'état final de l'objet après l'exécution de la méthode, ainsi que les données éventuellement retournées. Le processus par lequel l'état de l'objet est modifié au cours de la méthode est appelé un **effet de bord**.

L'un des concepts les plus puissants de la programmation orientée objet est l'**héritage**, qui permet de réutiliser des classes existantes pour en créer de nouvelles. Une classe fille peut hériter d'une classe mère, ce qui signifie qu'elle acquiert passivement tous les attributs et méthodes définis par la classe mère. Ce mécanisme favorise la réutilisation du code et facilite l'extension des fonctionnalités existantes. Par ailleurs, il est possible d'ajouter de nouveaux attributs ou de redéfinir des méthodes dans la classe fille, permettant ainsi de spécialiser ou d'enrichir le comportement hérité de la classe mère. Un autre principe fondamental de l'héritage en programmation orientée objet est le **principe de substitution de Liskov** [54]. Ce principe stipule qu'une instance d'une classe fille doit pouvoir remplacer une instance de la classe de mère sans altérer la validité du programme. Cette notion est définit plus formellement par la suite.

Bien que les paradigmes de programmation impérative et de programmation orientée objet offrent des approches puissantes pour structurer et développer des programmes, ils n'abordent pas directement la question de la communication entre différents threads au sein d'un programme. Cette communication est cruciale dans les environnements multithreads et multiprocesseurs, où différents threads doivent coordonner leurs actions et partager des données efficacement.

Dans la suite de cette section, nous utiliserons l'exemple de Java pour illustrer diverses méthodes de création de parallélisme et aborder la gestion de la synchronisation qui en découle.

2.3.2 Présentation du Runtime Java

Java est un langage de programmation orienté objet largement répandu, offrant aux développeurs un éventail d'outils et de techniques pour simplifier la gestion du parallélisme. Lorsqu'un programme Java est lancé, le système d'exploitation crée un processus pour la ***Java Virtual Machine*** (***JVM***), responsable de l'exécution du bytecode généré par la compilation du code source Java.

Après avoir chargé les bibliothèques nécessaires et réservé l'espace mémoire adéquat pour la JVM, celle-ci démarre un thread Java correspondant à l'exécution du programme. Durant cette exécution, la JVM peut soit interpréter le bytecode directement, soit le compiler dynamiquement en code machine grâce au compilateur ***Just-In-Time*** (***JIT***), ce qui améliore les performances.

La mémoire allouée au programme, connue sous le nom de ***heap*** (ou tas), permet de gérer les allocations dynamiques de manière globale. À l'inverse, la ***stack*** (pile) est utilisée pour les allocations locales à courte durée, comme celles de variables internes à une méthode. Tandis que les données de la pile sont automatiquement libérées en fonction du contexte d'exécution comme lors de la sortie d'une méthode, la gestion du tas est assurée par le ***garbage collector***, qui libère l'espace des objets inutilisés. Ce mécanisme a pour rôle d'identifier les objets devenus inutiles pour les libérer et optimiser l'utilisation de la mémoire. Il existe plusieurs types d'algorithmes de garbage collection. Certains, comme ceux dits "*stop-the-world*", suspendent l'exécution du programme pour effectuer la collecte des objets inutilisés. D'autres, comme les garbage collectors parallèles, n'interrompent les

opérations du programme que pour marquer les objets à supprimer, laissant le processus de suppression se faire de manière concurrentielle. Une autre catégorie d'algorithmes, appelés "*copying collectors*", fonctionne en copiant les objets encore actifs dans une nouvelle région mémoire, libérant ainsi l'espace occupé par les objets inutilisés. Les algorithmes de garbage collector peuvent cumuler plusieurs de ces caractéristiques. Par exemple, l'algorithme *Garbage-First* (G1) GC est un copying collector qui suit une approche "stop-the-world", tandis que le *Z Garbage Collector* (ZGC) minimise ces interruptions pour améliorer les performances globales du programme [14, 79].

2.3.3 Création du parallélisme dans Java

En Java, lorsque plusieurs threads accèdent et modifient une donnée partagée, l'ordre dans lequel ces modifications deviennent visibles aux autres threads n'est pas garanti par défaut. Cela peut entraîner des comportements imprévisibles si les opérations de synchronisation ne sont pas correctement implémentées.

Cette absence de garantie s'explique par les optimisations effectuées à plusieurs niveaux, effectuées notamment par le processeur et le contrôleur de cache. Parmi ces optimisations, on trouve le **réordonnancement des instructions**, où le processeur peut exécuter des instructions dans un ordre différent de celui spécifié dans le code. Ces réarrangements sont réalisés pour améliorer l'efficacité et la performance, en particulier dans les programmes séquentiels, en maximisant l'utilisation des pipelines d'exécution et en réduisant les latences liées à l'accès mémoire.

Cependant, dans un contexte de programmation parallèle, ces optimisations deviennent problématiques. Par exemple, un thread peut observer une mise à jour d'une variable alors que d'autres modifications plus anciennes, pourtant effectuées dans le même bloc d'instructions, ne sont pas encore visibles. Cette situation est connue sous le nom de **réordonnancement mémoire**. Le développeur n'a généralement pas de contrôle direct sur ces optimisations, et les effets peuvent varier en fonction de l'architecture matérielle et des décisions de l'ordonnanceur.

Pour garantir la cohérence des données partagées entre plusieurs threads, Java fournit un ensemble de mécanismes de synchronisation qui permettent de contrôler l'ordre d'exécution des opérations et d'éviter les problèmes de concurrence. Mais

avant d'aborder ces mécanismes de synchronisation, il est utile de comprendre comment Java permet de créer et gérer des threads, qui sont les unités de base du parallélisme.

L'héritage de la classe `Thread` constitue une méthode simple et directe pour créer un nouveau thread. Dans cette approche, le code à exécuter en parallèle est défini dans la méthode `run()` de la classe qui étend `Thread`. Pour démarrer l'exécution du code parallèle, il suffit d'appeler la méthode `start()` sur l'objet instanciant cette classe. À ce moment-là, la *JVM* sollicite le système d'exploitation pour créer un thread natif, mappé au thread Java pour exécuter ses instructions. Ce thread natif correspond généralement aux threads POSIX mentionnés précédemment, qui sont gérés directement par le système d'exploitation.

Cette méthode de création du parallélisme impose une contrainte majeure : en Java, une classe ne peut hériter que d'une seule classe parente. Cette limitation restreint les possibilités d'héritage multiple et peut complexifier la conception des classes. Pour contourner ce problème, la composition, qui consiste à inclure un objet `Thread` dans la classe, offre une solution plus flexible, même si elle peut rendre le code légèrement plus verbeux.

Une alternative plus élégante consiste à recourir aux interfaces, qui permettent de combiner plusieurs comportements tout en contournant les restrictions de l'héritage unique. Java propose deux **interfaces** principales pour définir du code parallèle : `Runnable` et `Callable`. L'interface *Runnable* est utilisée pour exécuter des tâches qui ne nécessitent pas de valeur de retour, comme des tâches qui se limitent à changer l'état du système. En revanche, lorsque l'on souhaite obtenir une valeur de retour, par exemple lors de calculs asynchrones, l'interface `Callable` est préférable. En utilisant ces interfaces, une classe peut à la fois hériter d'une autre classe et exécuter du code en parallèle, sans les limitations de l'héritage unique. Lors de la création d'une instance d'un objet implémentant l'interface `Callable`, il est possible de l'associer à un objet de type `Future` ou `promesse`. Ce dernier agit comme un conteneur pour la valeur de retour qui sera calculée de manière asynchrone, en parallèle avec le reste du programme. Une fois le calcul terminé, la valeur résultante sera accessible à travers cette promesse. Pour récupérer cette valeur, on utilise la méthode `get()`, qui est un appel bloquant. Cette mécanique permet de gérer efficacement les tâches parallèles tout en maintenant la possibilité

de synchroniser leur résultat à un moment précis du déroulement du programme.

Une autre option pour exécuter du code asynchrone tout en récupérant une valeur de retour est d'utiliser la classe `CompletableFuture`. Cette classe permet de composer des tâches asynchrones, où la sortie d'une tâche peut servir d'entrée à une autre tâche, créant ainsi une chaîne de calculs asynchrones. De plus, `CompletableFuture` offre la possibilité de spécifier une valeur par défaut en cas d'erreur ou de suspendre l'exécution du programme jusqu'à la fin des calculs asynchrones, offrant ainsi une grande flexibilité et robustesse dans la gestion des opérations parallèles.

Lorsqu'on souhaite accélérer le traitement d'une grande quantité de données, les **flux parallèles** en Java sont une solution efficace. En transformant un flux séquentiel en flux parallèle via `parallelStream()`, on déclenche un mécanisme de division du travail. Le `Fork/Join Framework` découpe récursivement la tâche en sous-tâches plus petites, jusqu'à ce qu'elles puissent être exécutées rapidement par des threads distincts. Ces threads sont issus d'un pool de threads dont la taille est ajustée automatiquement par la *JVM* en fonction des ressources disponibles. Une fois toutes les sous-tâches terminées, les résultats sont agrégés pour former le résultat final.

Lorsqu'il s'agit d'exécuter plusieurs tâches identiques en parallèle, le code peut rapidement devenir surchargé sans l'utilisation de **pools de threads**. La classe `ExecutorService` permet de définir un pool de threads, un ensemble fixe de threads auquel on peut assigner un ensemble de tâches. Cette approche optimise l'utilisation des ressources, car dès qu'un thread termine une tâche, il peut en récupérer une autre en attente, ce qui améliore l'efficacité et la gestion des ressources.

Pour des raisons de performance, il serait idéal de générer autant de threads qu'il y a de tâches à exécuter, conformément à la **loi de Little** [55], qui stipule que pour maintenir une durée de traitement stable lorsque le nombre de requêtes augmente, le nombre d'unités de calcul gérant ces requêtes doit croître en conséquence. Cependant, le **JDK** (*Java Development Kit*) implémente les threads Java à partir des threads du système d'exploitation, limitant ainsi leur nombre. Pour surmonter cette limitation, Java a introduit en 2021 les **threads virtuels**.

Les threads virtuels ne sont pas limités en nombre car ils ne sont pas mappé sur un threads natif de l'OS. En effet, un thread virtuel peut être exécuté sur différents

threads natifs au cours de son cycle de vie. Ils offrent donc l'illusion d'un nombre élevé de threads en associant plusieurs threads virtuels aux threads physiques du système d'exploitation, qui sont beaucoup moins nombreux. Cette virtualisation permet de créer des milliers de threads légers, ce qui est particulièrement utile dans des applications de type *thread-per-request*, où chaque tâche est courte et peut être exécutée de manière asynchrone. Comme les threads virtuels sont légers, il est déconseillé de les gérer via un pool. L'usage optimal des threads virtuels se trouve dans les programmes qui effectuent de nombreuses petites tâches indépendantes. Dans ces scénarios, chaque nouvelle tâche peut simplement impliquer la création d'un nouveau thread virtuel, sans les contraintes des threads traditionnels. Lorsqu'un thread virtuel est actif, il est attaché à un processeur logique, appelé processeur porteur. Lorsqu'il termine son exécution ou rencontre une opération bloquante, il est détaché du processeur porteur, qui devient alors disponible pour un autre thread virtuel. Ce processus d'attachement et de détachement est géré de manière transparente par la *JVM*, permettant une gestion efficace des threads sans bloquer les threads.

Toutes ces fonctionnalités permettent de créer du code concurrent en Java. Cependant, pour éviter les problèmes de cohérence des données, il est essentiel d'utiliser des outils de synchronisation appropriés pour coordonner les différents threads et assurer l'intégrité des opérations concurrentes.

2.3.4 Mécanismes de synchronisation

Quand plusieurs threads tentent d'accéder simultanément à une même donnée, Java ne peut pas garantir que ces accès se feront de manière indivisible (atomicité) ni dans un ordre précis. Ces comportements imprévisibles sont liés aux optimisations effectuées au niveau du système d'exploitation. Pour éviter les erreurs liées à la concurrence et préserver la cohérence des données, Java propose divers mécanismes de synchronisation [27] que nous détaillons par la suite.

L'un des moyens le plus simple d'introduire de la synchronie est d'utiliser le mot-clé **volatile**. Ce mot-clé permet de s'assurer que les modifications apportées à une variable par un thread sont immédiatement visibles par les autres threads. Pour ce faire, lorsque qu'une variable volatile est modifiée, la valeur est directement

écrite en mémoire principale, et les lignes de cache où elle est présente sont invalidées. Cela force les prochains accès à la variable à se faire directement en mémoire principale, garantissant ainsi la visibilité des modifications. L'utilisation du mot-clé volatile permet d'obtenir une relation importante : la relation ***happens-before*** [49]. Cette relation offre deux garanties essentielles : *(i)* Les lectures et écritures sur d'autres variables effectuées avant une écriture sur une variable volatile ne peuvent pas être réordonnées après cette écriture. Il est cependant important de noter que cette restriction n'est valable que dans un sens : les lectures et écritures qui se produisent après l'écriture sur une variable volatile peuvent être réordonnées avant cette dernière. *(ii)* Les lectures et écritures sur d'autres variables effectuées après une lecture sur une variable volatile ne peuvent pas être réordonnées avant cette lecture. En revanche, les lectures sur d'autres variables qui ont eu lieu avant cette lecture peuvent être réordonnées après la lecture sur la variable volatile. On parle alors de critère de cohérence volatile. Nous définissons plus précisément la notion de critère de cohérence en Section 2.4.1.

Une utilisation courante du critère de cohérence volatile est dans les scénarios où une variable est lue par plusieurs threads, mais modifiée par un seul.

En résumé, le mot-clé volatile en Java garantit la visibilité des modifications d'une variable entre différents threads et établit un ordre partiel sur les opérations de mémoire. Cependant, il ne résout pas tous les problèmes liés à la concurrence. En effet, l'ordre des opérations sur plusieurs variables marquées comme **volatile** n'est pas garanti.

En Java, il existe plusieurs critères de cohérence qui peuvent être spécifiés à l'aide des mécanismes de *VarHandles*. Ces modes incluent *Plain*, *Opaque*, *Release/Acquire*, et *Volatile*, classés par ordre croissant de contraintes imposées. Ces différents critères de cohérences seront expliqués plus en détail quand nous en ferons usage, à savoir dans le Chapitre 4.

Lorsqu'un thread cherche à modifier une ressource partagée, il doit d'abord obtenir un accès exclusif, par exemple en acquérant un **verrou** (ou *mutex*). Ce mécanisme assure que l'accès à la ressource est exclusif, évitant ainsi les *race conditions*. Ces dernières surviennent lorsque plusieurs threads tentent simultanément de modifier une ressource partagée, ce qui peut entraîner des comportements indésirables et une corruption des données.

Une fois le verrou acquis, le thread est le seul à pouvoir modifier la ressource. À l'issue de ses opérations, il libère le verrou, permettant à d'autres threads d'en solliciter l'acquisition à leur tour. Ce processus de verrouillage et de déverrouillage assure une synchronisation stricte entre les différents threads qui partagent la ressource.

En Java, un thread peut acquérir un verrou de manière explicite grâce à l'interface ***Lock***, qui offre une API flexible pour gérer les verrous. Un thread a la possibilité de tenter d'obtenir un verrou de façon bloquante ou non bloquante, avec la possibilité d'interruption si un autre thread interrompt celui en attente. De plus, l'interface permet de différencier les accès en lecture et en écriture, garantissant ainsi que plusieurs threads peuvent lire simultanément tandis qu'un seul peut écrire à un instant donné.

Alternativement, le mot-clé ***synchronized*** peut être utilisé pour restreindre l'accès à une section de code à un thread à la fois, que ce soit pour une méthode entière ou un bloc de code spécifique. Une classe qui utilise ce mécanisme de synchronisation est désignée comme un **moniteur** [33].

```
String text = /* Some text */;
String[] words = text.split(" ");
int numberOfThreads = /* N */;
int segmentSize = (int) Math.ceil((double) words.length /
    numberOfThreads);

ExecutorService executor =
    Executors.newFixedThreadPool(numberOfThreads);
ArrayList<Runnable> tasks = new ArrayList<>();
Map<String, Integer> globalCounter = new HashMap<>();

for (int i = 0; i < numberOfThreads; i++) {
  final int start = i * segmentSize;
  final int end = Math.min(start + segmentSize, words.length);
  tasks.add(() -> {
    Map<String, Integer> localCounter = new HashMap<>();
    for (int j = start; j < end; j++)
      localCounter.merge(words[j], 1, Integer::sum);
    synchronized (globalCounter) {
      localCounter.forEach((word, count) ->
    globalCounter.merge(word, count, Integer::sum));
    }
  });
}
for (Runnable task : tasks)
  executor.execute(task);
executor.shutdown();
globalCounter.forEach((word, count) -> System.out.println(word +
    ": " + count));


```

Listing 2.1 – Traitement parallèle du comptage de mots dans un texte avec `ExecutorService`, `Runnable` et `Synchronyzed`

Dans le Listing 2.1, on présente un exemple de code qui exécute des tâches en parallèles en utilisant des objets `Runnable` au sein d'un pool de threads géré par la classe `ExecutorService`. Afin de garantir une meilleure lisibilité, le code comprenant la lecture du fichier contenant le texte n'est pas inclus. Dans cet exemple, la variable `text` est partagée entre plusieurs threads qui comptabilisent localement les occurrences des mots dans leurs segments respectifs. Par la suite, les threads

mettent à jour une `HashMap` partagée de manière cohérente grâce à l'utilisation d'un bloc `synchronized`, garantissant ainsi l'absence de conflits lors de l'accès concurrent à la variable `globalCounter` (ligne 18).

Dans cet exemple, chaque thread accède individuellement à la variable partagée. Toutefois, il pourrait également être pertinent d'autoriser un nombre limité de threads à accéder simultanément à cette variable, notamment lorsque l'opération tolère un certain degré de parallélisme sans compromettre la cohérence des données. Cette approche peut convenir dans les cas où plusieurs opérations de lecture peuvent être effectuées en parallèle, ou lorsqu'un accès simultané, mais contrôlé, est nécessaire pour certains threads sur des ressources partagées. Cela peut être géré de manière efficace en ayant recours à l'utilisation d'un **sémaphore** [15].

Un sémaphore est un mécanisme de synchronisation conçu pour réguler l'accès à une ressource partagée en maintenant un ensemble de "permis". Ces permis représentent des jetons que les threads doivent acquérir, à l'instar d'un verrou, avant de pouvoir accéder à la ressource concernée. Une fois leur tâche terminée, ils sont tenus de relâcher le permis, permettant ainsi à d'autres threads d'y accéder.

Les sémaphores offrent deux modes d'acquisition des permis : bloquant et non bloquant. Dans le mode bloquant, si aucun permis n'est disponible, un thread est mis en attente jusqu'à ce qu'un autre thread libère un permis. Cela garantit que l'accès à la ressource est strictement contrôlé, mais peut introduire des délais d'attente. En revanche, dans le mode non bloquant, si un permis n'est pas disponible, le thread poursuivra son exécution sans attendre, évitant ainsi des blocages inutiles et augmentant la réactivité dans certains scénarios.

Lors de l'initialisation d'un sémaphore, il est possible de spécifier le nombre maximal de permis disponibles, offrant ainsi une grande flexibilité dans la gestion des ressources. Par exemple, un sémaphore initialisé avec un seul permis est communément appelé un *sémaphore binaire*, agissant de manière similaire à un *mutex* en garantissant un accès exclusif à un seul thread à la fois.

Un avantage notable de ce mécanisme est la possibilité, pour un thread, d'interrompre un autre thread détenant un permis, réduisant ainsi le risque de blocage permanent (*deadlock*). Cette capacité à forcer la libération des ressources en cas d'interruption permet de concevoir des systèmes plus robustes face aux erreurs ou aux défaillances des threads.

```
1  /* ... */
2  Map<String, AtomicInteger> globalCounter = new
     ConcurrentHashMap<>();
3
4  tasks.add(() -> {
5    for (int j = start; j < end; j++)
6      globalCounter.merge(word, new AtomicInteger(1), (obj, _) -> {
7        obj.incrementAndGet();
8        return obj;
9      });
10 });
11
12 /* ... */
13
```

Listing 2.2 – Traitement parallèle du comptage de mots dans un texte avec `ExecutorService`, `Runnable` et `Synchronyzed`

Enfin, les sémaphores peuvent être configurés pour garantir une attribution équitable des permis, par exemple en les accordant dans l'ordre des demandes. L'utilisation d'un sémaphore introduit une relation de type *happens-before* entre les opérations d'un thread qui relâche un permis et celles d'un autre thread qui acquiert ce même permis, assurant ainsi une synchronisation précise entre les différents accès à la ressource partagée.

Limiter le nombre de threads accédant à une ressource permet d'éviter les conflits d'accès et d'améliorer les performances. Pour contrôler le débit des opérations dans un système, les ***TimedSemaphores*** sont particulièrement adaptés [72]. Ces sémaphores libèrent automatiquement les permis après un délai prédéfini, ce qui permet de limiter le nombre d'opérations par unité de temps.

L'utilisation de ces différentes primitives de synchronisation peut toutefois poser un problème : l'impossibilité de garantir un accès concurrent aux données. Pour résoudre cette difficulté, les développeurs ont recours à des objets partagés. Ces objets assurent que, même dans un contexte d'exécution parallèle, les opérations effectuées sur eux se comportent de manière identique à celles réalisées en séquentiel.

On peut voir dans le Listing 2.2 une utilisation d'un objet concurrent. Il est par exemple possible de substituer la `HashMap` utilisée à la ligne 8 du Listing

2.1 par un objet de type `ConcurrentHashMap`. En optant pour cette structure de données concurrente, on élimine la nécessité d'utiliser un bloc `synchronized` pour toute la structure. Ainsi, les différents threads peuvent partager leurs résultats sans avoir à attendre les uns après les autres, ce qui améliore l'efficacité de l'exécution parallèle. Nous avons testé les deux méthodes avec un texte de 1 Gigaoctet et la méthode décrite dans le Listing 2.2 est 3.41× plus rapide que celle qui utilise le bloc `synchronized`.

Alors que l'utilisation de `synchronized` impose un accès séquentiel aux données partagées, limitant les threads à une utilisation exclusive de la variable à chaque instant, `ConcurrentHashMap` offre un accès concurrentiel optimisé. Dans la section suivante, nous explorons les propriétés assurant que ce type de gestion du parallélisme ne compromet pas la correction du programme.

Résumé

Dans cette section, nous avons exploré différentes méthodes de création et de gestion du parallélisme en Java, en commençant par un rappel des paradigmes de **programmation procédurale** et **orientée objet**.
Nous avons introduit le rôle du **Runtime Java** et expliqué comment la **JVM** gère l'exécution des programmes à travers la **compilation Just-In-Time** (JIT) et la gestion de la mémoire via le **garbage collector**.
Nous avons ensuite détaillé les mécanismes de création de threads, que ce soit par l'héritage de la classe ***Thread*** ou l'utilisation des interfaces ***Runnable*** et ***Callable***, en mettant l'accent sur les limitations de l'héritage. Nous expliquons aussi l'intérêt d'utiliser des promesses via les **CompletableFuture** pour composer les tâches asynchrones. Par ailleurs, nous avons expliqué le fonctionnement des **pools de threads** et des **flux parallèles**, tout en abordant l'introduction des **threads virtuels**, qui permettent de créer un grand nombre de threads légers sans surcharge excessive du système.
Enfin, nous avons étudié les mécanismes de synchronisation comme le mot-clé **volatile**, les **verrous** (*mutex*) ou encore les **sémaphores**, indispensables pour garantir l'intégrité des données dans les environnements multithreads.

2.4 Objets partagés

Dans la section précédente, nous avons exploré diverses approches pour créer et gérer le parallélisme en Java, notamment la création de threads et les techniques de synchronisation.

En programmation orientée objet, un objet séquentiel assure un accès unique à ses données. Toutefois, lorsqu'un objet doit être manipulé par plusieurs threads en parallèle, il doit être adapté pour garantir un comportement correct malgré les accès concurrents. C'est le cas de `ConcurrentHashMap`, abordé dans la section précédente, qui permet un accès sécurisé et performant aux données partagées

Dans cette section, nous présentons plusieurs propriétés de correction qui permettent de prédire et de contrôler le comportement des objets partagés, avant d'examiner différentes conditions de progrès, essentielles pour assurer que le programme puisse aboutir à une exécution complète.

2.4.1 Propriétés de corrections

Prenons l'exemple d'une file d'attente comme objet partagé. Lorsqu'un thread invoque une opération sur cette file, par exemple une opération d'extraction, il effectue une série d'opérations sur les registres. Cette opération d'extraction, bien que conceptuellement simple, implique plusieurs lectures et écritures et, par conséquent, n'est pas intrinsèquement atomique, c'est à dire qu'elle n'est pas considéré comme indivisible et ininterrompue.

L'absence d'atomicité signifie que les opérations sur les objets partagés peuvent être réarrangées de manière arbitraire par les différents threads. Ce comportement est source de complexité et de risques dans la conception des programmes parallèles. Par exemple, lorsque deux threads tentent d'extraire simultanément l'élément en tête d'une file d'attente, il est légitime de se demander quel thread parviendra à accéder à cet élément. Mais au-delà de cela, nous pouvons nous interroger : l'objet récupéré sera-t-il vraiment valide ? Correspondra-t-il à une référence d'un élément inséré précédemment dans la file ? Ou encore, que se passe-t-il si cet élément est modifié par un autre thread pendant le processus d'extraction ?

Ces ambiguïtés soulèvent des questions critiques : comment garantir que les

opérations concurrentes n'entraînent pas des incohérences ou des comportements inattendus ? Comment s'assurer que chaque thread accède aux données de manière prévisible et correcte ?

Pour s'assurer qu'une opération ou de manière générale un programme respecte des principes qui facilitent leur compréhension, on établit ce qu'on appelle des **critères de cohérence**. Ces critères sont des ensembles de règles claires qui guident le développeur dans la création du programme, offrant un cadre pour anticiper et maîtriser le comportement du système.

Ces critères sont cruciaux, car ils réduisent l'incertitude lors de l'exécution du programme, en garantissant que les opérations suivent une logique prévisible.

La relation entre les règles de cohérence et la complexité de leur implémentation dans les systèmes multiprocesseurs est un équilibre délicat. Plus les règles qui dictent le comportement des opérations sont strictes, plus leur mise en œuvre sera complexe, ce qui peut entraîner une diminution des performances des opérations. En effet, une cohérence stricte implique souvent des mécanismes de synchronisation complexe, ce qui peut ralentir l'exécution des programmes.

On parle de **cohérence forte** lorsqu'une telle rigueur est appliquée : toutes les opérations doivent être perçues par tous les threads dans le même ordre, comme si l'exécution du programme se déroulait de manière séquentielle. Dans ce modèle, chaque thread observe les opérations dans un ordre global et identique, ce qui simplifie la programmation en rendant le comportement du système prévisible et intuitif. Cependant, cette approche peut nécessiter des mécanismes de synchronisation coûteux et peut réduire les performances globales du système.

À l'inverse, la **cohérence faible** n'impose aucune garantie stricte sur l'ordre dans lequel les opérations sont perçues d'un thread à l'autre. Ce niveau de cohérence permet une plus grande flexibilité et des performances améliorées, car il n'y a pas besoin de maintenir un ordre global strict des opérations. En conséquence, les threads peuvent fonctionner de manière plus autonome et optimisée. Toutefois, cette flexibilité comporte un coût en termes de prévisibilité. Les effets induits par les opérations deviennent plus difficiles à anticiper, rendant la programmation plus complexe et potentiellement exposée à des erreurs difficiles à diagnostiquer.

La complexité d'utilisation et le manque de prévisibilité de la cohérence faible incitent fréquemment les développeurs à privilégier des modèles de cohérence forte,

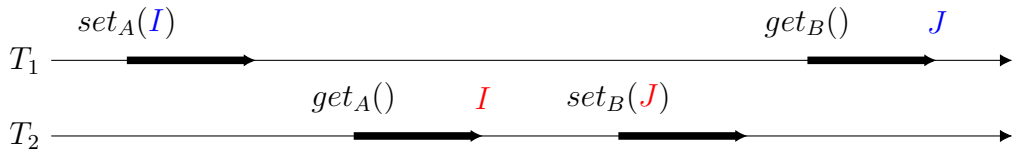

FIGURE 2.3 – Exécution linéarisable d'un objet `AtomicReference` avec les opérations `set(x)` et `get()` pour la mise à jour et la récupération de références d'objets. I et J sont des références vers deux objets.

qui permettent de garantir un comportement plus déterministe et fiable des programmes. Parmi ces modèles, on y trouve la **linéarisabilité** [29].

La linéarisabilité impose un cadre rigoureux dans lequel les opérations sur un objet partagé semblent se produire instantanément à un moment précis, situé quelque part entre l'invocation de l'opération et la réception de sa réponse. En d'autres termes, chaque opération sur l'objet partagé est perçue comme ayant un effet immédiat à un instant unique, tout en respectant l'ordre des opérations tel qu'elles apparaissent dans le programme. On peut par exemple observer dans la Figure 2.3 que, après avoir respectivement mis à jour les objets `Référence` A et B, les threads $T1$ et $T2$ sont en mesure de constater les effets de bord des opérations effectuées par l'autre.

Ce modèle offre donc une illusion d'exécution atomique, où l'intégrité de l'état de l'objet est préservée à tout moment. Le moment précis où une opération semble prendre effet est connu sous le nom de **point de linéarisation**. Ce point marque l'instant où tous les effets de bord d'une opération deviennent observable par les autres opérations exécutées sur l'objet partagé. En pratique, le point de linéarisation correspond au moment où la modification de l'état de l'objet devient visible pour les autres threads, assurant ainsi que tous les observateurs partagent une vue cohérente et synchronisée de l'état de l'objet.

Une manière alternative d'obtenir un critère de cohérence plus flexible que la linéarisabilité consiste à ne pas tenir compte de l'ordre temporel réel des événements dans le programme. Ce type de cohérence est connu sous le nom de **cohérence séquentielle** [46]. Ce critère de cohérence garantit que chaque unité de calcul exécute ses opérations dans l'ordre spécifié par le programme, sans se

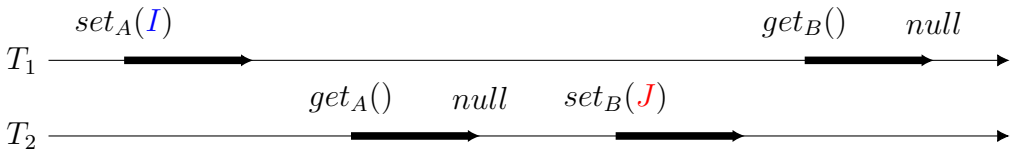

FIGURE 2.4 – Exécution séquentiellement cohérente d'un objet `AtomicReference`.

préoccuper de l'ordre temporel global observé sur l'ensemble des threads. Puisque l'ordre mutuel des opérations entre différents threads n'est pas strictement maintenu, les séquences d'opérations des différents threads peuvent être réarrangées les unes par rapport aux autres, permettant ainsi une plus grande flexibilité dans l'exécution du programme. Comme l'illustre la Figure 2.4, les threads $T1$ et $T2$ renvoient tous deux la valeur *null* alors que les écritures ont eu lieu avant. Cette exécution ne peut pas être considérée comme linéarisable, car dans une telle situation, il serait impératif que les threads observent les écritures effectuées en amont, ce qui n'est manifestement pas le cas ici. Toutefois, cette exécution demeure séquentiellement cohérente, car il est possible de reconstituer une histoire séquentielle qui respecte l'ordre d'exécution propre à chaque processus. Par exemple $\text{T2}.get_A() = null \rightarrow \text{T1}.set_A(i) \rightarrow \text{T1}.get_B() = null \rightarrow \text{T2}.set_B(j)$. Dans un système séquentiellement cohérent, les threads observent tous les opérations dans le même ordre.

Bien que la cohérence séquentielle offre un modèle rigoureux et intuitif pour la gestion des accès mémoire, elle peut parfois être trop contraignante. Dans certains cas, un modèle moins strict tel que la **cohérence causale** [2] peut suffire.

Dans certains contextes, il est possible d'observer deux opérations effectuées dans des ordres différents sans que cela ne pose de problème. Illustrons ce point avec un exemple : considérons les commentaires laissés sur des articles dans un magasin en ligne. Si deux commentaires sur deux articles distincts apparaissent dans des ordres différents, cela ne pose pas de problème fondamental pour l'utilisateur. En revanche, lorsque l'on reçoit une réponse à un commentaire, suivie d'une seconde réponse à la première, il est crucial que l'ordre dans lequel ces deux réponses sont affichées soit maintenu pour des raisons évidentes de logique et de clarté. La

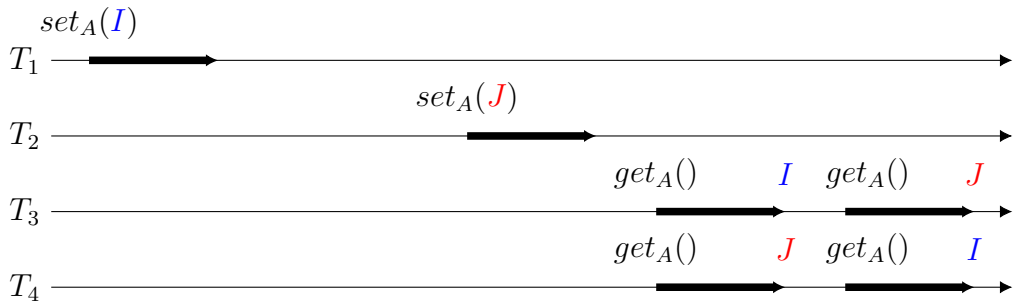

FIGURE 2.5 – Exécution causalement cohérente d'un objet `AtomicReference`.

cohérence causale assure que cette relation soit respectée.

Pour établir un ordre entre les opérations selon la cohérence causale, deux étapes sont essentielles. Premièrement, des relations d'ordre direct se forment lorsque le même thread exécute plusieurs opérations, celles-ci étant réalisées dans l'ordre d'exécution. Deuxièmement, des relations d'ordre indirect sont établies lorsqu'une opération X d'un thread A perçoit les effets d'une opération Y d'un thread B, ce qui signifie que Y est considérée comme effectuée avant X. Par conséquent, toutes les opérations qui suivent directement X, et qui sont liées à celle-ci, se situent après Y.

La cohérence causale implique donc que les opérations effectuées par un même thread respectent l'ordre séquentiel dans lequel elles ont été réalisées. On peut voir dans la Figure Figure 2.5 que l'exécution n'est pas séquentiellement cohérente car il est impossible de réarranger les lectures de $T3$ et $T4$ pour obtenir une exécution séquentielle valide sans modifier l'ordre des lectures au sein du même thread. En revanche, étant donné qu'aucune relation d'ordre n'est établie entre les écritures réalisées par $T1$ et celles effectuées par $T2$, cette exécution peut être considérée comme causalement cohérente. Une autre *AntidoteDB* [6] est une base de données spécialement conçue pour garantir la cohérence causale. Chaque opération y est associée à une horloge vectorielle [49], permettant d'assurer que seules les opérations présentant une relation causale directe sont exécutées dans un ordre strict, tandis que celles qui ne sont pas liées causalement peuvent être traitées de manière asynchrone.

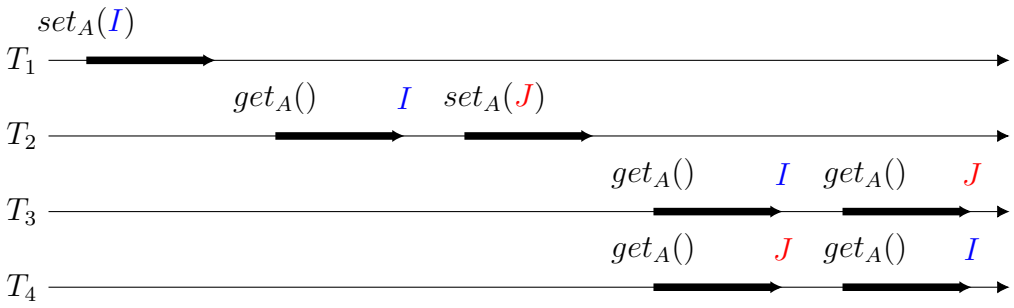

FIGURE 2.6 – Exécution *FIFO* cohérente d'un objet `AtomicReference`.

La **cohérence *FIFO*** [12] (*First-In-First-Out*) établit un ordre uniquement entre les opérations d'écriture effectuées par un même thread, sans imposer de contraintes entre les écritures réalisées par différents threads. On peut implémenter ce critère de cohérence de la manière suivante : chaque thread dispose d'un espace mémoire réservé dans lequel il est exclusivement habilité à effectuer des écritures, et ce, sans avoir à se coordonner avec les autres threads. Lorsqu'un thread désire écrire dans une zone de mémoire qui ne lui est pas propre, il doit notifier le thread responsable de cette zone via une file de communication. De ce fait, les écritures d'un même thread sont maintenues dans un ordre strict, tandis que les écritures entre plusieurs threads peuvent être perçues dans un ordre différent et désordonné. Dans la Figure Figure 2.6, nous avons ajouté une lecture avant l'écriture effectuée par le thread $T2$ en comparaison avec l'exécution présentée dans la Figure 2.5. Cette modification engendre ainsi une relation de causalité entre les écritures des threads $T1$ et $T2$, signifiant que l'écriture réalisée par $T1$ doit nécessairement être observée avant celle de $T2$. Cependant, ce n'est pas le cas pour le thread $T3$, ce qui entraîne une violation de la cohérence causale dans cette exécution. En revanche, cette exécution respecte la cohérence *FIFO*, car l'ordre des écritures effectuées par les threads $T1$ et $T2$ peut être perçu différemment par les threads $T3$ et $T4$. En effet, ces derniers ont été informés des écritures de $T1$ et $T2$ à travers des files de communication distinctes, permettant ainsi à chaque thread de percevoir les écritures dans un ordre propre à son propre espace mémoire.

La **cohérence éventuelle** [75] est une forme de cohérence peu restrictive,

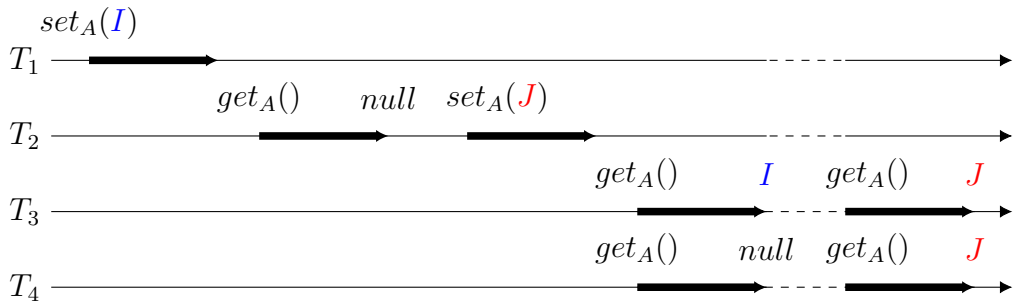

FIGURE 2.7 – Exécution éventuellement cohérente d'un objet `AtomicReference`.

offrant des possibilités de calcul rapide. La seule garantie qu'elle propose est que, lorsqu'aucune nouvelle opération d'écriture n'est effectuée sur une donnée, toutes les lectures de cette donnée finiront par retourner la dernière valeur écrite après un certain délai. En revanche, elle ne fournit aucune garantie concernant l'ordre dans lequel les opérations sont vues par les threads.

La Figure 2.7 montre que les threads $T3$ et $T4$ ne voient pas tout de suite l'effet de l'opération `set` effectuée par $T1$. Néanmoins, ils lisent la même valeur mise à jour après un certain temps.

On distingue deux variantes de la cohérence éventuelle : la cohérence éventuelle forte [73] et la cohérence éventuelle faible, souvent désignée simplement par cohérence éventuelle. La distinction repose sur la manière dont les conflits sont gérés. En cohérence éventuelle forte, toutes les répliques finissent par converger vers un état commun, même en présence de conflits. Cela implique que soit les effets des opérations concurrentes sont commutatifs, soit un mécanisme de gestion des conflits garantit que ces opérations sont appliquées dans le même ordre. À l'inverse, en cohérence éventuelle faible, la résolution des conflits est plus arbitraire, ce qui peut entraîner des incohérences temporaires dans la manière dont les différents threads perçoivent les effets des opérations concurrentes.

Bien que la cohérence éventuelle puisse être difficile à appréhender dans certains systèmes, elle peut offrir des garanties suffisantes dans des contextes spécifiques. Par exemple, dans un réseau social où l'on souhaite compter le nombre de *likes*, l'ordre dans lequel ces *likes* sont comptés n'a pas d'importance fondamentale.

2.4.2 Conditions de progrès

Lorsqu'un programme s'exécute, il ne suffit pas de garantir que les données respectent un critère de cohérence. Il est également crucial de veiller à ce que le programme progresse efficacement. À l'instar des critères de cohérence, il est possible de définir des conditions de progrès, plus ou moins strictes, qui spécifient les circonstances sous lesquelles les threads avancent dans l'exécution du programme.

Ces conditions de progrès peuvent être soit dépendantes, soit indépendantes de l'avancement des autres threads dans le système. Par exemple, une opération qualifiée de ***wait-free*** [24] garantit que chaque thread peut terminer son opération en un nombre fini d'étapes, indépendamment de la vitesse d'exécution des autres threads. Une propriété notable d'une opération *wait-free* est sa tolérance aux fautes : aucun thread exécutant une opération *wait-free* ne peut être bloqué en raison de la lenteur ou de la défaillance d'un autre thread. Si cette opération est assurée de se terminer en un nombre borné d'étapes, on la qualifie de ***bounded wait-free***.

Une autre condition de progrès importante est la condition ***lock-free***. Un programme *lock-free* garantit qu'à tout moment, au moins une opération est certaine de se terminer en un nombre fini d'étapes. On peut ainsi observer qu'une exécution *wait-free* est nécessairement *lock-free*, bien que l'inverse ne soit pas forcément vrai.

Enfin, lorsqu'une opération est exécutée en isolement — c'est-à-dire sans qu'aucun autre thread n'effectue d'étapes de calcul en parallèle — et qu'elle se termine en un nombre fini d'étapes, on parle alors d'une opération ***obstruction-free***. Ce modèle de progrès est moins contraignant que le *lock-free* et le *wait-free*, car il ne garantit le progrès que dans l'absence de concurrence. Toute exécution *lock-free* est *obstruction-free* mais l'inverse n'est pas vrai.

Ces distinctions permettent de mieux comprendre et formaliser les garanties de progression offertes par différents algorithmes ou structures de données dans un contexte concurrentiel, et soulignent l'importance de choisir le modèle de progrès le plus adapté aux besoins spécifiques de chaque application.

Résumé

Dans cette section, nous avons étudié les différentes propriétés de correction qui permettent d'anticiper et de maîtriser le comportement des objets partagés. Parmi ces propriétés figurent la **linéarisabilité**, la **cohérence séquentielle** et la **cohérence causale**, qui assurent un certain ordre d'exécution des opérations. Chaque approche propose un équilibre entre rigueur et performance : la linéarisabilité offre un ordre strict et prévisible, tandis que des modèles plus flexibles, comme la cohérence causale ou **éventuelle**, favorisent l'optimisation en échange d'une complexité plus importante.
Nous avons également examiné les conditions de progrès, telles que les propriétés **wait-free**, **lock-free** et **obstruction-free**, qui garantissent que les threads continuent à progresser efficacement sans être bloqués, même en situation de concurrence. Ces conditions sont essentielles pour s'assurer de l'achèvement correct des opérations dans les programmes parallèles.

2.5 Approches récentes

Après avoir examiné en détail les propriétés de correction et les conditions de progrès qui permettent de garantir le bon comportement des objets partagés, il est maintenant crucial de se tourner vers des approches récentes qui cherchent à optimiser l'accès concurrent aux objets partagés. C'est ce que nous allons faire dans la suite de cette section.

Dans la Section 2.3.4, nous avons présenté plusieurs mécanismes de synchronisation, chacun avec ses avantages et inconvénients. Par exemple, bien que l'utilisation des verrous soit simple à mettre en œuvre, elle peut conduire à des situations de *deadlock* lorsque plusieurs threads tentent d'acquérir les verrous sur un même ensemble d'objets. Bien que ces problèmes puissent souvent être évités grâce à l'expertise des développeurs, la complexité croissante des programmes parallèles rend cette tâche de plus en plus difficile. Un autre défi réside dans la difficulté de composer ces primitives de synchronisation. Prenons l'exemple d'un système où les tâches sont gérées dans des files de priorité. Si l'on souhaite transférer une tâche d'une file à une autre de manière atomique, c'est-à-dire qu'elle doit être soit

dans l'une, soit dans l'autre file, sans jamais être dans les deux ou dans aucune d'entre elles, cela devient complexe. Même si chaque file est protégée par sa propre primitive de synchronisation, il est difficile de combiner ces primitives. Les threads pourraient essayer d'acquérir les deux verrous, mais cela crée un risque de *deadlock* si les verrous ne sont pas obtenus dans le même ordre. Une autre solution serait d'utiliser un seul verrou pour garantir un accès exclusif aux deux files, mais cela pourrait introduire un goulot d'étranglement, limitant la performance. De plus, cette approche nécessiterait de connaître à l'avance les deux files que l'on souhaite manipuler.

Il existe un modèle de programmation capable de résoudre ce type de problème : **la mémoire transactionnelle**.

La mémoire transactionnelle

La mémoire transactionnelle [26] repose sur l'utilisation de transactions entre différents threads, permettant d'exécuter plusieurs opérations de manière atomique et ordonnée. Autrement dit, ces transactions sont **sérialisables**, ce qui signifie que les effets des opérations apparaissent instantanément, comme si elles avaient été réalisées en une seule étape, entre l'appel de la première opérations et la réponses de la dernières opérations. À haut niveau, on définit une transaction en utilisant un bloc *atomic*, qui est exécuté de manière spéculative. Cela signifie que le thread va tenter d'apporter des modifications, et si la transaction se termine sans conflits, les changements sont validés (*committed*) et rendus permanents. En cas de conflit, les modifications seront annulées.

L'équilibre entre cohérence et performance

Bien que les objets partagés offrent un niveau d'abstraction élevé, simplifiant ainsi la programmation parallèle, leur utilisation soulève néanmoins de nouvelles questions sur les choix de conception à adopter et les compromis inévitables à envisager. En effet, la sélection des propriétés de correction joue un rôle crucial dans l'impact sur les performances d'un programme. Choisir des critères de cohérence forts, comme la linéarisabilité, non seulement garantit un comportement plus prévisible, mais facilite également la programmation. En imposant un ordre strict et

global des opérations, ces modèles réduisent la complexité mentale pour les développeurs, qui n'ont pas à anticiper les effets imprévisibles des réarrangements d'opérations. Cette prévisibilité simplifie la détection d'erreurs et le raisonnement sur le programme, bien que cela puisse se faire au détriment de la performance. Ainsi, bien que des modèles plus souples puissent offrir des gains d'efficacité, ils nécessitent une plus grande expertise pour être mis en œuvre correctement, en raison des comportements non déterministes qu'ils peuvent introduire.

C'est pourquoi les chercheurs s'efforcent de proposer des propriétés innovantes capables de concilier une cohérence suffisamment forte pour garantir un raisonnement correct, tout en n'imposant pas de contraintes excessives qui nuiraient à la performance globale du système. Ainsi, en fonction de la nature spécifique des programmes, il est possible d'ajuster et de moduler la propriété de correction en conséquence.

La Linéarisabilité des Valeurs Intermédiaires

Rinberg and Keidar [66] introduisent dans leurs travaux un concept novateur selon lequel, dans certains contextes, une valeur intermédiaire entre deux opérations de lecture pourrait être considérée comme valable. Ils formulent ainsi un nouveau critère de cohérence, nommé **linéarisabilité des valeurs intermédiaires**, selon lequel toute valeur de retour située entre deux valeurs de retour légitimes dans une exécution linéarisable est considérée comme valide.

Prenons pour illustration l'exemple d'un compteur initialisé à 0, où un thread t effectue une opération de lecture en parallèle à un autre thread q, qui réalise une seule opération consistant en trois incréments. Dans une exécution classique linéarisable, l'opération de lecture de t pourrait être linéarisée soit avant soit après l'opération de q, offrant ainsi à t une valeur de retour de 0 ou 3. En revanche, dans une exécution linéarisable des valeurs intermédiaires, le thread t peut retourner n'importe quelle valeur x telle que $0 \leq x \leq 3$. Cette souplesse permet d'employer des implémentations plus efficaces, aboutissant à une amélioration significative des performances.

Dans l'exemple précédent, on pourrait envisager une implémentation spécifique d'un compteur, où chaque thread disposerait de son propre registre ($SWMR$) per-

mettant une écriture unique mais plusieurs lectures. Cette approche sera déclinée en plusieurs variantes dans la suite de cette thèse. Il est bien connu que, pour utiliser une telle construction, il est nécessaire d'implémenter un algorithme de *snapshot* afin d'obtenir une vue cohérente de l'objet dans son ensemble. Nous savons également que, pour mettre à jour un tel objet tout en s'assurant qu'il soit linéarisable, il faut effectuer $\Omega(n)$ pas de calcul dans un système comportant n threads [16, 35]. En revanche, dans une exécution linéarisable des valeurs intermédiaires, il est possible d'effectuer une mise à jour en $O(1)$ pas de calcul.

La Quasi-Linéarisabilité

Un autre critère de cohérence plus flexible que la linéarisabilité est la **quasi-linéarisabilité**. Afek et al. [1] introduisent ce concept qui repose sur l'idée qu'une exécution H, même si elle n'est pas linéarisable, peut néanmoins être relativement proche d'une exécution H' qui est, elle, linéarisable. Pour quantifier cette notion de proximité ou de distance, on considère que S et S' sont respectivement des linéarisations (qu'elles soient légales ou non) d'exécutions parallèles H et H'. On note $S[o]$ (respectivement $S'[o]$) l'indice de l'opération o dans l'histoire S (respectivement S'). Ainsi, on peut dire que $S[o1] < S[o2]$ si, et seulement si, $o1$ précède $o2$ dans S.

La mesure de la distance entre deux séquences S et S' n'est possible que si ces séquences sont des permutations l'une de l'autre. Cette distance est évaluée en observant l'indice de chaque opération dans chacune des deux séquences, puis en calculant la différence absolue entre ces indices. La distance entre les deux séquences est alors définie comme la différence maximale observée pour une opération donnée de S (et donc de S').

De cette manière, une séquence devient légale si elle est à une distance bornée d'une séquence respectant la spécification séquentielle de l'objet. Chaque opération définie dans la spécification séquentielle de l'objet O est associée à un **facteur de quasi-linéarisation**. Ce facteur représente la limite supérieure de distance permise pour cette opération, déterminant ainsi si, dans le cas où cette opération possède la plus grande distance entre les deux séquences, la séquence est valide ou non.

Soit Q l'ensemble des facteurs de quasi-linéarisation d'une histoire H. On dit alors que H est **Q-quasi-linéarisable** si chaque opération présente dans H possède une distance inférieure à son facteur de quasi-linéarisation respectif.

L'exemple précédent du compteur partagé, employé dans un contexte de monitoring est un objet pouvant se contenter d'un critère de cohérence quasi-linéarisable. Dans cette situation, la précision exacte n'est pas nécessaire ; il est davantage crucial d'obtenir une idée générale de la tendance.

Un autre exemple pertinent est une file de tâches partagée par plusieurs threads. Dans ce scénario, les threads n'ont pas besoin de récupérer les tâches dans l'ordre exact où elles ont été ajoutées. Ainsi, on peut autoriser les threads à récupérer une tâche parmi les k premières de la file, ce qui réduit la contention sur la tête de la file et améliore l'efficacité globale du système. Cela permet non seulement de répartir les tâches de manière plus souple mais aussi d'optimiser les ressources disponibles pour un meilleur fonctionnement.

Adapter la propriété de correction en fonction des opérations spécifiques permet un choix plus précis des objets utilisés, en s'alignant avec les besoins uniques de chaque opération plutôt que d'imposer une seule propriété de correction à l'ensemble du programme.

La cohérence des objets promesses

Kogan et Herlihy [42] suggèrent d'étendre ces propriétés de correction aux objets promesses, objets que nous avons introduits en Section 2.1. Ces objets étant spécifiquement conçus pour exécuter des tâches en parallèle et de manière asynchrone. Ils introduisent donc trois niveaux de correction, tous dérivés de la linéarisabilité : la linéarisabilité **forte**, **moyenne** et **faible des promesses**.

La linéarisabilité forte des promesses est équivalente à la linéarisabilité telle que nous la connaissons. Les opérations sont évaluées entre le moment où la promesse est créé et le moment où elle est attribué à une variable. Ainsi, si trois opérations sont associées respectivement aux promesses `a`, `b`, et `c`, elles seront exécutées dans l'ordre exact dans lequel elles ont été invoquées. Ce critère de cohérence n'offre pas les meilleures performances mais a l'avantage d'être très simple à utiliser et à comprendre.

La linéarisabilité faible des promesses garantit des performances optimales en autorisant une large gamme de réordonnancements d'opérations. En effet, celles-ci peuvent s'exécuter à tout moment entre la création et l'évaluation de la promesse. Ainsi, si trois promesses `a`, `b` et `c` sont créées puis évaluées dans cet ordre, l'ordre exact de prise d'effet des opérations demeure imprévisible.

La linéarisabilité moyenne des promesses, comme son nom l'indique, se situe entre les deux précédents niveaux de correction. Dans ce cas, tout comme pour la linéarisabilité faible des promesses, les opérations sont évaluées entre la création de la promesse et son évaluation. Cependant, si deux promesses sont créés par un même thread sur un même objet, alors ces deux opérations seront évaluées dans l'ordre dans lequel leurs promesses associées ont été créées.

Après avoir introduit ces niveaux de correction, les auteurs proposent plusieurs algorithmes d'objets basiques comme la file, la pile ou encore la liste chaînée qui sont capables de tirer avantage de la linéarisabilité forte, moyenne et faible des promesses. Pour améliorer les performances du programme, la manière dont les opérations sont évaluées peut être optimisée grâce à l'utilisation des promesses. On peut prendre l'exemple d'une pile. Dans son implémentation à linéarisabilité faible des promesses, les threads ont chacun une file unique d'opérations en attente lorsqu'une promesse est créée. Quand un thread effectue une opération `push(x)`, il vérifie s'il y a des opérations `pop()` dans sa liste des opérations en attente. Si c'est le cas, il attribue la valeur x à une des opérations `pop()` en attente. La même logique est appliquée si c'est une opération `pop()` qui est effectuée. Le thread retire alors une opération `push(x)` de sa liste des opérations en attente. Avec cet algorithme, soit on a une seule opération dans la file d'attente, soit on a plusieurs fois une opération du même type.

On ne peut pas appliquer cet algorithme à une pile avec une linéarisabilité moyenne des promesses car les opérations ne peuvent pas être réordonnées. Ainsi, une telle pile ne pourra pas combiner une opération `push(x)` avec une opération `pop()` en attente. Cependant, une opération `pop()` peut être combinée avec l'opération `push(x)` en attente la plus récente.

L'implémentation qui respecte une linéarisabilité forte des promesses utilise une liste unique qui contient les opérations en attente. Quand un thread veut évaluer une promesse, il cherche à acquérir un verrou, puis parcourt la liste. Il élimine les

paires d'opérations `pop()` et `push()` successives et ensuite applique les opérations en attente avant de relâcher le verrou.

Avec les algorithmes qui respectent une linéarisabilité moyenne et faible des promesses, la suppression des paires `pop()` et `push()` permet de réduire la contention sur l'objet et ainsi d'améliorer les performances.

Selon le critère de cohérence utilisé, certains objets ont montré des performances jusqu'à deux fois supérieures à leurs alternatives qui n'utilisent pas de verrous.

La cohérence RedBlue

Le modèle **RedBlue** [50] est une autre façons d'adapter les propriétés de corrections en fonction des opérations. Dans ce modèle, les opérations capables de tolérer une cohérence éventuelle [75] sont marquées en bleu, tandis que celles qui nécessitent une forte cohérence sont identifiées par la couleur rouge. Les opérations bleues sont ainsi exécutées localement par chaque thread, puis répliquées de manière paresseuse, ce qui permet de rendre leur exécution particulièrement rapide. À l'inverse, les opérations rouges doivent être sérialisées et exigent que les différentes répliques du système se coordonnent afin de les appliquer dans le même ordre, ce qui, par conséquent, ralentit leur mise en œuvre.

Pour aller plus loin dans l'optimisation, le modèle propose de décomposer les opérations en deux parties distinctes : d'une part, une opération dite génératrice sans effets de bord, qui est effectuée localement par le thread, et d'autre part, une opération dite effective, générée par l'opération initiale, qui sera appliquée ultérieurement à chaque réplique. L'opération effective a pour fonction d'appliquer les effets de bords de l'opération génératrice. En dissociant ainsi la décision d'effectuer une opération de son effet de bord, le modèle permet à un plus grand nombre d'opérations de devenir commutatives.

L'exemple fourni illustrant cette notion est celui d'une banque où l'utilisateur peut effectuer trois types d'opérations : `deposit(x)`, `withdraw(x)` et `accrue()` qui permet d'accumuler des intérêts. En principe, les opérations `accrue()` et `deposit(x)` ne sont pas commutatives. Cependant, si l'on envisage que l'opération `accrue()` soit décomposée en une opération génératrice `accrue()` et une

opération effective `accrue(x)`, qui ajoute la somme des intérêts gagnés, alors dans ce contexte précis, les opérations effectives `accrue(x)` et `deposit(x)` deviennent commutatives.

Pour déterminer quelles opérations doivent être labellisées en bleu et lesquelles doivent être labellisées en rouge, trois règles fondamentales sont appliquées :

— Toute paire d'opérations effectives non commutatives est labellisée rouge.

— Toute opération effective susceptible de violer un invariant est également labellisée rouge (par exemple, `withdraw(x)` si l'on ne peut pas retirer une somme supérieure au solde restant).

— Enfin, toutes les opérations qui ne sont pas rouges sont, par défaut, labellisées bleues

En somme, le modèle RedBlue allie cohérence et performance en adaptant le traitement des opérations selon leur importance, optimisant ainsi les systèmes distribués. Le choix de l'architecture joue aussi un rôle important sur la capacité d'un programme à passer à l'échelle.

Comprendre le lien entre synchronization et performance

Comme nous l'avons mentionné précédemment, la programmation parallèle est difficile pour deux raisons. D'autre part, les opérations concurrentes doivent être correctement orchestrées pour maintenir la correction du programme. D'autre part, il est difficile d'utiliser pleinement les ressources matérielles disponibles. Sur ce second point, il est crucial d'identifier les goulots d'étranglement et interférences entre threads lié au parallélisme. Une approche efficace ici est l'exécution différentielle, qui compare les configurations associées aux exécutions rapides et lentes d'une section de code répétée en parallèle. Bouksiaa et al. [10] introduisent un outil qui utilise un score, le *SCI* (*Slowdown Caused by Interference*), pour mesurer l'impact des interférences entre threads. Ce score est calculé en observant les différences de performance entre le temps d'exécution optimal (sans interférence) et les autres exécutions, permettant de repérer les parties du code affectées par des interférences.

Un autre travail sur ce sujet est celui de David et al. [13]. Dans cet article, les auteurs examinent les mécanismes de synchronisation modernes, couvrant à la fois

les aspects matériels (e.g., protocole de cohérence de cache) et logiciels (tels que les verrous). Ce travail est réalisé sur plusieurs architectures, à la fois multicœurs et multiprocesseurs. Il identifie que la synchronisation est onéreuse entre des cœurs situés sur différents sockets est onéreux. C'est aussi le cas dans les architectures *NUMA*, du fait du coût des accès mémoire distants. Par ailleurs, l'article montre que la meilleure technique de synchronisation dépend du matériel et/ou du cas d'usage. Ce résultat est corroboré par d'autres travaux postérieurs, comme par exemple celui de Guiroux et al. [22] sur les verrous. De telles conclusions s'inscrivent dans la ligne droite de ce travail de thèse dont l'objet est la spécialisation d'un objet partagé à son cas d'usage.

Adapter le designs des systèmes de stockage

Les systèmes de stockage clé-valeur sont souvent utilisés par les applications à grande échelle pour garantir de bonnes performances. Il existe de nombreux designs adaptés à divers cas d'utilisation pour assurer une efficacité optimale du programme. Cela peut aller de systèmes avec un seul serveur où les calculs sont effectués localement, offrant une grande efficacité, à des systèmes géo-distribués garantissant une scalabilité adaptable. Certains systèmes, comme Redis [68], sont même à fil d'exécution unique, assurant simplicité et rapidité mais sans pouvoir tirer parti du parallélisme multi-cœur. Dans les systèmes avec un seul serveur, le modèle de calcul principalement utilisé est le modèle multi-cœur avec une architecture à mémoire partagée. Les systèmes géo-distribués avec plusieurs serveurs exploitent peu ou pas l'architecture multi-cœurs de leurs serveurs. Un autre design consiste à partitionner les clés dans des groupes sans chevauchement répartis sur différents cœurs ou serveurs, permettant aux unités de calcul de travailler sans avoir besoin de communiquer. Pour éviter un déséquilibre de charge de travail où seul un groupe de clés est utilisé, un système de stockage clé-valeur comme MICA [51] partitionne uniquement les écritures tout en permettant à toutes les unités de calcul de lire n'importe quelle clé. De manière générale, les systèmes de stockage clé-valeur avec un seul serveur garantissent un critère de cohérence fort comme la linéarisabilité ou la sérialisabilité, tandis que ceux avec plusieurs serveurs garantissent des critères de cohérence relâchés comme la cohérence causale. Cependant,

les systèmes multi-serveurs peuvent garantir une cohérence forte en utilisant la réplication de machine à états (State Machine Replication), où les serveurs appliquent et reçoivent les commandes dans le même ordre via des protocoles de consensus comme Paxos [48] ou Raft [61]. L'utilisation de protocoles de consensus pour maintenir un ordre total sur les opérations limite les performances d'un programme.

Un exemple ; Anna

Wu et al. [78] introduisent **Anna**, un système de stockage clé-valeur qui assure à la fois de bonnes performances et une bonne scalabilité. Pour ce faire, Anna n'utilise pas un modèle à mémoire partagée, qui implique que les différentes unités de calcul doivent se synchroniser, mais plutôt un modèle à passage de messages où chaque unité de calcul maintient un état local de l'objet qui n'est modifié que par elle-même. Dans ces modèles, les clés peuvent être assignées à une seule unité de calcul ou à plusieurs ; on appelle cela respectivement une **réplication à maître unique** ou une **réplication à maître multiple**.

Anna adopte une réplication à maîtres multiples, car bien que la réplication à maître unique garantisse la cohérence des valeurs associées à chaque clé, elle impose une contrainte sur la fréquence des modifications possibles pour chaque clé. Afin de contourner le surcoût lié à la coordination du nombre d'opérations et à l'ordre d'exécution des unités de calcul dans un modèle de passage de messages, Anna s'appuie sur des **treillis** (ou lattices), ce qui permet de réaliser des changements d'état de manière efficace. En effet, chaque réplique est représentée sous forme de treillis, de sorte que l'ordre d'application des modifications n'influence pas l'état final obtenu.

Un serveur dans Anna est structuré de la manière suivante : il est constitué d'un ensemble de threads, chacun étant affecté à un processeur unique. Ainsi, le nombre de threads est, au maximum, équivalent au nombre de processeurs disponibles. Ces threads gèrent de manière asynchrone et efficace les requêtes de stockage clé-valeur en surveillant en continu les requêtes entrantes, en les traitant, puis en enregistrant les mises à jour des paires clé-valeur dans un ensemble de changements local. Par la suite, ces mises à jour sont propagées aux maîtres des clés concernées, et l'ensemble

de changements local est réinitialisé.

Grâce à l'utilisation de treillis, lorsqu'une clé avec plusieurs maîtres est modifiée à plusieurs reprises, plutôt que d'envoyer chaque modification individuellement, ce qui risquerait de saturer le réseau, le thread envoie uniquement le résultat de la fusion de ces mises à jour. De plus, en offrant à l'utilisateur la possibilité de définir de manière flexible la façon dont ces fusions sont réalisées, Anna permet de garantir différents critères de cohérence selon les exigences du système.

Tester les objets partagés avec Lincheck

Après avoir sélectionné le critère de cohérence optimal pour améliorer les performances de l'application, il est essentiel de mettre en œuvre un objet partagé qui respecte ce critère. Koval et al. [44] ont développé **Lincheck**, un outil simple qui invite les développeurs à spécifier les opérations que leurs objets effectuent, puis génère une série de tests concurrents. Si une erreur se produit, l'outil renvoie le scénario précis menant au résultat incorrect. Cela facilite considérablement le processus de débogage en fournissant une trace d'exécution détaillée aboutissant à l'erreur.

L'un des principaux avantages de Lincheck réside dans le fait qu'il ne demande aux utilisateurs que de définir ce qu'ils souhaitent tester, sans exiger qu'ils précisent comment ces tests doivent être exécutés. Cela simplifie grandement la tâche pour les développeurs qui cherchent à vérifier si un objet partagé comporte des bogues. L'outil génère automatiquement plusieurs scénarios, les répète plusieurs fois et vérifie si les résultats obtenus sont conformes à une exécution séquentielle, garantissant ainsi que l'objet est linéarisable.

Pour des raisons de performance, Lincheck ne génère pas toutes les exécutions séquentielles possibles. À la place, il construit un **système de transitions labellisées** (LTS), un graphe orienté où les nœuds représentent les différents états de l'objet, et les arêtes, les opérations qui permettent de passer d'un état à un autre, ainsi que le résultat de chaque opération. Le LTS est généré en manipulant l'objet de façon séquentielle, et un scénario est considéré comme valide s'il correspond à un chemin fini dans le LTS.

Pour utiliser Lincheck, le développeur doit spécifier l'instance de l'objet à tester

et définir les opérations à évaluer. Lincheck propose également plusieurs options, comme le nombre de threads, le nombre d'opérations par thread, le nombre de scénarios et le nombre d'itérations par scénario. Les options permettent de tester différents modèles de concurrence, y compris des modèles avec un seul producteur et plusieurs consommateurs.

Il existe deux principales méthodes pour tester un objet partagé. La première, **StressOption()**, consiste à exécuter l'objet avec plusieurs threads sous une charge intense pendant une période donnée. Cette méthode déclenche de nombreuses opérations simultanées pour simuler des conditions de concurrence réalistes. Bien qu'elle ne garantisse pas la détection de toutes les anomalies possibles, elle est souvent efficace pour mettre en évidence des problèmes apparaissant sous des charges concurrentes élevées. La seconde méthode, **ModelCheckingOption()**, génère un nombre limité d'ordonnancements pour les opérations spécifiées et introduit des points de changement de contexte. Afin de maximiser la couverture des tests, Lincheck priorise les ordonnancements non encore explorés à chaque nouvelle exécution. En limitant le nombre d'ordonnancements, Lincheck parvient à maintenir des temps de test raisonnables, quelle que soit la complexité du scénario ou de l'algorithme. En définitive, plus la configuration des tests est étendue, c'est-à-dire plus le nombre de scénarios évalués est élevé, plus l'outil a de chances de révéler des bogues potentiels.

Tester les objets partagés en amont permet de vérifier leur correction, mais certains problèmes persistent, qui ne peuvent pas être résolus uniquement par cette approche. Par exemple, les résultats des tests dépendent fortement de la machine sur laquelle ils sont exécutés. Cela signifie qu'il est nécessaire de relancer une nouvelle série de tests dès que la charge de travail de l'application évolue ou lorsqu'une nouvelle machine est utilisée. Par ailleurs, le besoin d'implémenter chaque objet à tester et de vérifier chaque charge de travail pour identifier d'éventuels goulots d'étranglement représente un investissement de temps important.

La *scalable commutativity rule*

Clements et al. [11] introduisent et formalisent une règle visant à orienter le choix des interfaces de programme afin de garantir leur scalabilité. Cette règle,

connue sous le nom de *scalable commutativity rule*, stipule que lorsque plusieurs opérations sont commutatives, il est possible de concevoir une implémentation sans conflits lors de leur exécution.

Pour déterminer si une implémentation est scalable, il est essentiel de vérifier qu'elle ne génère pas de conflits. À cette fin, les auteurs modélisent les opérations de la manière suivante : chaque opération représente un pas de calcul qui fait évoluer le système d'un état à un autre et produit une réponse. Deux pas de calcul entrent en conflit s'ils accèdent au même composant dans un même état, et que l'un de ces pas est une écriture. Une implémentation, composée de plusieurs pas de calcul, produit ainsi une *histoire*. L'implémentation est considérée correcte, et l'histoire valide, si les réponses générées respectent toujours une spécification donnée.

Pour formaliser la *scalable commutativity rule*, ils définissent une nouvelle forme de commutativité qui prend en compte à la fois l'état du système et les arguments des opérations concurrentes, appelée *SIM*-commutativité. Un ensemble d'opérations SIM-commute si chaque paire d'opérations au sein de cet ensemble commute entre elle, et si les opérations exécutées après cet ensemble produisent les mêmes effets de bords tout en conservant les mêmes valeurs de retour. Un ensemble d'opérations SIM-commute si toutes les opérations de cet ensemble commutent deux à deux et que les opérations effectuées après cette ensemble effectuent les mêmes effets de bords et garde les mêmes valeurs de retour. Plus précisément, dans une histoire H, si l'on désigne par Y cet ensemble d'opérations, par X l'ensemble des opérations précédant Y, et par Z l'ensemble des opérations suivant Y, on dit que Y SIM-commute dans H si, pour tout réordonnancement Y' de Y, la séquence $X||Y||Z$ est légale si et seulement si $X||Y'||Z$ l'est également. Ainsi, si une partie de l'histoire produite *SIM*-commute, alors il est possible de créer une implémentation qui est capable de passer à l'échelle pour ces pas de calcul.

Cependant, les interfaces pouvant être complexes, il est souvent difficile de déterminer quand les opérations SIM-commutent. Pour remédier à ce problème, les auteurs ont développé un outil appelé *COMMUTER*. Cet outil se divise en trois parties : *Analyzer*, qui analyse l'interface d'une application afin d'estimer quelles opérations commutent et sous quelles conditions ; *TestGen*, qui génère des tests pour ces opérations supposées commutatives ; et enfin *MTrace*, qui vérifie si l'implémentation est effectivement sans conflits pour les tests générés.

Résumé

Dans cette section, nous examinons divers mécanismes de synchronisation modernes et leurs outils associés, en soulignant les avantages et inconvénients. En particulier, nous présentons la mémoire transactionnelle, qui permet d'exécuter des blocs d'opérations de manière atomique. Nous étudions le compromis entre cohérences forte et faible, via des approches telles que la linéarisabilité des valeurs intermédiaires, la quasi-linéarisabilité ou encore le modèle Red-Blue. Nous avons vu des cas d'usage de ce compromis, comme par exemple le système de stockage clé-valeur Anna. Enfin, nous avons fait une analyse de deux techniques récentes. D'une part, Lincheck qui permet aux développeurs de tester une implémentation sans en avoir à écrire les tests. D'autre part la règle de SIM-commutativité a été énoncée. Cette règle stipule que si les opérations d'une interface commutent, il est possible d'implémenter cette interface de manière scalable.

2.6 Conclusion du chapitre

Contexte

Les infrastructures informatiques modernes utilisent en nombre les systèmes multiprocesseurs et multicœurs. La conception d'un programme parallèle capable de passer à l'échelle sur ces architectures constitue un défi de taille pour les développeurs. Comme nous l'avons vu dans ce chapitre, il existe de nombreuses façons d'aborder l'écriture et la structuration d'un programme parallèle. Par exemple, anticiper le nombre de données fréquemment accédées dans une application permet d'adapter l'architecture sur laquelle ce programme sera exécuté, optimisant ainsi l'utilisation de la hiérarchie mémoire. De même, partitionner les données entre les différentes unités de calcul aide à éviter les conflits d'accès en mémoire. En effet, comme souligné en Section 2.2.2, quand les accès mémoire sont disjoints alors le nombre d'opérations par seconde augmente conjointement avec le nombre de threads effectuant des opérations.

Comme nous l'avons vu, plusieurs stratégies existent pour introduire du paral-

lélisme dans un programme : exécuter des tâches de fond de manière asynchrone grâce aux *futures*, créer des threads virtuels éphémères pour chaque tâche ou encore utiliser des verrous afin d'isoler les sections critiques du programme. Après avoir fait ces choix, le développeur doit encore choisir les propriétés de correction indispensables au bon fonctionnement d'un programme parallèle, ainsi que les conditions de progrès qui garantissent son avancement.

Notre étude a mis en lumière l'intérêt que présentent des critères de cohérence faible, tels que la linéarisabilité des valeurs intermédiaires ou encore la quasi-linéarisabilité. Néanmoins, bien que ces critères de cohérence puissent améliorer les performances d'un programme en offrant une garantie de correction moins stricte que celle de la linéarisabilité, ils exigent du développeur un certain niveau d'expertise pour comprendre la multiplicité des entrelacements [56, 57]. C'est pourquoi nos recherches se sont focalisées sur les objets garantissant une forte cohérence, tels que la linéarisabilité et la cohérence séquentielle.

Problématique

Face à la difficulté de programmer les architectures parallèles modernes, il est essentiel d'offrir un support language adapté aux développeurs d'applications. Des bibliothèques d'objets partagés sont disponibles à cet effet dans de nombreux langages (telle que Boost en C++ [70]). Elles sont mis à disposition pour simplifier la création et la gestion de la concurrence. Toutefois, les objets de ces bibliothèques sont conçus pour un usage général. Cela a pour effet qu'ils ne sont pas toujours optimisés pour des cas d'utilisation spécifiques. Certains développeurs experts ont donc parfois recours à des objets ad-hoc afin de rendre leurs programmes plus scalables.

Des outils comme *Lincheck* et *COMMUTER* aident ces développeurs à tester leurs objets partagés ou à s'assurer que le programme peut passer à l'échelle, mais comme nous l'avons vu, ces outils ont certaines limites. Premièrement, *Lincheck* exige que l'objet soit déjà implémenté pour pouvoir être testé, ce qui représente un effort de développement sans garantie que l'objet soit réellement adapté à son utilisation. De plus, bien que *Lincheck* ne produise pas de faux positifs, il ne permet pas de garantir qu'un objet ne générera jamais d'erreurs avec les options

fournies à l'outil. Deuxièmement, *COMMUTER* permet de vérifier si les opérations d'une interface sont commutatives, ce qui indique la possibilité d'implémenter cette interface de manière à améliorer la scalabilité. Cependant, cette approche ne tient pas compte de la fréquence d'utilisation des différentes opérations de l'interface. Il se peut qu'un petit sous-ensemble d'opérations soit commutatifs, mais si ces opérations sont fréquemment appelées, l'implémentation de cette interface peut alors devenir pertinente. Contrairement à l'outil *COMMUTER*, notre approche propose une méthode permettant de se focaliser sur les opérations les plus utilisées de la spécification séquentielle. Par ailleurs, nous considérons aussi l'amélioration de toutes les opérations à l'interface, même si certaines ne sont pas forcément commutatives.

Suite de la thèse

Dans ce manuscrit, nous présentons, définissons et évaluons *le principe d'ajustement* : ajuster un objet partagé permet de le rendre efficace pour un cas d'usage spécifique. Pour ce faire, nous sondons d'abord de nombreuses bases de code afin de mieux comprendre les usages et pratiques des objets partagés. Fort de cette analyse, nous introduisons un outil comparatif, dit graphe d'indistinguabilité, qui permet d'évaluer l'efficacité relative de deux objets partagés. Á l'aide de cet outil, nous formalisons le principe d'ajustement. Ce principe est ensuite utilisé à la construction de la bibliothèques d'objets ajustés `DEGO`. Nous montrons ensuite l'intérêt de cette bibliothèque à l'aide de microbenchmarks et en utilisant une application complexe de type réseau social.

Chapitre 3

Principe d'ajustement

Comme discuté précédemment, les systèmes multicœurs et multiproesseurs sont largement répandus aujourd'hui dans les infrastructures informatiques. Afin de bénéficier au mieux des performances offertes par ces architectures, il est essentiel de paralléliser les programmes. Les objets partagés ont un rôle pivot dans ce contexte. Ils permettent d'orchestrer et de coordonner efficacement les différentes entités de calcul (processus, threads) afin de permettre l'accès aux ressources partagées tout en assurant l'intégrité des données.

Au cours de ce chapitre, nous allons introduire le principe d'ajustement. L'idée centrale est de designer un objet partagé spécifiquement pour son usage dans un programme en vue d'améliorer les performances.

Afin d'exposer cette idée, nous allons d'abord faire un état des lieux de l'usage des objets partagés dans les programmes (Section 3.1). Plus précisément, nous allons faire une analyse quantitative de plusieurs bases de code Java afin de connaître comment les développeurs utilisent les objets partagés. À travers cette analyse, nous verrons que *(i)* les objets partagés sont employés rarement mais dans des classes clés des programmes, *(ii)* une faible partie de leur interfaces est utilisée dans chaque cas applicatif, et *(iii)* certains développeurs experts ont déjà recours à des objets ad-hoc pour des situations spécifiques dans lesquelles ils cherchent à obtenir de meilleures performances.

Fort de cette analyse, nous allons ensuite introduire la notion d'objet ajusté. Pour ce faire, nous allons d'abord présenter un outil théorique appelé graphe d'in-

distinguabilité (Section 3.2). Ce graphe permet de caractériser la capacité d'un objet partagé à différencier différents entrelacements d'opérations. Nous allons montrer en quoi la densité de ce graphe et sa connexité sont liés à la possibilité d'implémenter l'objet de manière scalable.

Ensuite, nous définirons le principe d'ajustement (Section 3.3). En un mot, ajuster un objet revient à restreindre son interface (exprimer sous la forme d'un sous-typage) et/ou à rendre asymétrique son utilisation par les threads. Nous verrons comment un ajustement se traduit par une augmentation du nombre d'arcs dans les graphes d'indistinguabilité de l'objet, et donc potentiellement en de meilleures performances. Une fois ce principe défini, nous allons illustrer différents types d'ajustements possibles. Ces exemples motiveront la bibliothèque d'objets ajustés (`DEGO`) qui sera décrite et évaluée dans le prochain chapitre.

3.1 Usage des objets partagés

L'étude qui suit couvre les usages des objets partagés dans des bases de codes représentatives pour le language Java. Cette étude est scindée en deux parties. Tout d'abord nous allons faire une analyse statique de l'utilisation par les développeurs de la fondation Apache de la bibliothèque d'objets `java.util.concurrent` (Section 3.1.1). Ensuite, nous allons nous pencher sur l'existence d'objets ad-hoc spécifiquement dédiés à l'obtention de performances supplémentiares pour des situations applicatives bien définies (Section 3.1.2).

3.1.1 Analyse de bases de code

Comme discuté précédemment, les systèmes multicœurs et multiproesseurs sont aujourd'hui omniprésents dans l'informatique. Afin de maximiser les performances offertes par ces architectures, il est essentiel de paralléliser les tâches et d'optimiser l'accès concurrent aux données partagées. En conséquence, l'usage des objets partagés a considérablement augmenté dans les programmes modernes. Afin d'illustrer cette tendance, nous avons réalisé une analyse statique approfondie sur 50 projets d'envergure issus de la fondation Apache, tous hébergés sur GitHub. L'analyse porte spécifiquement sur les objets partagés suivants : `AtomicLong`,

`ConcurrentLinkedQueue`, `ConcurrentSkipListSet` et `ConcurrentHashMap`.

Un usage en augmentation

Dans un premier temps, nous avons recensé le nombre de déclarations de ces objets partagés ainsi que leurs proportions au sein de ces projets. En Figure 3.1, nous reportons les résultats de cette analyse pour `ConcurrentHashMap`. Les résultats sont similaires pour les trois autres structures de données.

Cette figure montre les résultats de l'analyse au cours des dix dernières années. L'axe des ordonnées s'étale de 2015 à 2024. En abscisse, nous avons indiqué (*à gauche*) le nombre moyen de déclarations de `ConcurrentHashMap`, et (*à droite*) le pourcentage que ces déclarations représentent sur la base de dans son ensemble.

De manière générale, on observe une augmentation progressive du nombre de déclarations de `ConcurrentHashMap`. Ceci témoigne d'une adoption croissante de cette structure de données pour gérer le parallélisme. Ainsi, de 2015 à 2024, le nombre de déclarations a bondi, passant de 46 déclarations en moyenne à 116 par projet.

Toutefois, il est pertinent de noter que, bien que le nombre d'objets partagés augmente, cette proportion reste relativement faible dans l'ensemble des projets analysés. Dans la figure Figure 3.1, les déclarations de `ConcurrentHashMap` représente (en moyenne) moins de 1 % de l'ensemble des déclarations sur le projet. Ainsi, si on considère ces chiffres au regard du nombre de lignes de code, il s'agit d'une croissance de 25% en dix ans.

Ayant un rôle central

Pour évaluer si ces objets sont centraux dans les programmes malgré leur faible représentation, nous avons examiné les 20 fichiers les plus modifiés au cours des dix dernières années. Puis, nous avons déterminé si ces fichiers utilisent ou non des objets de la bibliothèque *java.util.concurrent*. Les résultats de cette fouille sont présentés dans la Figure 3.2.

Sur l'axe des abscisses, on retrouve les 50 projets de la fondation Apache, chaque colonne de la matrice représentant les 20 fichiers les plus modifiés de chaque projet. En bleu, les fichiers utilisant des objets de la bibliothèque *java.util.concurrent*,

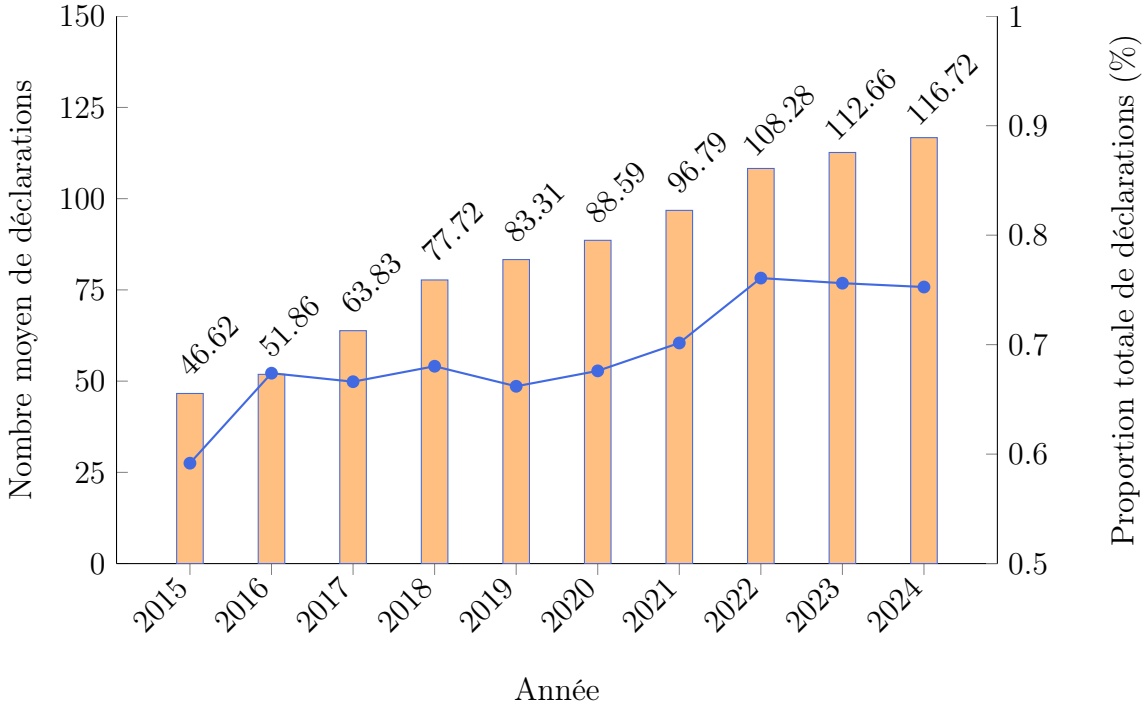

FIGURE 3.1 – Nombre moyen et proportion des déclarations de `ConcurrentHashMap` dans 50 projets de la fondation Apache de 2015 à 2024 (données extraites de GitHub).

et en orange, ceux qui n'en contiennent pas. L'intensité de la couleur reflète la fréquence de modification d'un fichier : plus la couleur est sombre, plus le fichier a été modifié (nombre de commits).

Comme l'indique la figure, les fichiers utilisant des objets de *java.util.concurrent* constituent presque la moitié des fichiers les plus modifiés. Ainsi, bien que le nombre d'objets partagés déclarés reste faible, ils apparaissent fréquemment dans les fichiers centraux au développement de programmes parallèles. C'est aussi pour cette raison que de nombreuses bibliothèques d'objets partagés existent, malgré leur faible représentation dans le code.

Avec des opérations privilégiées

Nous mentionnons à la fin du Chapitre 2 que les bibliothèques d'objets partagés sont conçues pour faciliter la programmation parallèle pour un large éventail de

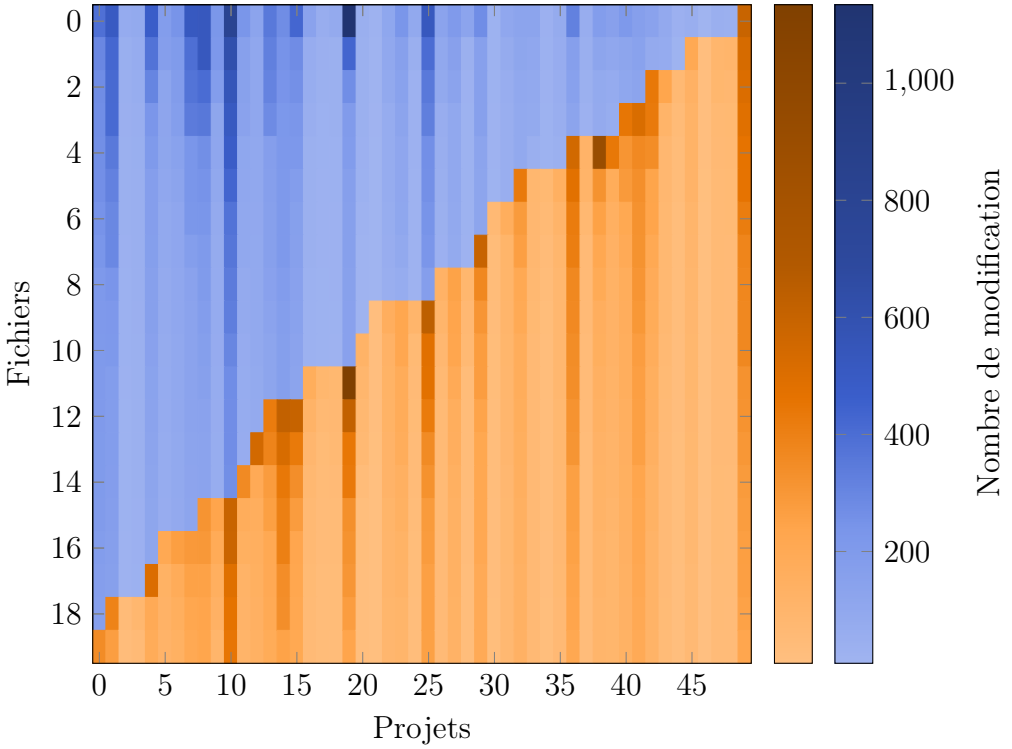

FIGURE 3.2 – Analyse des 20 fichiers les plus modifiés dans 50 projets de la fondation Apache entre 2015 et 2024

développeurs. En conséquence, les objets partagés proposent une interface étendue avec des fonctionnalités complexes. Cela s'explique par le fait que l'interface doit couvrir une grande diversité d'utilisations possibles, adaptées à une vaste gamme d'applications. Cependant, dans la majorité des cas, nous avons observer qu'une application n'exploite qu'une partie très restreinte de l'interface.

Pour démontrer empiriquement ce fait, nous avons analysé la répartition des méthodes utilisées dans les différents projets, les classes dans lesquelles ces méthodes sont appelées, ainsi que l'utilisation ou non de la valeur de retour de ces appels. La Figure 3.3 illustre les usages observés pour les types de données susmentionnés. Pour des raisons de clarté, seules les méthodes représentant plus de 10% des appels totaux sont affichées dans cette figure. Celles qui représentent moins de 5% des appels ont été regroupées dans le segment intitulé *autres*.

Nous constatons, de manière générale, que seul un sous-ensemble de l'interface d'un objet est effectivement utilisé. Par ailleurs, certaines méthodes sont appe-

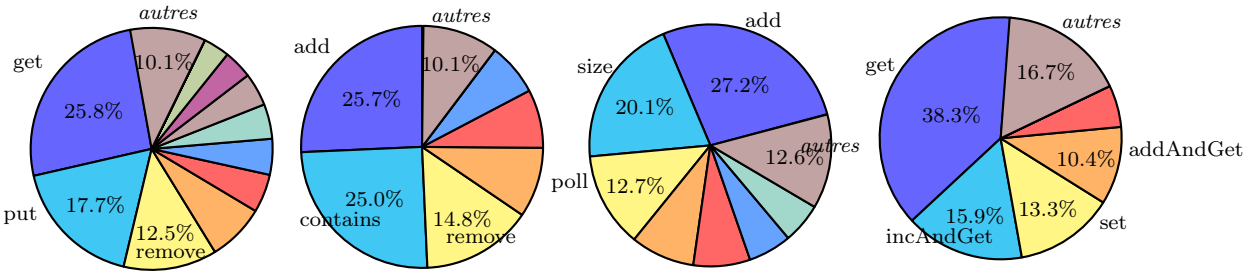

FIGURE 3.3 – Utilisation de quatre objets partagés dans 50 projets Java open source. De gauche à droite, les classes `ConcurrentHashMap`, `ConcurrentSkipListSet`, `ConcurrentLinkedQueue` et `AtomicLong`.

lées de manière significativement plus fréquente que d'autres. Ainsi, parmi les 34 méthodes disponibles dans `AtomicLong`, seulement trois d'entre elles représentent 67.5% des appels. De même, sur les 60 méthodes de `ConcurrentHashMap`, trois comptent pour 55.99% des utilisations. Concernant `ConcurrentLinkedQueue` et `ConcurrentSkipListSet`, ces proportions atteignent respectivement 59.99% et 65.62%.

Dans la Figure 3.4, l'axe des ordonnées répertorie les méthodes utilisées à travers différentes classes. Pour chaque méthode, la figure précise si la valeur de retour est utilisée ($+$) ou non ($\times$). Cette figure illustre l'utilisation de `AtomicLong` dans les classes du projet *Apache Cassandra*. Des observations similaires peuvent être faites pour les autres types de données dans d'autres programmes.

Il est intéressant de noter que certaines méthodes, comme `incrementAndGet` et `get`, ont tendance à être invoquées dans les mêmes classes. Les deux méthodes les plus fréquemment utilisées sont appelées conjointement dans 29.5% des classes pour `AtomicLong`, dans 40% des cas pour `ConcurrentSkipListSet` et dans 45.2% pour `ConcurrentHashMap`. De plus, il est frappant de constater que dans de nombreux cas, des méthodes comme `incrementAndGet` et `addAndGet` ne font pas usage de leur valeur de retour.

En conclusion, cette analyse révèle qu'il existe des opportunités d'utiliser des objets spécifiques pour les besoins spécifiques d'un programme. Dans la section ci-après, nous allons faire un recensement de certains de ces objets ad-hoc construits par des développeurs experts.

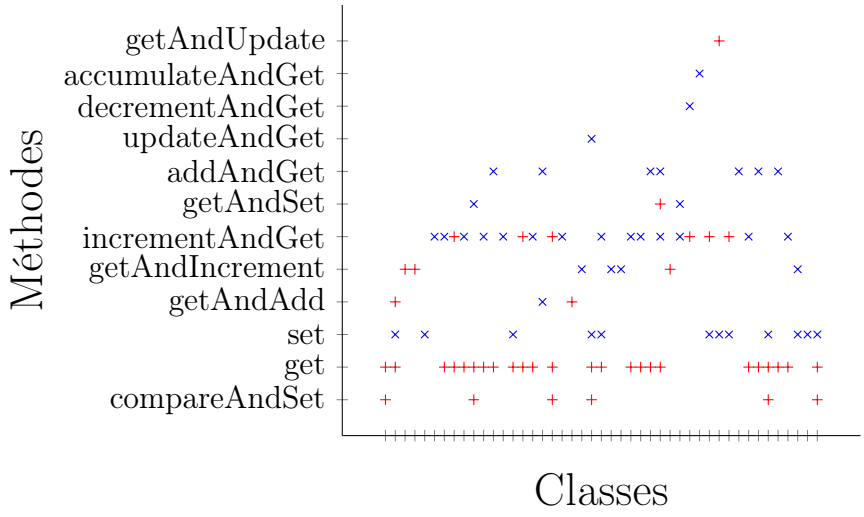

FIGURE 3.4 – Utilisation des méthodes d'`AtomicLong` dans les différentes classes d'*Apache Cassandra*. Le symbole (+) indique que la valeur de retour de la méthode est utilisée, tandis que le symbole (×) signale qu'elle ne l'est pas.

3.1.2 Objets ad-hoc

Suite à cette analyse, une question intéressante survient : *si l'intégralité de l'interface n'est pas utilisé, est-il possible de proposer une version optimisée de l'objet, adaptée aux besoins spécifiques de l'application ?* Nous pouvons répondre par l'affirmative à cette question. En effet, plusieurs développeurs expérimentés ont déjà adopté une telle approche. C'est à dire qu'ils ont conçu des objets ad-hoc spécialisés pour certains cas d'usage applicatifs, le but étant de maximiser les performances. Nous allons voir quelqu'uns de ces objets à travers de cette section.

Une référence *write-once*

Le *Listing* 3.1 illustre un exemple concret tiré du projet *Concurrentli* de Linkedin [52], où les développeurs ont conçu une version spécialisée de la classe `AtomicReference`, baptisée `AtomicWriteOnceReference`. Cette implémentation est optimisée pour des scénarios où la référence n'est écrite qu'une seule fois, c'est-à-dire qu'une fois la valeur initiale est définie, elle ne peut plus être modifiée. Dans cette version spécifique, la stratégie d'optimisation repose sur l'utilisation du cache du processeur pour accroître les performances.

```
1  public class AtomicWriteOnceReference<T> {
2  private T _cachedObj = null;
3  private volatile T _obj = null;
4
5  public T get() {
6    if (_cachedObj != null) return _cachedObj;
7    _cachedObj = _obj;
8    return _cachedObj;
9  }
10
11 public boolean set(T value) {
12   if (!trySet(value))
13     throw new IllegalStateException("Value has already been
     set");
14   return true;
15 }
16
17 public boolean trySet(T value) {
18   if (get() != null) return false;
19   if (!UPDATER.compareAndSet(this, null, value)) return false;
20   _cachedObj = value;
21   return true;
22 }
23
```

Listing 3.1 – Extrait de la librairie Concurrentli

De manière plus détaillée, la méthode `get()` regarde le contenu de la variable `_cachedObj` (lignes 6-7). Si la variable est initialisée, son contenu est retournée. Dans le cas contraire, le thread accède à la variable `_obj` qui contient la référence atomique à l'objet. Pour garantir l'atomicité cette variable est volatile, assurant donc que toute mise à jour est visible immédiatement. Toutefois cela à un coût et requiert des barrières en lecture et en écriture, comme nous l'avons vu en Section 2.3.4. A contrario, la variable `_cachedObj` n'a pas de propriété spécifique ce qui permet d'y acccéder efficacement.

Notons que la classe `AtomicWriteOnceReference` s'assure que la référence est initialisée au plus une fois. Pour ce faire, la méthode `set()` effectue une comparaison atomique via l'utilisation du champ `UPDATER` (ligne 19), de type `AtomicReferenceFieldUpdater` fourni par le JDK. Cet appel garantit l'unicité

```
1  public void add(long x) {
2    Cell[] cs; long b, v; int m; Cell c;
3    if ((cs = cells) != null || !casBase(b = base, b + x)) {
4      int index = getProbe();
5      boolean uncontended = true;
6      if (cs == null || (m = cs.length - 1) < 0 ||
7        (c = cs[index & m]) == null ||
8        !(uncontended = c.cas(v = c.value, v + x)))
9        longAccumulate(x, null, uncontended, index);
10   }
11 }
12
```

Listing 3.2 – Implémentation de la méthode `add(x)` du `LongAdder` de la bibliothèque *java.util.concurrent*.

de l'écriture en utilisant la primitive de synchronization `compareAndSet`. En cas de tentative d'écriture après son initialisation, une exception est levée.

Un compteur parallèle

Lorsque les opérations de modification sur un objet sont commutatives, il paraît intéressant de paralléliser les accès à cet objet en permettant à différents threads de manipuler des parties distinctes de l'objet. Cela réduit la contention entre threads. Dans la bibliothèque `java.util.concurrent.atomic`, la classe abstraite `Striped64` illustre ce principe en utilisant une structure dynamique, le tableau `cells`, constitué de plusieurs instances de `Cell`. Ces cellules agissent comme des "compartiments" qui répartissent la charge, allégeant ainsi la pression concurrentielle initialement sur un seul point.

Le Listing 3.2 présente l'utilisation de cette classe abstraite par `LongAdder`, un compteur pouvant être incrémenté, décrémenté, consulté ou réinitialisé de manière efficace. L'implémentation de la méthode `add` est décrite dans ce listing afin d'analyser comment la contention est réduite. Lorsque plusieurs threads doivent modifier la valeur du compteur, celui-ci essaye d'abord de limiter la contention via plusieurs mécanismes. Initialement, un thread tente d'ajouter une valeur x en vérifiant si le tableau `cells` est nul ou en essayant de modifier la base du compteur de façon atomique à l'aide d'une opération `CompareAndSwap` (abbrégé en `CAS`). Si

la modification est réussie, la méthode se termine.

Cependant, si le thread rencontre de la contention (parce que le `CAS` échoue) ou voit que le tableau de cellules est initialisé. Le thread calcule ensuite un index pour sélectionner une cellule spécifique dans ce tableau `cells` (ligne 4). Cet index est utilisé pour assigner un emplacement où la modification peut être effectuée de manière plus localisée, réduisant ainsi le conflit avec les autres threads.

Avant d'accéder à cette cellule, le thread vérifie plusieurs conditions : il s'assure que le tableau `cells` est bien initialisé et qu'il possède une taille valide (ligne 6). Si ce n'est pas le cas, la méthode `longAccumulate` est appelée pour soit créer le tableau, soit ajuster sa taille pour accueillir plus de cellules et ainsi mieux distribuer les mises à jour.

Si le tableau est valide, le thread tente d'accéder à la cellule qui lui est assignée (ligne 7). Si celle-ci est déjà occupée ou n'existe pas encore, la méthode `longAccumulate` est encore invoquée pour ajuster ou initialiser cette cellule. Enfin, le thread tente d'appliquer sa mise à jour via un `CAS` (ligne 8). En cas d'échec de cette opération (par exemple si plusieurs threads accèdent simultanément à la même cellule), la taille du tableau `cells` sera augmentée, permettant ainsi une meilleure distribution des threads dans des cellules distinctes et réduisant la probabilité de collisions futures.

Ainsi, cette approche permet de réduire la contention, tout en offrant une capacité d'adaptation dynamique à une charge concurrentielle croissante. Dans le Chapitre 4 nous testons et comparons les performances du `LongAdder` avec un compteur concurrent classique de la librairie *java.util.concurrent* ainsi qu'avec une de nos propre implémentation d'un compteur partagé.

Une file optimisée

Il est possible d'optimiser un objet partagé en fonction de la partie de son interface utilisée. Dans ce cas, certaines opérations vont être rendues légèrement moins efficaces afin de maximiser les performances des opérations utilisées fréquemment. Dans le Listing 3.3, tiré d'*Apache Ignite*, l'implémentation d'une `FastSizeDeque` est utilisée pour améliorer les performances de la méthode `size()`, qui retourne le nombre d'éléments présents dans la file.

Dans une implémentation classique de `Deque`, le calcul de la taille implique de parcourir toute la structure, depuis la tête de la file jusqu'à la queue, ce qui donne à la méthode `size()` une complexité en $O(n)$, où n est le nombre d'éléments. Cette approche devient inefficace pour des files volumineuses, surtout si la méthode `size()` est appelée fréquemment. Á la place, la classe `FastSizeDeque` optimise ce procédé en utilisant un compteur basé sur le `LongAdder`. Ce compteur est incrémenté à chaque ajout d'un élément et décrémenté à chaque retrait. Ainsi, la méthode `size()` atteint une complexité constante de $O(1)$, ce qui la rend extrêmement rapide, même lorsque la taille de la file devient très grande. Cette optimisation présente un compromis : les opérations d'ajout et de retrait peuvent être légèrement moins performantes en raison de la gestion du compteur. Toutefois, dans les contextes où la méthode `size()` est fréquemment appelée, cette approche améliore significativement les performances globales.

Dans les systèmes distribués comme *Apache Ignite*, il est fréquent que des mises à jour soient envoyées de manière asynchrone d'un nœud à un autre. Dans ce contexte la `FastSizeDeque` peut être utilisée pour jouer le rôle de tampon (buffer). Ce tampon accumule les mises à jour et les envoie lorsqu'il atteint une taille limite prédéfinie, optimisant ainsi les échanges réseau en réduisant la fréquence des envois. Dans ce type de mécanisme, la méthode `size()` est sollicitée de manière répétée afin de vérifier la capacité du tampon avant chaque envoi. Il devient donc crucial d'optimiser les performances de cette méthode, car son efficacité influe directement sur le comportement global du système. Une méthode `size()` trop lente pourrait devenir un goulot d'étranglement, impactant la rapidité de traitement des mises à jour et l'efficacité des communications entre les nœuds. Dans cette utilisation de la `FastSizeDeque`, seul un sous-ensemble de la spécification séquentielle est utilisé à un instant donné (à savoir, `offer` et `size`).

En somme, l'utilisation d'une structure de données comme `FastSizeDeque`, permet de concilier gestion efficace des tampons et optimisation des ressources dans des environnements concurrents comme ceux rencontrés dans *Apache Ignite*. Ainsi, la `FastSizeDeque` illustre bien l'idée selon laquelle une optimisation ciblée peut transformer des opérations coûteuses en opérations plus efficaces, selon les besoins spécifiques de l'application.

```
1  private final Deque<E> deque;
2
3  private final LongAdder adder = new LongAdder();
4
5  ...
6
7  @Override public boolean offer(E e) {
8    boolean res = deque.offer(e);
9
10   if (res)
11     adder.increment();
12
13   return res;
14 }
15
16 @Override public E poll() {
17   E res = deque.poll();
18
19   if (res != null)
20     adder.decrement();
21
22   return res;
23 }
24
25 public int sizex() {
26   return adder.intValue();
27 }
28 ...
29
```

Listing 3.3 – Extrait de l'implémentation de la `FastSizeDeque` du projet *Apache Ignite*.

Autre exemple de file

Il est possible d'adapter l'implémentation d'un objet partagé lorsqu'on constate une utilisation asymétrique de sa spécification séquentielle, c'est-à-dire lorsque les threads n'utilisent pas tous le même ensemble d'opérations. Dans ce contexte, la `MpscLinkedQueue` du projet *JCTools* constitue un exemple pertinent, offrant une file chaînée avec plusieurs producteurs et un seul consommateur, ce qui n'est pas nativement pris en charge efficacement dans le *JDK*.

La classe `MpscLinkedQueue` optimise l'opération `remove(Object o)` en profitant de l'absence de retraits concurrents, permettant ainsi au consommateur unique de gérer les suppressions internes de manière simplifiée. En détail, la méthode `offer` de cette structure peut être appelée simultanément par plusieurs threads, chacun tentant d'ajouter un nœud à la file. Afin d'ordonner ces ajouts, les threads essayent de mettre à jour le premier nœavec un appel à la méthode (`AtomicReference.compareAndExchange`), ce qui garantit qu'un seul thread à la fois peut ajouter un nœud à la liste.

Une fois insérés, les éléments sont supprimées par le thread unique, lequel enlève les nœuds sans synchronisation explicite s'il sont avant le dernier nœuds. Lorsqu'il s'agit de retirer le dernier nœud, le consommateur tente de le remplacer par le précédent via un appel à `compareAndExchange`. En cas d'échec de cette opération, le consommateur attend alors que le thread qui ajoute un nœud termine son insertion en vérifiant que le nœud suivant du nœud producteur devient non nul. Cette construction n'est donc pas wait-free. Si un producteur est interrompu juste avant d'attacher son nouveau nœud à la file, il crée une "bulle" qui bloque la progression du consommateur, lequel doit attendre la fin de l'insertion pour poursuivre le parcours de la file.

3.1.3 Conclusion de l'étude

Cette étude approfondie des objets partagés dans 50 projets open source de la fondation Apache révèle que, bien que leur utilisation reste relativement limitée en nombre, ces objets jouent un rôle crucial dans le développement parallèle moderne. Le recours croissant aux objets partagés s'explique par la nécessité d'optimiser la gestion des ressources partagées dans des environnements multicœurs et multiproesseurs, omniprésents aujourd'hui.

L'analyse statistique montre une augmentation progressive de l'utilisation des objets partagés, notamment la `ConcurrentHashMap`, au cours des dix dernières années, reflétant une adoption croissante des structures concurrentes pour résoudre des problèmes de synchronisation et de performance. Néanmoins, malgré leur importance dans la gestion efficace du parallélisme, la proportion de ces objets reste faible dans les projets globaux, représentant moins de 1 % des déclarations totales.

Cependant, l'étude des 20 fichiers les plus modifiés dans chaque projet révèle que les objets de *java.util.concurrent* apparaissent souvent dans des fichiers centraux et fréquemment modifiés. Ce constat souligne que ces objets, bien que peu nombreux, sont concentrés dans des fichiers essentiels au développement de programmes parallèles et, par conséquent, jouent un rôle de pivot dans ces programmes. En effet, les fichiers utilisant ces objets sont souvent plus modifiés, ce qui témoigne de leur importance dans les ajustements et les améliorations continues des projets.

L'étude des méthodes spécifiques des objets partagés a permis d'observer une utilisation asymétrique de leurs interfaces : seules quelques méthodes sont fréquemment appelées, tandis qu'une large part de leurs fonctionnalités reste sous-exploitée. Cette observation ouvre la voie à des optimisations ciblées pour adapter les objets aux besoins spécifiques de chaque application. Des objets ad-hoc, comme l'`AtomicWriteOnceReference` dans le projet *LinkedIn* ou la `FastSizeDeque` dans *Apache Ignite*, illustrent cette tendance à spécialiser les méthodes pour optimiser les performances. Ces objets spécialisés permettent une réduction de la contention et une meilleure gestion des ressources.

Dans les sections qui viennent, nous allons proposer un canevas formel pour comprendre ces objets spécialisés, dits *ajustés* par la suite. Ce canevas comprend la notion de graphe d'indistinguabilité qui capture la capacité de l'objet à discerner des entrelacements d'opérations distincts. Nous allons définir la notion d'ajustement, qui sera interprété comme une augmentation de la densité, ou de la connexité du graphe.

Résumé

Les objets partagés occupent une place limitée mais essentielle dans les bases de codes. En particulier, il permettent de révéler le plein potentiel des architectures parallèles, multicœurs et multiproesseurs, modernes. Dans certains cas, des objets ad-hoc sont spécialisés par des programmeurs experts afin de mieux répondre aux besoins de synchronisation et de performance d'un programme.

3.2 Graphe d'indistinguabilité

L'analyse que nous avons faite plus haut indique que les objets partagés jouent un rôle essentiel dans les applications informatiques modernes. Afin d'améliorer les performances de ces objets, les développeurs font parfois usage d'implémentations ad-hoc, propres aux cas d'usage rencontrés. On parlera d'objets ajustés. Dans cette section, nous allons présenté les bases théoriques des objets ajustés. Pour ce faire, nous allons introduire un nouvel outil, le graphe d'indistinguabilité, construit à partir de la spécification séquentielle d'un objet partagé. A l'aide de cet outil, plusieurs résultats seront énoncés qui permettent de jauger la propention d'un objet à passer à l'échelle (scalabilité). Notons que pour des raisons de facilité de lecture les preuves de ces résultats apparaissent en annexe.

3.2.1 Modèle de calcul

On se positionnne dans le modèle de calcul distribué standard basé sur la mémoire partagée. Ce modèle est détaillé en Annexe A, et est rappelé dans ce qui suit.

On considère un ensemble de $n \geq 2$ threads qui communiquent via des objets partagés. Chaque objet respecte une spécification séquentielle, également connue sous le nom de **type de données**. Pour un objet O, nous notons $O.T$ son type de données. Les objets fondamentaux de ce modèle sont les registres, auxquels les threads ont accès pour lire et écrire. En outre, la mémoire met à disposition des primitives de synchronisation communes, telles que *compare-and-swap* et *test-and-set*, pour permettre la mise à jour des registres.

Les registres servent à l'implémentation d'objets de niveau supérieur, tels que les *map*, les files et les arbres. Nous nous concentrons ici sur les objets qui sont à la fois déterministes, linéarisables et sans attente.

Certains objets peuvent présenter des usages restreints. Par exemple, ils peuvent être conçus pour un usage unique (dit *one-shot*, ce qui signifie que chaque thread ne peut les appeler qu'une seule fois, contrairement aux objets conçus pour une utilisation prolongée (dit *long-lived*). Une autre restriction concerne la partie de l'interface accessible à chaque thread. Les registres à **écrivain unique et lec-**

teurs multiples (SWMR) sont un exemple courant dans la littérature [47]. Pour modéliser ces restrictions, nous introduisons la notion de **permissions d'accès** définie par une relation $O.m$ pour chaque objet O. Cette relation définit les opérations que chaque thread est autorisé à exécuter. Par simplicité, on considèrera que chaque opération définit par le type de données $O.T$ doit être exécutable par au moins un thread.

Les threads peuvent accéder à des registres communs lorsqu'ils utilisent un objet partagé. Lorsqu'un tel accès se produit et qu'un registre est mis à jour, on parle de **conflit**. Plus précisément, deux opérations c et d, effectuées par des threads distincts, sont **en conflit** lorsqu'elles accèdent à un registre commun que l'une d'elles met à jour. A l'inverse, si c n'entre pas en conflit avec d, alors c est dite **sans conflits** (en anglais, *conflict-free*) avec d. Les opérations c et d sont dites en **conflit de mise à jour** (*update-conflict*) lorsqu'elles mettent toutes les deux à jour le même registre [74]. L'opération c est qualifiée d'**invisible** pour d si elle n'effectue pas de mise à jour sur un registre lu par d [5]. Lorsqu'il n'y a pas de conflits pour toute paire d'opérations, l'implémentation est dite **sans conflits**. Dans ce cas, les données partagées sont en lecture seule. Si les threads n'accèdent pas à un registre commun, l'implémentation est dite **parallèle à accès disjoints** (en anglais, disjoint access parallelism, aka., DAP). On dira aussi qu'elle est *share nothing*.

Pour rappel, une implémentation est scalable lorsqu'elle améliore ses performances avec l'augmentation du nombre de threads. En pratique, plusieurs paramètres peuvent influencer la capacité d'une application à être scalable, tels que la taille et la localité de l'ensemble des données. Lorsque la charge de travail est suffisamment parallèle, le facteur clé est le taux de conflit. Sur le matériel moderne, une implémentation peut passer à l'échelle lorsque les conflits sont rares. Nous avons vu ceci lors du Chapitre 2. Il est donc crucial de comprendre dans quels cas une telle implémentation pour un objet partagé est réalisable.

3.2.2 Motivation

En calcul distribué, un objet central est le consensus, qui permet à un groupe de threads de s'accorder sur une valeur commune. Pour un type de données T, le

nombre maximal de threads pouvant parvenir à un accord en utilisant des objets de type T et des opérations de lecture/écriture sur des registres est défini comme le **nombre de consensus** de T. Ce nombre est noté $CN(T)$, et nous notons CN_k l'ensemble des objets partagés ayant un nombre de consensus égal à k. Par exemple, les registres appartiennent à CN_1, signifiant qu'ils sont trop faibles pour résoudre l'accord. En revanche, des objets comme les files permettent d'atteindre un consensus entre plusieurs threads [24]. Par la suite, on dira d'une opération c de T qu'elle possède **un pouvoir de consensus** si T, restreint à cette opération, a un nombre de consensus supérieur à 1.

Le nombre de consensus d'un objet partagé capture sa capacité de synchronisation. Cependant, il n'est pas un bon indicateur de la capacité d'un objet à passer à l'échelle. Ce résultat est bien connu de la littérature. Par exemple, prenons le cas des objets appartenant à CN_1. Un objet *snapshot* (capture d'état global) nécessite un nombre d'étapes de l'ordre de $\Omega(n)$ lorsqu'il est implémenté sur des registres partagés [16]. À l'inverse, un *max register* capable de stocker jusqu'à m valeurs peut être implémenté en $O(\min(n, \log(m)))$ étapes [3]. Bien que ces deux objets performent différemment quand le nombre de threads augmente, ils appartiennent tous les deux à CN_1.

De ce qui précède, nous avons donc besoin d'un indicateur permettant de mesurer la disposition d'un objet partagé à passer à l'échelle. Dans ce but, nous introduisons le graphe d'indistinguabilité, une structure construite à partir du type de données de l'objet en question. Ce graphe capture les interactions entre les opérations des threads sur l'objet partagé.

Dans la suite de ce chapitre, nous allons expliquer le lien entre l'évolution des performances d'un objet quand la taille du système augmente (scalabilité) et la densité du graphe d'indistinguabilité. En effet, la manière dont les opérations d'un objet interagissent — représentée par la densité des arêtes dans ce graphe — a un impact direct sur la capacité de l'objet à passer à l'échelle.

3.2.3 Définition

Considérons un objet de type T, un état s et un multiensemble d'opérations, noté B, c'est-à-dire un ensemble où les répétitions d'opérations sont permises. Soit

x une permutation des éléments de B, aussi notée $x \in perm(B)$, et soit c un élément de B. La réponse de c à partir de l'état s correspond à la valeur retournée par c après avoir appliqué toutes les opérations précédentes dans x, jusqu'à c, en partant de l'état initial s.

Deux permutations, notées x et x', sont dites **indistinguables** de s pour une opération c si les conditions suivantes sont remplies :

— L'opération c retourne la même valeur dans les deux permutations.

— Il existe un état commun s' accessible après l'exécution de c dans les deux permutations x et x'.

Nous écrivons alors $x \overset{c,s}{\sim} x'$. Cette relation est symétrique. Pour simplifier, nous omettons parfois c et s, et notons simplement $x \sim y$ lorsque cela est clair dans le contexte.

Une représentation utile de cette relation est le **graphe d'indistinguabilité**. Pour un état s donné et un multiensemble B, les nœuds du graphe $\mathcal{G}_T(B, s)$ correspondent à toutes les permutations possibles de B. Il existe une arête entre deux nœuds x et x' dans ce graphe, labellisé par les opérations C, si et seulement si x et x' sont indistinguables de s pour toutes les opérations $c \in C$.

La **classe d'indistinguabilité** d'une permutation x dans l'état s, notée $[x]_s^B$, est la composante connexe du graphe $\mathcal{G}_T(B, s)$ à laquelle x appartient. En d'autres termes, c'est l'ensemble des permutations de B contenant x qui sont indistinguables deux à deux à partir de l'état s.

Une opération c est dite **labellisante** dans le graphe $\mathcal{G}_T(B, s)$ si elle est utilisée pour labelliser toutes les arêtes du graphe. Lorsque le multiensemble B et l'état s sont clairs dans le contexte, ces notations sont abrégées en $\mathcal{G}_T$ et $[x]$.

Ces concepts s'étendent naturellement aux objets partagés. Plus précisément, soit O un objet quelconque, les graphes d'indistinguabilité de O, notés Γ_O, sont des graphes $\mathcal{G}_T(B, s)$, où s est un état possible de $O.T$, et B respecte les permissions d'accès $O.m$. Cela signifie que chaque thread t est associé à une opération unique c_p dans B, et que c_p est autorisée pour t par $O.m[p]$.

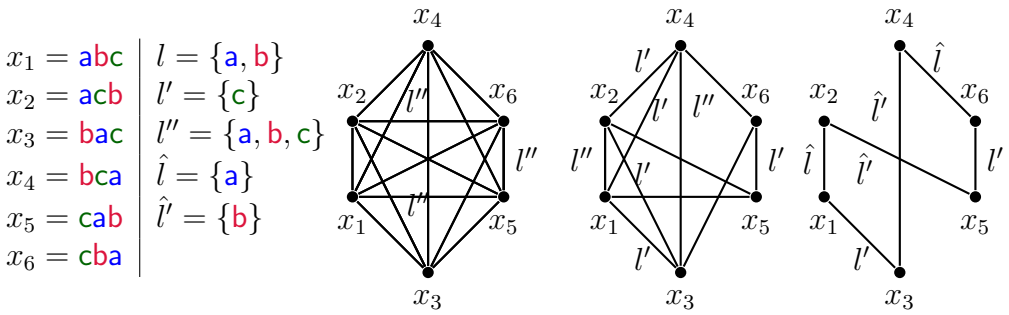

FIGURE 3.5 – De gauche à droite, $\mathcal{G}(a, b, c)$ d'une référence ($a = set(1)$, $b = set(2)$, $c = get()$), d'un *set* ($a = add(1)$, $b = add(1)$, $c = contains(1)$) et d'un compteur ($a = inc(1)$, $b = inc(3)$, $c = inc(5)$). Le label par défaut pour une arête est l.

Un exemple

La Figure 3.5 illustre les notions décrites précédemment, en présentant les graphes d'indistinguabilité de trois types d'objets courants : une référence, un *set*, et un compteur. Ces objets sont formellement spécifiés dans la deuxième colonne de la Table 3.1. La figure montre comment les opérations $\{a, b, c\}$ appliquées à partir de l'état initial engendrent un graphe d'indistinguabilité.

Dans ce graphe, on retrouve un total de $3! = 6$ nœuds, correspondant aux permutations des trois opérations.

Pour l'objet de type référence (Figure 3.5a), le graphe est complet puisque l'opération *set* ne renvoie pas de réponse. Toutes les arêtes du graphe sont labellisées par défaut avec l'ensemble d'opérations $l = \{a, b\}$. Lorsque l'opération *set* précède directement l'opération *get*, cette dernière renvoie la même réponse. Ainsi, l'opération c labellise les arêtes reliant les paires de nœuds (x_1, x_4), (x_2, x_3), et (x_5, x_6).

Passons maintenant à l'objet de type *set* (Figure 3.5b). Lorsque l'opération $c = contains(1)$ n'est pas la première à être exécutée, elle devient labellisante, car dans ce cas, elle renvoie toujours *true*. Cette opération labellise également l'arête entre x_5 et x_6, car elle ne peut pas observer l'ordre des deux autres opérations. De plus, lorsque les deux opérations *add* sont effectuées dans le même ordre, leurs

réponses ne changent pas. Dans ces situations, a et b sont elles aussi labellisantes.

Enfin, la figure la plus à droite illustre les incréments de valeurs 1, 3 et 5 appliqués à un compteur. Pour chaque permutation donnée, la dernière opération voit le même résultat si les deux précédentes sont échangées. Cela explique la présence de 6 arêtes au total.

3.2.4 Distinguabilité

Comme on peut l'observer dans Figure 3.5, tous les graphes d'indistinguabilité sont connexes par arcs. Il existe donc qu'une seule classe d'indistinguabilité pour ces exemples. En général, le nombre maximum de classes d'indistinguabilité est égal à $|B|$, car si deux permutations x et y partagent la même première opération ($x[0] = y[0]$), alors $[x] = [y]$.

Lorsqu'un type de données T a la capacité de distinguer jusqu'à l classes d'indistinguablité parmi $k = |B|$ opérations, on dit que T est $\mathcal{D}(k, l)$. Il est facile de voir que si T est $\mathcal{D}(k, l)$, il est au plus $\mathcal{D}(k, l + 1)$. Par exemple, un compteur est $\mathcal{D}(2, 2)$ mais seulement $\mathcal{D}(3, 1)$, car la troisième opération ne peut pas distinguer comment les précédentes ont été ordonnées (Figure 3.5c). C'est également le cas pour un *set* (Figure 3.5b) et l'objet *fetch-and-increment*.

Dans ces exemples, la transition clé est vers $\mathcal{D}(k, 1)$. Ci-dessous, nous établissons que lorsque cette transition existe et que T est lisible, c'est-à-dire que n'importe quel thread peut récupérer l'état le plus récent, k est le nombre de consensus de T.

Theorem 1. *Considérons un type de données lisible $T \notin CN_1$. Alors, $CN(T) = \max\{k : \exists l \geq 2 \cdot T \in \mathcal{D}(k, l)\}$*

Ce résultat est à rapprocher de celui de Ruppert [67] qui caractérise le nombre de consensus des objets read-modify-write disposant d'une opération de lecture. Il diffère de ce dernier dans le sens où nous considérons des objets quelconques et non simplement read-modify-write. La technique de preuve est toutefois très similaire (voir Annexe B.1).

3.2.5 Prédire la mise à l'échelle

Les résultats qui suivent posent les bases de l'analyse des graphes d'indistinguabilité pour comprendre, construire et évaluer les objets ajustés. Ces résultats relient la scalabilité d'un objet partagé O, à savoir sa performance quand le nombre de threads augmente, à la forme de ses graphes d'indistinguabilité Γ_O. Tout d'abord, nous fournissons des conditions nécessaires et suffisantes sur ces graphes pour qu'une implémentation sans conflits existe. Ensuite, nous nous penchons sur les conflits de mise à jour ainsi que sur les opérations invisibles.

Absence de conflits

Dans la Section 3.2.1, nous avons vu qu'une implémentation est dite sans conflits quand pour toute paire d'opérations, ces opérations ne modifient pas un registre commun. Pour commencer, nous examinons les objets dits one-shot, c'est-à-dire ceux qui peuvent être invoqués au plus une fois par thread. On a le résultat suivant :

Proposition 1. Supposons que l'objet O soit one-shot. Il existe une implémentation sans conflits de O si, et seulement si, B est labellisant pour tout $\mathcal{G}_T(B, s)$ dans Γ_O.

Dans la Section 2.5, nous avons introduit la règle de SIM-commutativité [11]. Pour rappel, cette règle détermine quand l'intervalle d'une exécution concurrente peut être mis en œuvre de manière sans conflits. Ceci correspond à la partie suffisante de la Proposition 1 : une telle implémentation existe lorsque toutes les opérations de B dans l'intervalle sont labellisantes pour $\mathcal{G}_T(B, s)$, où s est l'état de l'objet au début de l'intervalle. La Proposition 1 montre que c'est également nécessaire quand les objets déterministes.

On peut étendre ce résultat aux objets *long-lived*, à savoir utilisables plusieurs fois par chaque thread. Le résultat est énoncé ci-dessous.

Proposition 2. Il existe une implémentation sans conflits de O si, et seulement si, B est labellisant pour tout $\mathcal{G}_T(B, s)$ dans Γ_O, avec $|B| = 2$.

Ce résultat indique qu'une implémentation sans conflits existe lorsque n'importe quelle paire d'opérations issues de threads distints est commutative. Par

exemple, ceci peut se produire lorsque les threads accèdent à différents segments d'un grand objet. Un cas typique est une base de données clé-valeur où chaque thread est responsable d'une certaine plage de clés. Lorsqu'une implémentation sans conflits existe, il est possible de faire une implémentation DAP. Pour ce faire, les données partagées, qui sont forcément en lecture seule seront dupliquées.

Dans le Listing 3.1, la classe `AtomicWriteOnceReference` ne satisfait pas les pré-requis de la proposition Proposition 2 lorsque B contient une opération `set`. Toutefois, cet objet demeure performant même avec un grand nombre de threads—nous le verrons empiriquement dans le Chapitre 4. Une observation importante ici est que le graphe d'indistinguabilité de la classe `AtomicWriteOnceReference` est particulièrement dense. En effet, permuter les opérations avant (ou après) une opération `set` n'affecte ni la réponse de ces opérations, ni l'état de l'objet. Les résultats qui suivent généralisent cette observation. Ils explorent le lien entre la forme des graphes d'indistinguabilité d'un objet et sa capacité à passer à l'échelle.

Conflits de mise à jour

Comme souligné au chapitre précédent, l'aptitude d'un objet à passer à l'échelle est liée aux nombres de conflits. Intuitivement, un conflit survient lorsque deux opérations s'influencent mutuellement. Pour formaliser cette idée, nous introduisons la notion de ***moverness*** [43, 53, 76].

Considérons un graphe d'indistinguabilité $\mathcal{G}_T(B, s)$ et une permutation $x = c_1 \dots c_{m\geq 2}$ de B. On dit qu'une opération c_i se **déplace à gauche** dans x lorsque c_i labellise l'arête (x, x') dans $\mathcal{G}_T(B, s)$, où $x' = c_1 \dots c_i c_{i-1} \dots c_m$ (c'est-à-dire que l'ordre des deux opérations c_i et c_{i-1} peut être échangé sans affecter les résultats). Par extension, une opération se déplace à gauche dans $\mathcal{G}_T(B, s)$ si elle se déplace à gauche dans toutes les permutations possibles de B. Si cette propriété est vraie pour tous les graphes d'indistinguabilité associés à l'objet, on dit que l'opération est un ***left-mover***.

Les opérations d'écriture à l'aveugle (*blind write*), c'est-à-dire celles qui ne renvoient pas de valeur (ou renvoient une valeur vide), sont souvent des *left-movers*. Par exemple, considérons un objet de type `Set`. Si l'opération $add(x)$ est une écriture à l'aveugle (comme spécifié pour l'objet S_2 dans la Table 3.1), elle peut

se déplacer vers la gauche vis-à-vis de toutes les autres opérations. En effet, pour toute opération c, soit *(i)* c commute avec $add(x)$ et l'état après $c.add(x)$ est le même que suite à $add(x).c$, ou bien *(ii)* c est l'ajout/retrait de x et $c.add(x)$ mène au même état que $add(x)$. Un autre exemple est l'opération *offer* pour une file d'attente (objet Q_1 dans la Table 3.1), qui peut se déplacer vers la gauche par rapport à *poll*. Cela signifie que si un seul thread appelle *offer*, alors cette opération est un *left-mover*. Un tel cas illustre une situation où ajuster l'objet pour son usage serait pertinent. Nous le détaillerons en Section 3.3.

La proposition ci-dessous affirme que les *left-movers* peuvent être implémentés de manière efficace, sans conflits de mise à jour.

Proposition 3. L'opération c est implémentable sans conflits de mise à jour si c se déplace à gauche dans chaque $\mathcal{G}_T(B, s)$ de Γ_O. Lorsque c possède un pouvoir de consensus, cette condition est également nécessaire.

Opérations invisibles

Une opération est qualifiée de right-mover si elle n'affecte pas la valeur de retour d'une opération qui lui succède lorsqu'elles échangent leurs positions. Plus précisément, une opération c_i se **déplace à droite** dans une permutation $x = c_1 \dots c_{i-1} c_i \dots c_m$ lorsque c_{i-1} labellise l'arête (x, x') dans $\mathcal{G}_T(B, s)$, où $x' = c_1 \dots c_i c_{i-1} \dots c_m$. Autrement dit, c_i se déplace à droite dans x si et seulement si c_{i-1} se déplace à gauche dans la permutation x'. On définit formellement les *right-movers* de manière analogue aux *left-movers*. En d'autres termes, c se déplace à droite dans $\mathcal{G}_T(B, s)$ si elle se déplace à droite dans toutes les permutations de B et par extension, c est ***right-mover*** si elle se déplace à droite dans tous les graphes d'indistinguabilité de l'objet.

A l'inverse des *left-movers*, les *right-movers* n'influencent pas les opérations des autres threads. Étant donné qu'elles n'ont pas d'effets de bord, les opérations en lecture seule sont des right-movers typiques. Par exemple, pour un objet de type `Map`, c'est le cas de l'opération $contains(k)$ qui retourne la valeur stockée sous la clé k. Cependant, ces opérations ne sont pas les seules. Par exemple, toujours pour le type `Map`, $remove(k)$ se déplace à droite de toute opération qui n'est pas un $contains(k)$ quand il n'y a pas de valeur de retour (S_2 dans Table 3.1). Ainsi, si

seul le thread retirant la clé k peut exécuter *contains*(k), cette opération est aussi un *right-mover*.

Le résultat suivant établit que les right-movers peuvent être implémentés de manière invisible. A savoir, ils peuvent être coder de manière à ne jamais effectuer de mise à jour sur un registre accessible par une opération d'un autre thread.

Proposition 4. L'opération c est implémentable de manière invisible si c se déplace à droite dans chaque $\mathcal{G}_T(B, s)$ de Γ_O.

Résumé

Cette section a introduit la notion de graphe d'indistinguabilité pour les objets partagés. Ce graphe traduit la capacité de l'objet à distinguer différents entrelacements d'opérations. En particulier, nous avons relié la connexité du graphe à la puissance de consensus de l'objet. Nous avons ensuite énoncé plusieurs résultats liant la scalabilité de l'objet partagé à la densité de ses graphes d'indistinguabilité. Nos résultats indiquent qu'un graphe dense est un bon indicateur de la capacité de l'objet à pouvoir passer à l'échelle. Par exemple, quand les conditions sont réunies, il est possible d'implémenter une opération de manière sans conflit, sans conflits de mise à jour, ou encore à ce qu'elle soit invisible pour les opérations des autres threads. La section qui suit va tirer parti de ces observations pour définir le principe d'objet ajusté.

Counter	$[true]\ rmw(f,x)\ [s' = f(s,x) \wedge \mathbf{r} = s']$ $[true]\ inc()\ [s' = s+1 \wedge \mathbf{r} = s']$ $[true]\ get()\ [\mathbf{r} = s]$ $[true]\ reset()\ [s' = 0]$ C_1	$[true]\ rmw(f,x)\ [true]$ $[true]\ inc()\ [s' = s+1 \wedge \mathbf{r} = s']$ $[true]\ get()\ [\mathbf{r} = s]$ $[true]\ reset()\ [s' = 0]$ C_2	$[true]\ rmw(f,x)\ [true]$ $[true]\ inc()\ [s' = s+1]$ $[true]\ get()\ [\mathbf{r} = s]$ $[false]\ reset()\ [s' = 0]$ C_3
Set	$[true]\ add(x)\ [s' = s \cup \{x\} \wedge \mathbf{r} = x \notin s]$ $[true]\ remove(x)\ [s' = s \setminus \{x\} \wedge \mathbf{r} = x \in s]$ $[true]\ contains(x)\ [\mathbf{r} = x \in s]$ S_1	$[true]\ add(x)\ [s' = s \cup \{x\}]$ $[true]\ remove(x)\ [s' = s \setminus \{x\}]$ $[true]\ contains(x)\ [\mathbf{r} = x \in s]$ S_2	$[true]\ add(x)\ [s' = s \cup \{x\}]$ $[true]\ remove(x)\ [true]$ $[true]\ contains(x)\ [\mathbf{r} = x \in s]$ S_3
Queue	$[true]\ offer(x)\ [s' = s \circ x]$ $[true]\ poll()\ [if\ \lvert s\rvert = 0\ then\ \mathbf{r} = \perp\ else\ \mathbf{r} = head(s) \wedge s' = s \setminus \{head(s)\}]$ $[true]\ contains(x)\ [\mathbf{r} = x \in s]$ Q_1		
Reference	$[x \in \texttt{Addr}]\ set(x)\ [s' = x]$ $[true]\ get()\ [\mathbf{r} = s]$ R_1	$[x \in \texttt{Addr} \wedge s = \perp]\ set(x)\ [s' = x]$ $[true]\ get()\ [\mathbf{r} = s]$ R_2	
Map	$[true]\ put(k,v)\ [s'[k] = v \wedge \mathbf{r} = s[k]]$ $[true]\ remove(k)\ [s'[k] = \perp \wedge \mathbf{r} = s[k]]$ $[true]\ contains(k)\ [\mathbf{r} = (s[k] \neq \perp)]$ M_1	$[true]\ put(k,v)\ [s'[k] = v]$ $[true]\ remove(k)\ [s'[k] = \perp]$ $[true]\ contains(k)\ [\mathbf{r} = (s[k] \neq \perp)]$ M_2	

TABLE 3.1 – Versions ajustés de types de données commun décrites en logique de Hoare [32]. Nous désignons par s l'état initial de l'objet, par s' le nouvel état résultant de l'application des effets de bord de l'opération, et par $\mathbf{r}$ la valeur de retour de cette opération. Lorsque ces informations ne sont pas spécifiées, cela signifie que l'état reste inchangé et que la valeur de retour est inexistante ($\perp$).

3.3 Les objets ajustés

Un objet ajusté est un objet partagé adapté à un usage spécifique dans un programme. Les objets ajustés passent mieux à l'échelle que leurs homologues génériques car ils ont des graphes d'indistinguabilité plus denses. Cette section définit formellement les objets ajustés, puis établit un tel résultat. En outre, elle présente une méthodologie permettant d'ajuster un objet en limitant la manière dont les threads y accèdent et/ou en modifiant son interface.

3.3.1 Principes

Définition

Dans de nombreuses circonstances, l'accès à un objet partagé se caractérise par une certaine asymétrie. Un exemple fondamental est celui d'un registre à un seul écrivain et plusieurs lecteurs (SWMR). Ce type de registre est conçu de manière à ce qu'un seul thread, l'écrivain, soit capable d'invoquer l'opération *write*. Quant aux autres threads, qui agissent en tant que lecteurs, leur interaction se limite à l'invocation de *read*. Un autre cas courant est celui d'une file d'attente à un seul producteur et un seul consommateur (voir par exemple la Section 3.1.2). Dans ce contexte, le producteur est seul à invoquer l'opération *offer*, tandis que le consommateur intervient uniquement via l'opération *poll*.

Dans les situations décrites ci-dessus, l'objet est ajusté afin de traduire l'asymétrie des rôles respectifs. Plus précisément, rappelons que nous représentons par $O.m$ les permissions d'accès associées à l'objet partagé O. Cet ajustement contraint $O.m$ en fonction des opérations exécutées par les threads dans le cadre du cas d'utilisation spécifique. Par exemple, pour un registre SWMR, un seul thread, noté w, l'écrivain, dispose de $\mathit{write} \in O.m[w]$. En revanche, pour chaque autre thread r, la restriction est telle que $O.m[r] = \{\mathit{read}\}$.

Un deuxième type d'ajustement réside dans la modification de l'interface de l'objet. Par exemple, comme illustré dans la Figure 3.4, la valeur de retour d'un appel peut ne pas être utilisée dans certaines parties d'un programme. Autrement dit, dans ces cas particuliers, l'opération devient aveugle. Un autre exemple est fourni par la Figure 3.3, où l'interface de l'objet n'est utilisée que partiellement,

étant réduite à un sous-ensemble d'opérations. Ces modifications d'interface relèvent de notre deuxième forme d'ajustement, où l'ajustement correspond à un sous-typage.

Pour formaliser la notion d'objets ajustés, nous nous appuyons sur les travaux de Liskov and Wing [54], qui définissent les concepts de types et sous-types. Plus précisément, un type de données S est un sous-type de T si, et seulement si, deux conditions sont respectées : *(i)* il existe une fonction d'abstraction qui associe à chaque état (valide) du sous-type S un état du super-type T ; *(ii)* le sous-type S conserve les méthodes du super-type T. Nous dirons que S est un **sous-type restreint** de T si S est un sous-type de T et T implémente uniquement les méthodes définies par S. Sur cette base, nous pouvons alors définir formellement la notion d'objet ajusté.

Definition 1. Soient O et O' deux objets partagés. L'objet O **ajuste** O' lorsque $O'.T$ est un sous-type restreint de $O.T$ et $O.m \subseteq O'.m$.

En d'autres termes, un objet est considéré comme ajusté lorsque soit ses permissions d'accès sont restreintes, soit son interface est modifiée. Comme le précise la définition ci-dessus, ces deux formes d'ajustement peuvent être combinées. Nous allons illustrer ce principe à travers quelques exemples.

Illustration

Pour mieux comprendre la définition précédente, considérons l'exemple d'un objet de type référence, que l'on rencontre couramment dans divers langages de programmation. L'état de cet objet est constitué d'une seule variable s, initialement définie à null ($\perp$). Une référence offre deux opérations : *(i) get*, qui renvoie la valeur de s, et *(ii)* $set(x)$, qui modifie la valeur de s pour lui attribuer x, sous réserve que x soit une adresse valide. Cet objet peut être ajusté en imposant une contrainte d'écriture unique. Dans ce cas, une précondition $v = \perp$ est ajoutée à l'opération *set*. Ce principe est illustré dans Table 3.1, où plusieurs versions ajustées de types de données courants sont présentées. Chaque type de données y est spécifié en utilisant la logique de Hoare [32]. Dans cette table, l'objet référence standard est représenté par R_1, tandis que sa version ajustée est donnée par R_2 (quatrième ligne de Table 3.1).

Une autre forme d'ajustement possible pour un objet référence consiste à restreindre l'accès en écriture à un seul thread. Autrement dit, un seul thread est autorisé à invoquer l'opération *set*, ce qui implique une modification des permissions d'accès de l'objet.

Intérêts

Lorsqu'un objet partagé O ajuste un autre objet partagé O', O' est au moins aussi "puissant" que O. Cela découle du fait que $O'.T$ est un sous-type de $O.T$. Plus formellement :

Proposition 5. Considérons que l'objet O ajuste l'objet O'. Alors, toute tâche distribuée pouvant être résolue avec O (en conjonction avec des registres) peut également être résolue avec O'.

Comme escompté, la proposition réciproque ne se vérifie pas. Nous illustrons cette observation avec Table 3.1. Dans cette table, l'objet S_2 ajuste S_1 en supprimant la valeur de retour de l'opération *set*. Il est aisé de constater que S_1 appartient à la classe CN_2, tandis que S_1 est dans la classe CN_1.

À la lumière de cette observation, il semble naturel de s'attendre à ce que l'ajustement d'un objet réduise le nombre de conflits nécessaires à sa mise en œuvre. En conséquence, cela devrait améliorer ses performances lorsque la contention est élevée, c.à.d. lorsque l'objet passe à l'échelle. Cette propriété fondamentale est démontrée ci-après. Précisément, elle indique que l'ajustement d'un objet augmente (ou a minima conserve) le nombre d'arêtes dans les graphes d'indistinguabilité associés à cet objet.

Proposition 6. Supposons que O ajuste O'. Soit s un état commun à O et O', et B un ensemble d'opérations tel que B respecte à la fois $O.m$ et $O'.m$. Alors, $G_{O'.T}(B, s) \subseteq G_{O.T}(B, s)$.

Ainsi, ajuster un objet ouvre des perspectives pour améliorer son efficience, en particulier à mesure que la taille du système augmente. Dans la section suivante, nous exposons une méthodologie permettant de tirer parti de cet avantage. Nous appliquerons ensuite cette méthodologie à la construction de `DEGO`, une bibliothèque performante d'objets ajustés pour le langage Java, dans le Chapitre 4.

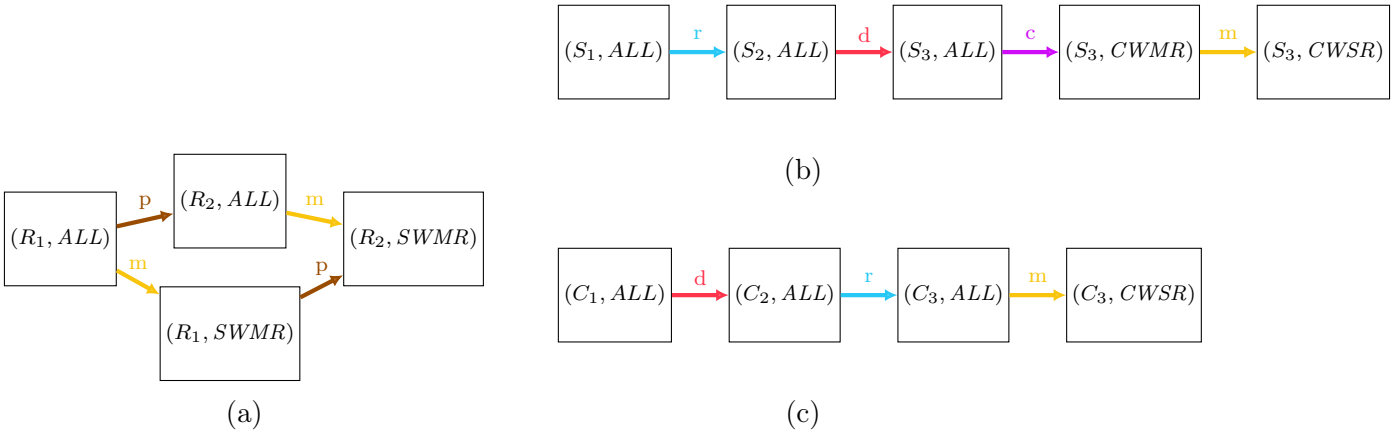

FIGURE 3.6 – Pour ajuster un objet, il est possible de combiner sous-typage et restriction d'accès.

3.3.2 Comment ajuster un objet ?

Ajuster un objet consiste soit à modifier sa spécification séquentielle, soit à restreindre l'accès aux opérations. Dans la Figure 3.6, nous illustrons plusieurs exemples où ces deux techniques sont combinées. Cette figure montre, pour un type de données particulier, comment celui-ci peut être ajusté de manière successive.

Par exemple, dans la Figure 3.6c, le type de données compteur (C_1, ALL) est ajusté étape par étape. Dans cette figure, une paire (X, Y) désigne par X le type de l'objet et par Y les permissions accordées. ALL correspond à la permission par défaut, où un thread peut accéder à toutes les opérations de l'interface. Cela correspond au modèle classique multi-écrivains, multi-lecteurs, également connu sous le nom de $MWMR$.

Ci-dessous, nous décrivons les différents ajustements présentés dans la Figure 3.6 ainsi que la méthodologie employée pour les obtenir.

Un moyen d'ajuster un objet est de supprimer une opération, comme indiqué par la notation $\xrightarrow{\text{d}}$ dans la Figure 3.6. Par exemple, cela se produit avec l'opération *reset* dans l'objet compteur. Dans C_3 (voir Table 3.1), la précondition de cette opération est définie à *false*. Ainsi, l'opération n'apparait plus dans les graphes d'indistinguabilité de l'objet. Un résultat similaire est obtenu lorsque la postcondition est nullifiée, comme pour l'opération read-modify-write (rmw) dans C_2. Dans ce

cas, l'opération n'a plus d'effet et elle est donc toujours labellisante dans le graphe d'indistinguabilité.

Un autre ajustement possible consiste à renforcer les préconditions d'une opération, comme le montre la notation $\xrightarrow{\text{p}}$ dans la Figure 3.6. Par exemple, cela permet d'exiger que la référence R_2 soit à écriture unique dans Table 3.1. Le raisonnement est similaire : supposons qu'une opération d ait une précondition plus forte qu'une autre opération c. Si la précondition de c est satisfaite dans une permutation x mais que celle de d ne l'est pas, alors d échoue silencieusement, ce qui renforce son potentiel à être labellisante.

Une troisième méthode consiste à affaiblir la postcondition d'une opération, par exemple en annulant la valeur de retour, ce qui transforme l'opération en une écriture aveugle. Cela se produit avec l'opération *add* dans S_2 (voir Table 3.1 et la notation $\xrightarrow{\text{r}}$ dans la Figure 3.6). Cette modification peut densifier les graphes d'indistinguabilité en augmentant la capacité des autres opérations à être des *left-movers* ou des *right-movers* (voir Section 3.2).

Une quatrième méthode d'ajustement est de restreindre l'accès aux opérations, par exemple en exigeant que les écritures soient commutatives, comme l'indique $\xrightarrow{\text{c}}$ dans la Figure 3.6. Dans cette figure, l'opération *set* est ajustée en $(S_3, CWMR)$, où toutes les opérations ayant des effets de bord (telles que *add* et *remove*) doivent être commutatives lorsqu'elles sont exécutées par des threads concurrents. Dans un graphe d'indistinguabilité, deux opérations commutatives c et d peuvent labelliser toutes les permutations où elles diffèrent seulement par leur ordres d'exécution.

Les opérations peuvent également être ajustées pour être asymétriques, c'est-à-dire que certains threads ne sont autorisés à invoquer que certaines opérations, comme l'indique la notation $\xrightarrow{\text{m}}$ dans la Figure 3.6. Cela inclut des objets à un seul écrivain et plusieurs lecteurs ($SWMR$), ou à plusieurs écrivains et un seul lecteur ($MWSR$). Dans certains cas, les écrivains peuvent n'exécuter que des opérations commutatives ($CWMR$ et $CWSR$). Sans surprise, ces ajustements densifient également les graphes d'indistinguabilité associés à l'objet.

Comme le montre la Figure 3.6, plusieurs ajustements peuvent être combinés pour un même objet. Dans le cas le plus général, ces ajustements forment un graphe orienté acyclique, illustrant les différentes étapes d'ajustement possibles.

Résumé

Dans cette section, nous définissons formellement la notion d'ajustement d'un objet partagé. Elle est comprise comme une forme restreinte de sous-typage et/ou de retrictions de l'interface à certains threads. Un ajustement est motivé par l'augmentation du nombre d'arrêtes dans le graphe d'indistinguabilité. En effet, comme nous l'avons vu dans en Section 3.2, un graphe dense est plus scalable, c'est à dire qu'il est susceptible d'exister une mise en œuvre de l'objet qui passe mieux à l'échelle avec le nombre de threads y accédant.

3.4 Conclusion du chapitre

Au début de ce chapitre, nous avons illustré l'adoption croissante des objets partagés dans le développement de systèmes informatiques. Cette tendance claire est liée à l'omniprésence des architectures multicœurs et multiproesseurs dans les systèmes informatiques. Nous avons analysé l'utilisation des objets partagés dans 50 projets open source de la fondation Apache, révélant que, bien que la proportion d'objets partagés déclarés reste faible, leur rôle demeure fondamental pour exploiter efficacement les ressources matérielles. Nous avons également réalisé une analyse statique des appels de méthodes pour plusieurs types de données afin d'estimer dans quelle proportion leur spécification séquentielle était utilisée. Nous constatons qu'un sous-ensemble seulement de la spécification est exploité et que certaines méthodes sont utilisées plus fréquemment que d'autres. Nous avons ensuite présenté plusieurs constructions ad-hoc, dites par la suite *ajustées*, qui améliorent les performances des programmes parallèles. Ces constructions sont écrites par des programmeurs experts soucieux d'optimiser les performances de leurs programmes pour certains cas d'usage bien choisis.

Fort de ce constat, nous avons ensuite proposé une définition du principe d'ajustement. Ajuster un objet revient à restreindre son interface (exprimer sous la forme d'un sous-typage) et/ou à rendre asymétrique son utilisation par les threads. Nous avons vu comment un ajustement se traduit par une augmentation du nombre d'arcs dans les graphes d'indistinguabilité de l'objet. Un graphe dense rend l'objet

plus susceptible de passer à l'échelle relativement au nombre de threads qui l'utilisent (scalabilité). Une fois ce principe défini, nous allons illustrer différents types d'ajustements possibles. Ces exemples motiveront la bibliothèque d'objets ajustés (DEGO) qui sera décrite et évaluée dans le chapitre à venir.

Chapitre 4

Implémentation et Évaluation

Dans le chapitre précédent, nous avons mené une analyse approfondie de l'utilisation des objets partagés dans les programmes modernes. Cette analyse a révélé que certains objets partagés sont ajustés pour des cas d'usage spécifiques par les développeurs experts. Nous avons proposé la première formalisation de ce *principe d'ajustement*. Pour ce faire, nous avons introduit le graphe d'indistinguabilité. Nous avons montré que plus ce graphe est dense (et connexe), plus l'objet associé est susceptible d'avoir une implémentation dont les performances passent à l'échelle. Ainsi, un ajustement est compris et justifié au regard des changements sur ce graphe. En fin de chapitre, plusieurs approches pour ajuster un objet ont été présentées.

Dans le chapitre qui s'ouvre, nous allons appliquer les résultats et principes introduits lors du chapitre précédent. Nous présentons `DEGO`, une bibliothèque d'objets ajustés dont la performance est significativement plus élevée que celle des objets du JDK dans certains contextes.

En premier lieu, nous décrivons en détail l'implémentation de la bibliothèque `DEGO` (Section 4.1). Puis, nous exposons les résultats expérimentaux afin d'évaluer les performances des objets ajustés dans des cas concrets. Ceci est fait par une série de micro-benchmarks (Section 4.2), comparant les performances des objets `DEGO` avec leurs équivalents dans la bibliothèque `java.util.concurrent`. Ensuite, nous poursuivons avec une évaluation de `DEGO` dans une application concrète de type réseau social, inspirée de Retwis (Section 4.3). Cette seconde

évaluation nous permet d'observer le comportement des objets ajustés dans des environnements proches de cas d'usage réels. Le code source de la bibliothèque `DEGO` ainsi que les microbenchmarks associés sont disponibles à l'adresse suivante : `https://github.com/BoubacarKaneTSP/DegradableObject`.

4.1 La bibliothèque `DEGO`

Au cours de cette section, nous mettons en pratique les concepts précédemment exposés. Le résultat est une bibliothèque d'objet ajustés pour le langage Java, baptisée `DEGO`. Cette bibliothèque implémente les objets énumérés dans la Table 3.1. Elle comprend autour de 11000 lignes de code. Ci-après, nous donnons tout d'abord un aperçu général de la bibliothèque `DEGO`, avant de nous plonger dans certains aspects clés de son implémentation.

Vue d'ensemble

La bibliothèque `DEGO` organise les objets ajustés en plusieurs catégories, en fonction de deux aspects : leurs types de données et leurs permissions d'accès. Les types de données sont ceux disponibles dans le *package* `java.util.concurrent` du JDK, par la suite abrégé en JUC. Cela inclut des collections (comme les *set*, *list* et *map*), ainsi que d'autres objets courants tels que le compteur ou la référence. Quant aux permissions d'accès, elles suivent celles mentionnées précédemment, et rappelées ci-dessous :

— (ALL) sans restriction,

— ($SWMR$) un seul thread capable de faire des écritures,

— ($MWSR$) plusieurs écrivains, un seul lecteur,

— ($CWMR$) les accès en écriture des écrivains sont commutatifs, et

— et ($CWSR$) comme précédemment mais avec au plus un lecteur.

Tous les objets ajustés répertoriés dans la Table 3.1 sont implémentés dans la bibliothèque `DEGO`, et nous les détaillerons sous peu. Il est important de noter que `DEGO` ne fournit pas toutes les permissions d'accès pour chaque objet. Toutefois, elle propose des bons principes et des abstractions pratiques permettant de facilement

étendre les permissions d'accès. L'une de ces abstractions est la segmentation, que nous détaillons ci-dessous.

Stratégie d'implémentation : la segmentation

Les objets ajustés qui sont des collections multi-écrivains sont construits à l'aide de **segmentations**. Une segmentation est un tableau d'objets qui peut être dynamique ou statique. Chaque objet au sein de ce tableau est de type *SWMR* et est associé à un seul thread ; par la suite, nous nommons cet objet un **segment**. L'utilisation d'une segmentation permet d'implémenter efficacement un objet ajusté lorsque les écritures sont commutatives (c'est-à-dire dans les cas *CWMR* ou *CWSR*).

Pour illustrer ce concept, considérons l'objet ajusté $(C_3, CWSR)$ présenté en bas de la Figure 3.6. Dans un tel cas, chaque segment est un compteur. Lorsqu'un thread exécute l'opération *inc*, il met à jour le compteur dans son segment respectif. Pour lire la valeur du compteur, un thread somme simplement toutes les valeurs stockées dans les différents segments. On peut montrer que si les opérations *inc* sont unitaires, alors une telle lecture est linéarisable.

La bibliothèque `DEGO` offre plusieurs formes de segmentation. Ainsi, dans une `BaseSegmentation`, la correspondance entre les threads et les segments est statique. Cette approche est implémentée en utilisant un tableau partagé de la classe `CopyOnWriteArrayList` et une variable `ThreadLocal`. En conséquence, pour effectuer une lecture, par exemple lors de l'itération sur la collection, le thread doit parcourir tous les segments. Ainsi, la `BaseSegmentation` est particulièrement adaptée aux charges de travail où les opérations sont principalement des écritures, et moins efficace dans les scénarios avec des lectures fréquentes.

Pour pallier à cette inefficacité, deux types de segmentation alternatifs sont disponibles : `HashSegmentation` et `ExtendedSegmentation`. Avec ces types, chaque élément ajouté à une segmentation porte une information qui permet d'identifier le segment dans lequel il est stocké. Avec la `HashSegmentation`, un élément est placé dans le segment correspondant à sa valeur de hachage. Avec l'`ExtendedSegmentation`, lorsqu'un élément est inséré pour la première fois, il conserve l'information sur le segment où il a été stocké, grâce à un champ dédié

dans l'élément. Ces deux approches éliminent la nécessité de parcourir tous les segments lors de la lecture.

Détails d'implémentation

Pour construire des objets ajustés, tels que ceux utilisant une segmentation, nous nous appuyons sur le mécanisme de `VarHandle` disponible dans le JDK. Ce méchanisme mentionné précédemment en Section 2.3.4 est détaillé ci-après.

Les `VarHandles`, introduits dans la JEP 193, permettent de manipuler une variable Java tout en contrôlant sa cohérence à une granularité fine. Les modèles de cohérence des données fournis par les `VarHandles` incluent `Plain`, `Opaque`, `Release/Acquire` et `Volatile`, classés par modèle de cohérence le plus faible au plus fort (voir les définitions en Section 2.4.1).

Le modèle `Plain` ne fournit aucune garantie mémoire, les opérations pouvant être réordonnées. Ce modèle est équivalent au modèle de mémoire Java de base.

Le modèle `Opaque` assure que les opérations sur une seule variable forment un ordre partiel, les écritures étant totalement ordonnées, garantissant ainsi une cohérence éventuelle. De plus, le modèle `Opaque` garantit que lors d'une lecture, les bits provenant de plusieurs écritures ne se mélangent pas, à condition que ce modèle soit utilisé pour tous les accès.

Le modèle `Release/Acquire` assure la causalité. Par exemple, si un thread p modifie une variable a puis définit un drapeau à 1 avec `setRelease`, un autre thread q doit voir la modification de a s'il constate que le drapeau f est égal à 1 en utilisant `getAcquire`. Ce modèle est couramment utilisé dans les situations où un seul thread peut écrire dans une variable, tandis que d'autres threads la lisent.

Les opérations utilisant le modèle `Volatile` sont linéarisables.

Dans `DEGO`, les `VarHandles` sont employés dans diverses constructions *SWMR*. Nous détaillons ci-dessous les implémentations d'une table de hachage et d'une *skiplist* utilisant ces `VarHandles`.

Pour construire un objet *SWMR*, nous partons d'une implémentation séquentielle tirée, par exemple, du JDK. Ce code est ensuite étendu pour supporter les lecteurs concurrents. Par exemple, dans la `SWMRHashMap`, lorsqu'une nouvelle entrée est ajoutée, si sa clé est déjà présente, elle est mise à jour avec `setVolatile`.

Sinon, un nouveau nœud est créé et inséré de manière atomique dans le *bucket* approprié. Lors d'un appel à `resize`, les nœuds ne peuvent pas être réordonnés à la volée en raison des lecteurs potentiels. À la place, ils sont dupliqués puis insérés dans le nouveau tableau de *bucket* constituant la table de hachage.

Pour la `SWMRSkipListMap`, lorsqu'un nouveau nœud `n` est ajouté, il est inséré comme suit : à chaque niveau de la *skiplist*, on affecte `m` à `n.next`, où `m` est le plus petit nœud supérieur à `n`. Ensuite, le pointeur `next` du nœud précédant `m` est modifié pour pointer vers `n` en utilisant `setRelease`. Au niveau qui contient tous les éléments de la liste, nous exécutons un `setVolatile` pour garantir que l'insertion est globalement visible.

Notre bibliothèque inclut également une file d'attente multi-producteurs à consommateur unique. Cette file d'attente est implémentée sans utiliser les primitives `compareAndSwap` lors de l'appel à *poll*. Au lieu de cela, le thread déplace la tête de la file d'attente de manière non bloquante et appropriée. L'opération *offer* reste identique à celle de la classe `ConcurrentLinkedQueue` du JDK.

Les structures de données partagées à écrivain unique sont courantes dans les applications. Pour l'objet référence, nous utilisons l'implémentation du projet Concurrentli (Listing 3.1). Pour les autres objets, `DEGO` adopte une approche similaire au mécanisme *RCU* (*Read-Copy-Update*) de Linux, en utilisant une copie complète de l'objet et en échangeant la référence de manière atomique avec `setVolatile`. Java fournit déjà une telle implémentation pour certains types de données, comme le tableau `CopyOnWriteArrayList` utilisé pour implémenter la `BaseSegmentation`.

Il convient de noter que dans `DEGO`, les opérations de lecture sur les objets ajustés conservent une cohérence équivalente à celle des objets dans JUC. En particulier, les garanties de cohérence faibles sont maintenues lors de l'itération sur une segmentation.

Résumé

Dans cette section, nous concrétisons les concepts théoriques abordés précédemment en développant la bibliothèque `DEGO`, laquelle implémente des objets ajustés en fonction de leurs types de données et de leurs permissions d'accès. En nous appuyant sur l'exemple du compteur C_3 de la Table 3.1, nous expliquons le principe de segmentation avant de décrire deux autres types de segmentation visant à accélérer la lecture d'un objet segmenté : la `HashSegmentation` et l'`ExtendedSegmentation`. Après une présentation des `VarHandles`, nous détaillons leurs utilisations pour implémenter des structures `SWMR`, comme la `SWMRHashMap` et la `SWMRSkipListMap`. Cette section offre ainsi un aperçu général des différentes implémentations et introduit des aspects techniques essentiels, posant les bases des évaluations de performance abordées par la suite.

4.2 Microbenchmarks

Dans cette section, nous détaillons les résultats issus de notre évaluation, qui vise à mesurer l'intérêt des objets ajustés de la bibliothèque `DEGO`. Pour évaluer les performances des objets ajustés, nous avons mis en œuvre une série de microbenchmarks rigoureux. Ces benchmarks ont été conçus pour simuler des conditions de haute contention, en faisant varier les types de charge de travail (écriture intensive ou *workload* mixte), ainsi que la taille des jeux de données.

4.2.1 Configuration et méthodologie

Matériel et logiciel

Notre évaluation est réalisée sur une machine dotée de 362 Go de DRAM et ayant 4 sockets, chacun équipé d'un processeur Intel(R) Xeon(R) Gold 6230 fonctionnant à 2,10 GHz et disposant de 40 cœurs hyperthreadés. La machine tourne sous Linux 6.1.0-18-amd64 avec Java 22-oracle (OpenJDK). Les benchmarks effectués avec 40 threads (ou moins) sont réalisés sur un seul socket.

Dans `DEGO`, nous avons évalué les objets ajustés suivants :

— `CounterIncrementOnly`, correspondant à $(C_3, CWSR)$;

— `ExtendedSegmentedHashMap`, qui implémente $(M_1, CWMR)$ en utilisant le modèle `ExtendedSegmentation` ;

— `ExtendedSegmentedSkipListMap`, utilisant la même segmentation que pour `ExtendedSegmentedHashMap` ;

— `AtomicWriteOnceReference`, la classe mentionnée dans Listing 3.1 ; et

— `QueueMASP`, une file d'attente multi-producteur à un seul consommateur, correspondant à $(Q_1, MWSR)$.

Ces objets ajustés sont comparés à leurs équivalents dans le JDK, se trouvant dans le *package* `java.util.concurrent`.

Méthodologie de mesure

Nous exécutons chaque benchmark pendant 60 secondes après une phase d'échauffement de 30 secondes. Chaque valeur rapportée est une moyenne calculée à partir de 30 tests. Les threads exécutent un mix d'opérations, détaillé plus bas. Lors d'un appel à une opération, celle-ci est répétée 1000 fois. Les mesures sont ensuite moyennées pour éliminer l'impact du temps nécessaire pour faire un appel à la méthode `System.nanoTime`.

Pour les collections, telles que les *maps*, les threads effectuent des mises à jour commutatives. Cela correspond à une exécution courante où chaque requête est attribuée à un thread spécifique (par exemple, basée sur le hash de la donnée qu'il accède). Les collections commencent avec une taille initiale contenant 16384 éléments, et peuvent stocker jusqu'à 32768 éléments. Les éléments sont générés aléatoirement à l'aide d'une distribution uniforme. Les opérations de type *snapshot* ne sont pas testées, et les opérations de mise à jour ne sont pas composites (par exemple, `putAll`).

Nous évaluons d'abord les objets ajustés listés dans la Table 3.1 sous des scénarios de haute contention. Ensuite, nous évaluons les maps avec différents ratios de mise à jour.

Les paramètres de nos micro-benchmarks correspondent à l'exécution de Synchrobench [21] avec les paramètres suivants : *-u100-f1-l60000-s0-a0-i[16384]-r[32768]-W30-n30*.

Pour chaque micro-benchmark, nous calculons le débit par thread. Les résultats sont exprimés par thread, où une ligne horizontale indique une performance optimale de l'objet quand le nombre de thread augmente. À l'inverse, une ligne descendante indique une diminution de la performances à chaque thread supplémentaire. (Si la ligne monte, cela signifie que l'objet est *hyper-scalable*, ce qui peut être dû au partage du cache entre les cœurs.)

Lorsque plusieurs threads se disputent l'accès au même objet, un thread peut réussir à obtenir son accès, tandis que les autres attendent qu'il soit à nouveau disponible. Ce phénomène peut être observé en examinant via le compteur de performance *perf* l'événement *cycle_ activity.stalls_ total*, qui enregistre le nombre de cycles pendant lesquels au moins un thread matériel est en attente.

Nous recherchons une corrélation entre le débit et le nombre de cycles avec au moins un thread en attente en comparant chaque valeur rapportée avec les autres. Comme nous anticipons une corrélation linéaire entre le débit et le nombre de threads en attente, nous utilisons le *coefficient de corrélation de Pearson*. Ce coefficient varie entre -1 et 1 pour deux ensembles de valeurs A et B. Plus le coefficient se rapproche de 1, plus il indique que lorsque A augmente, B augmente également de manière linéaire. Inversement, un coefficient proche de -1 indique qu'à mesure que A augmente, B diminue. Un coefficient proche de 0 indique une absence de corrélation entre les valeurs de A et celles de B.

4.2.2 Écritures intensives

La Figure 4.1 présente la performance des objets ajustés disponibles dans `DEGO`, évalués dans un contexte de haute contention.

La charge de travail pour les compteur est réalisée avec tous les threads, ces derniers exécutant uniquement `incrementAndGet`. Pour les maps, seule l'opération `put` est appelée. La charge de travail pour les files suit un modèle producteur-consommateur, où tous les threads effectuent 100% d'opérations `offer`, à l'exception d'un thread qui effectue 100% d'opérations `poll`. Pour les références, comme nous testons une implémentation supportant une écriture unique, les threads exécutent uniquement des opérations `get`, une fois que l'objet est initialisé.

Analyse des résultats

Une réduction de 80% des événements *cycle_activity.stalls_total* est observée pour le `CounterIncrementOnly` par rapport à l'`AtomicLong`. Ce gain de performance est attribuable à une diminution significative de la contention sur les ressources partagées grâce à l'utilisation de la segmentation. Nous observons un coefficient de Pearson de -0.93, ce qui indique qu'à mesure que le nombre d'événements *cycle_activity.stalls_total* augmente, le débit d'opération diminue. Ce phénomène est également observé pour les autres objets, avec un coefficient de Pearson moyen de -0.88. Le `CounterIncrementOnly` est 350x fois plus rapide que l'`AtomicLong` lorsque 80 threads accèdent simultanément à l'objet. Java inclut l'objet `LongAdder`, qui utilise une approche similaire à la notre en distribuant les opérations à travers différents segments pour réduire la contention (Voir Section 3.1.2). Cependant, le `LongAdder` implémente la classe `Striped64`, qui emploie en interne `weakCompare&Set` pour les mises à jour. `CounterIncrementOnly` utilise exclusivement des `longs`, car il y a un seul propriétaire par segment, ce qui explique la différence de performance.

L'`ExtendedSegmentedHashMap` démontre des gains de performance allant jusqu'à 4.4x par rapport à la `ConcurrentHashMap`. Cette amélioration est due à une réduction de 23% des événements *cycle_activity.stalls_total* avec l'objet ajusté. De même, la `SkipListMap` dans `DEGO` est jusqu'à 1.7x plus rapide que son homologue dans JUC. Cette amélioration est également expliquée par le fait que les threads sont moins souvent bloqués dans l'implémentation ajustée.

Bien que l'`AtomicReference` exécute, en moyenne, 37% d'instructions supplémentaires par cycle par rapport à l'`AtomicWriteOnceReference`, cette dernière affiche un gain de performance moyen de 11.5x. L'objet `AtomicReference` utilise une variable volatile, ce qui implique l'utilisation de barrières. Lorsqu'une lecture est effectuée, deux barrières sont appliquées après que la valeur contenue dans la variable soit chargée : une barrière `LoadLoad` pour s'assurer que la lecture de la variable n'est pas réordonnée avec les lectures effectuées après la barrière, et une barrière `LoadStore` pour garantir que les écritures effectuées après la barrière ne sont pas ordonnées avant les lectures effectuées avant. Une troisième barrière `StoreLoad` est utilisée pour garantir que les opérations sur la même variable vola-

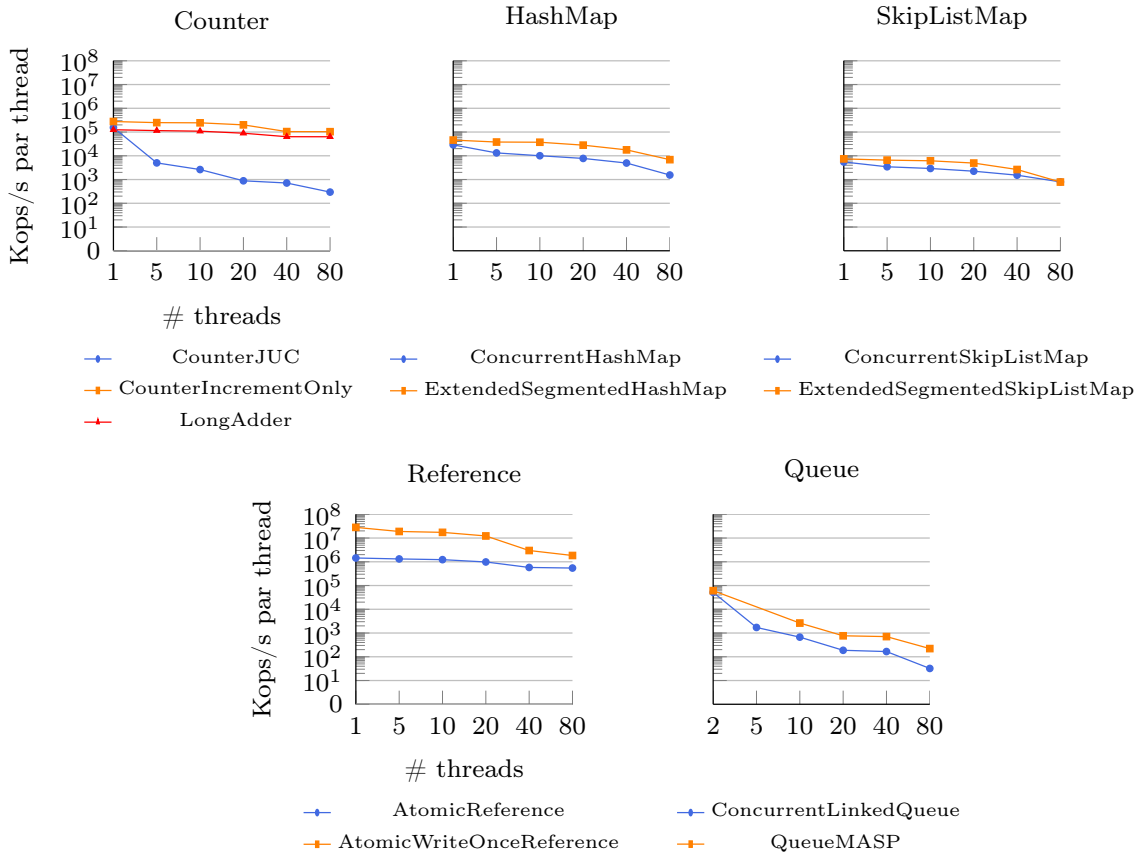

FIGURE 4.1 – Performances des objets dans `DEGO` par rapport à leurs homologues dans `java.util.concurrent` sous forte contention.

tile ne sont pas réordonnées. Cependant, puisque cette barrière est appliquée après une écriture, elle n'est pas utile dans le contexte de ce benchmark. La version ajustée, détaillée dans le Listing 3.1, maintient une copie de la référence en cache, ce qui signifie qu'aucune barrière n'est utilisée lors de la lecture de l'objet, entraînant ainsi une amélioration significative de la performance.

Comme noté dans la Section 4.1, notre file ajustée utilise un mécanisme plus simple pour mettre à jour la tête de file lorsqu'un seul thread exécute des opérations `poll`. Cela explique pourquoi la `QueueMASP` est en moyenne 4.3x fois plus rapide que la `ConcurrentLinkedQueue` dans la Figure 4.1.

4.2.3 Charge de travail mixte

La Figure 4.2 illustre les performances de deux types de *map* partagées : une basée sur les tables de hachage (Désordonnée) et l'autre sur les *skiplist* (Ordonnée). Comme dans les évaluations précédentes, la figure compare les implémentations disponibles dans `DEGO` et dans le *package* `java.util.concurrent`.

La charge de travail que nous appliquons répartit équitablement les mises à jour entre ajout et suppression d'un élément. Ces mises à jour sont faites sur des éléments distincts par thread, tandis que les lectures ne recherchent qu'un seul élément dans la *map*.

Analyse des résultats

Les résultats présentés dans la Figure 4.2 montrent que pour les deux types de *map*, qu'elles soient ajustées ou non, le débit tend à diminuer à mesure que la proportion de mises à jour augmente. Cette diminution est principalement due à la contention entre les threads qui tentent d'accéder aux mêmes ressources mémoire sur la machine.

Globalement, les objets ajustés affichent tout de même une meilleure performance. Cette amélioration provient de leurs implémentations, qui permettent aux threads d'accéder à différents emplacements mémoire, réduisant ainsi la contention. Comme mentionné précédemment, le compteur *cycle_activity.stalls_total* mesure l'activité des threads. En moyenne, l'`ExtendedSegmentedHashMap` affiche une réduction de 30% des cycles avec un thread en attente par rapport à la `ConcurrentHashMap`. Avec l'`ExtendedSegmentedSkipListMap`, la réduction moyenne est de 11% pour ces cycles.

Lorsque le ratio de mise à jour augmente, l'écart de performance entre les implémentations de `DEGO` et celles de JUC tend à se creuser. Par exemple, pour une *map* basée sur une table de hachage, `DEGO` est en moyenne 2.5x fois plus rapide avec 25% de mises à jour, et jusqu'à 4.5x fois plus rapide avec 100% de mises à jour.

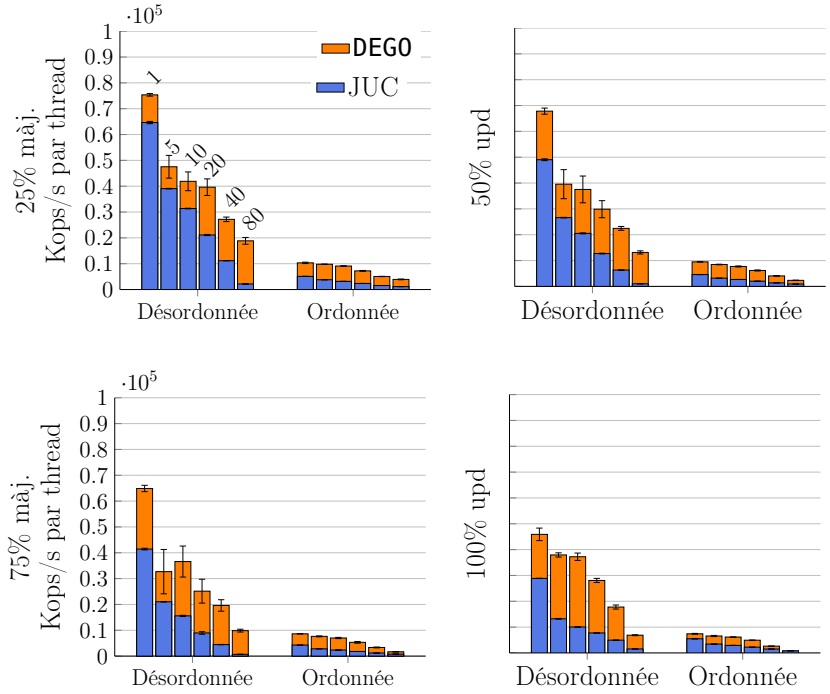

FIGURE 4.2 – Évolution des performances en faisant varier le taux de mise à jour pour une table de hachage (non ordonnée) et une *skiplist* (ordonnée)

.

4.2.4 Taille de la collection

La Figure 4.3 compare la performance de la *map* basé sur une table de hachage dans JUC et dans `DEGO` en fonction du nombre d'éléments stockés dans l'objet. À gauche de la Figure 4.3, les paramètres sont identiques à ceux des expériences précédentes : chaque collection commence avec 16 384 éléments de données, et le nombre maximal d'éléments ajoutés est de 32K éléments. Nous doublons ces valeurs à deux reprises pour observer l'impact de l'augmentation du nombre d'éléments dans l'objet sur sa performance, comme indiqué à droite de la figure.

Analyse des résultats

Cette augmentation entraîne une expansion du tableau de *bins* qui constitue la table de hachage. Chaque fois que la taille du tableau double, les valeurs contenues dans les *bins* sont rééquilibrées pour garantir un accès rapide aux objets. Par

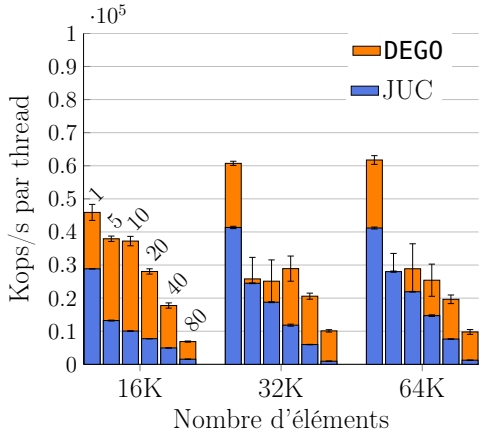

FIGURE 4.3 – Performances d'une table de hachage avec différentes tailles d'ensembles de données (75% de mises à jour)

conséquent, la contention sur un *bin* donné diminue. De plus, un plus grand nombre d'éléments réduit la probabilité que deux threads aient le même hash en même temps, ce qui contribue également à diminuer la contention. Ces facteurs expliquent pourquoi l'écart de performance entre les deux implémentations se réduit quand le nombre d'éléments contenus dans la *map* est élevé.

Résumé

Nous évaluons via des micro-benchmarks les performances des objets ajustés proposés dans `DEGO`. Ces performances sont comparées à celles obtenues avec leurs homologues de `java.util.concurrent`. Différents scénarios sont considérés, sous forte contention, avec un mix d'opérations et quand la taille des données varie. Dans l'ensemble nos résultats indiquent que les objets ajustés sont sensiblement plus rapides.

4.3 Retwis

Dans cette section, nous présentons une évaluation des avantages apportés par `DEGO` dans le cadre d'une application informatique moderne. Plus précisément, nous utilisons un benchmark basé sur Retwis, qui est un clone simplifié de Twitter. Ce benchmark permet de simuler les interactions typiques d'un réseau social.

Ci-après, nous présentons l'application, sa mise en œuvre, comment nous avons modélisé le réseau social, et enfin le benchmark. Plus loin, nous listerons et commenterons les résultats.

4.3.1 Structure de l'application

Notre implémentation de Retwis est multithreadée et elle est développée en Java. Le code source compte environ 1500 lignes de code, ce qui inclut à la fois la gestion des requêtes utilisateurs et les opérations sur les structures de données concurrentes, essentielles à la gestion des interactions en temps réel.

Notre implémentation étend la logique de l'application Retwis [69] afin de mieux évaluer l'efficacité de `DEGO` dans un contexte applicatif de type réseau social. L'application maintient un ensemble d'utilisateurs, où chaque utilisateur peut interagir de manière classique en publiant des messages, en suivant ou en se désabonnant d'autres utilisateurs, ainsi qu'en consultant son fil d'actualité, c'est-à-dire les messages publiés par les personnes qu'il suit. En outre, les utilisateurs peuvent également rejoindre ou quitter des groupes d'intérêt et mettre à jour leur profil personnel.

En interne, l'application repose sur les structures de données listées ci-après : *(i)* `mapFollowers` associe à chaque utilisateur la liste de ses abonnés, *(ii)* la table `mapFollowing` stocke les utilisateurs suivis par un utilisateur donné, *(iii)* la table `mapTimelines` gère le fil d'actualité personnel de chaque utilisateur, *(iv)* le profil de chaque utilisateur est enregistré dans `mapProfiles`, *(v)* `community` conserve les utilisateurs membres d'un groupe d'intérêt spécifique.

Lorsqu'un nouvel utilisateur est ajouté au système, les structures de données correspondantes sont mises à jour. Les abonnés d'un utilisateur, ainsi que les utilisateurs qu'il suit, sont stockés dans des ensembles distincts, qui sont mis à jour lors des opérations de suivi ou de désabonnement. Le fil d'actualité d'un utilisateur est géré sous la forme d'une file d'attente, où chaque nouveau message est inséré. Lorsqu'un utilisateur publie un message, ce dernier est ajouté aux fils d'actualités de ses abonnés. Cependant, pour des raisons de performances, le message est d'abord envoyé aux premiers abonnés, tandis que les autres reçoivent le message

Ajouter un utilisateur	5%
S'abonner/se désabonner d'un utilisateur	5%
Poster un message (tweet)	15%
Voir une timeline	60%
Joindre/quitter un groupe	5%
Mettre à jour un profil	10%

TABLE 4.1 – Charge de travail pour l'application de type réseau social.

de manière asynchrone.[1] Lors de la consultation du fil d'actualité, les messages présents dans la file d'attente sont récupérés, et uniquement les 50 messages les plus récents sont gardés.

Toutes ces structures de données sont conçues pour être accessibles de manière concurrente. Le nombre de threads utilisés par l'application est paramétrable, chaque thread se voyant attribuer une plage d'utilisateurs selon un hachage cohérent pour assurer une distribution équilibrée [38].

L'évaluation se concentre sur trois versions des objets de l'application :

— La première version (JUC) utilise les objets partagés du paquetage standard `java.util.concurrent`.

— La deuxième version (DAP) est de type *share-nothing*, à savoir que chaque thread accède toujours à des objets distincts, offrant ainsi une limite théorique des performances maximales.

— La troisième version repose sur `DEGO`, où les objets sont ajustés de manière optimale : `mapFollowers`, `mapFollowing`, et `mapTimelines` sont des objets à accès commutatifs multi-lecteurs et multi-écrivains (*CWMR*). La file d'attente utilisée pour gérer le fil d'actualité est un objet multi-producteurs et consommateur unique. Enfin, l'ensemble qui stocke les utilisateurs des groupes d'intérêt suit également un modèle *CWMR*.

Ces différentes versions permettent d'analyser les avantages en termes de performance que `DEGO` peut offrir dans le développement d'une application informatique moderne.

1. Cette fonctionnalité secondaire n'est pas implémentée dans la version actuelle du code.

4.3.2 Construction du réseau social

Pour construire le graphe représentant le réseau social, nous avons adopté la méthode décrite par Schweimer et al. [71]. Pour chaque utilisateur, nous générons un degré entrant, un degré sortant, ainsi qu'un degré réciproque. La distribution de ces degrés suit une loi χ^2, ce qui signifie que, dans notre modèle, peu d'utilisateurs suivent un grand nombre de personnes, tandis que la majorité en suivent très peu. De même, peu d'utilisateurs sont massivement suivis ou possèdent de nombreuses connexions mutuelles, reflétant une distribution asymétrique typique des réseaux sociaux. Cette approche produit un graphe où les distributions des degrés entrants et sortants suivent une loi χ^2, une propriété observée dans des réseaux sociaux réels tels que Twitter [60].

Dans l'étude de Schweimer et al. [71], les auteurs proposent une étape additionnelle pour augmenter le coefficient de clustering moyen du graphe, rendant ainsi la structure plus représentative des réseaux sociaux réels, où les connexions sont souvent regroupées en communautés denses. Toutefois, nous avons décidé d'omettre cette étape dans nos expériences, car elle s'est avérée trop coûteuse en termes de temps de calcul pour les échelles que nous avons étudiées. Par exemple, pour un réseau de 10^6 utilisateurs, cette étape supplémentaire aurait nécessité plusieurs centaines de jours de calcul, ce qui n'était pas envisageable dans le cadre de notre évaluation.

La génération du graphe s'effectue en trois phases principales : la génération des utilisateurs, leur assignation aux threads, et la phase de création des connexions (*following phase*). Pendant la première phase, nous attribuons à chaque utilisateur une valeur de probabilité, issue de la loi de puissance, représentant sa probabilité d'être suivi. Cela se traduit par un petit nombre d'utilisateurs ayant une forte chance d'être suivis, tandis que la vaste majorité a une probabilité très faible. Chaque utilisateur se voit alors associé à cette probabilité, et les structures nécessaires à la construction du graphe sont générées. Dans la phase d'assignation aux threads, les utilisateurs sont répartis en sous-ensembles, chacun étant attribué à un thread. À l'image de leur probabilité d'être suivis, l'activité des utilisateurs est également hétérogène : quelques-uns sont très actifs, tandis que la majorité l'est peu. Enfin, lors de la phase de création des connexions, nous construisons le graphe

initial en fonction des degrés associés à chaque utilisateur, finalisant ainsi le réseau social simulé. Cette méthode garantit une représentation fidèle des caractéristiques des réseaux sociaux tout en assurant la faisabilité de la simulation sur de grandes échelles.

4.3.3 Détails sur le benchmark

La charge de travail que nous évaluons ici est un mélange d'opérations inspirées de l'utilisation réelle des réseaux sociaux, tel que présenté dans la Table 4.1. Chaque test est exécuté 10 fois, et chaque exécution durant 20 secondes, avec une phase d'échauffement de 5 secondes.

Les utilisateurs qui effectuent les différentes opérations, ainsi que les utilisateurs qu'ils suivent (ou ne suivent plus), sont sélectionnés à l'avance selon des distributions modélisées par une loi de puissance. Ce choix permet de capturer la réalité où certains utilisateurs sont plus actifs ou plus populaires que d'autres. La distribution des accés peut être ajustée par un paramètre α. Lorsque $\alpha = 1$, la distribution est fortement biaisée, tandis que pour des valeurs de α proches de 0, la distribution devient plus uniforme.

Au cours de l'exécution d'un test, la distribution des abonnés peut changer légèrement, ce qui pourrait affecter les propriétés du réseau social. Pour éviter cela, à chaque fois qu'un utilisateur suit ou se désabonne d'un autre, une opération inverse est appliquée immédiatement après. Cette opération secondaire ne fait pas partie des mesures de performance du test. Il est à noter ici que certaines utilisations du benchmark Retwis n'ont pas pris garde à préserver cet invariant (e.g., [59]).

4.3.4 Analyse des résultats

Scalabilité

La Figure 4.4 présente les résultats obtenus en fonction du nombre d'utilisateurs et du nombre de threads utilisés. Le paramètre α est fixé à 1 dans cette expérience, ce qui correspond à une distribution biaisée des utilisateurs. Les performances rapportées dans cette figure sont comparées à celles de JUC.

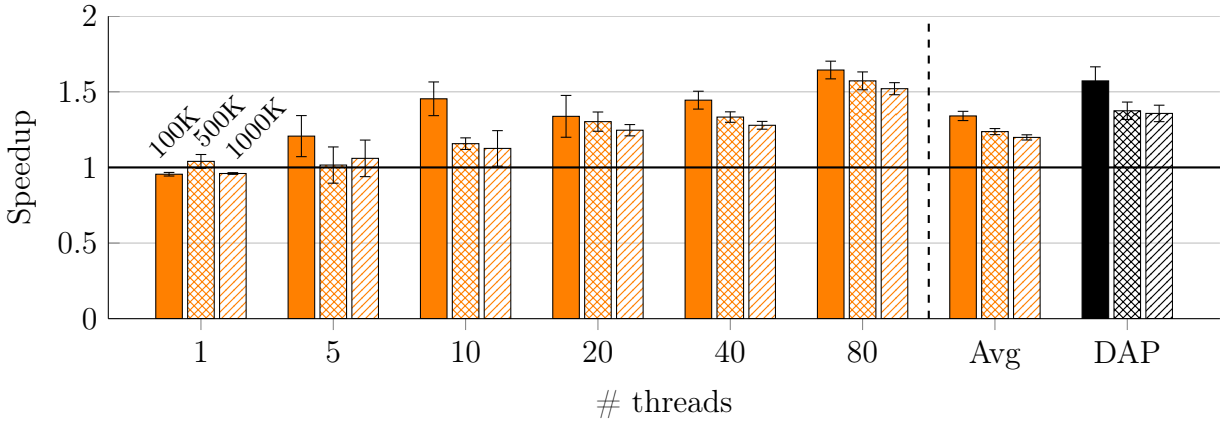

FIGURE 4.4 – Résultats expérimentaux pour l'application de type réseau social. La performance est relative à la mise en œuvre de référence utilisant `java.util.concurrent` (JUC). La colonne la plus à droite (DAP) correspond à une implémentation *share-noting*. Le nombre d'utilisateurs du réseau social varient, passant de 100000, 500000 puis 1000000.

Nous observons que `DEGO` surpasse systématiquement la mise en œuvre de référence JUC, sauf dans le cas où un seul thread est utilisé pour exécuter les tests. Plus précisément, `DEGO` offre des gains de performance variant entre 0.89x et 1.7x par rapport à JUC, le gain maximal étant obtenu avec 10^5 utilisateurs et 80 threads. Ces performances se rapprochent de celles obtenues avec l'implémentation DAP, qui est présentée à droite de la Figure 4.4.

Plusieurs facteurs peuvent influencer les performances dans ce contexte, tels que la taille du cache CPU et celle de l'ensemble des données dans l'application. Cependant, un facteur qui peut limiter les avantages de `DEGO` est l'augmentation de l'empreinte mémoire de l'application due aux métadonnées supplémentaires nécessaires pour gérer les objets ajustés. Dans une version initiale de nos tests, les structures de données qui stockaient les abonnés et les utilisateurs suivis étaient toutes deux ajustées (car ils sont de type *CWSR*). Néanmoins, cet ajustement a provoqué une surcharge mémoire qui annulait les gains de performance résultant de la réduction de la contention.

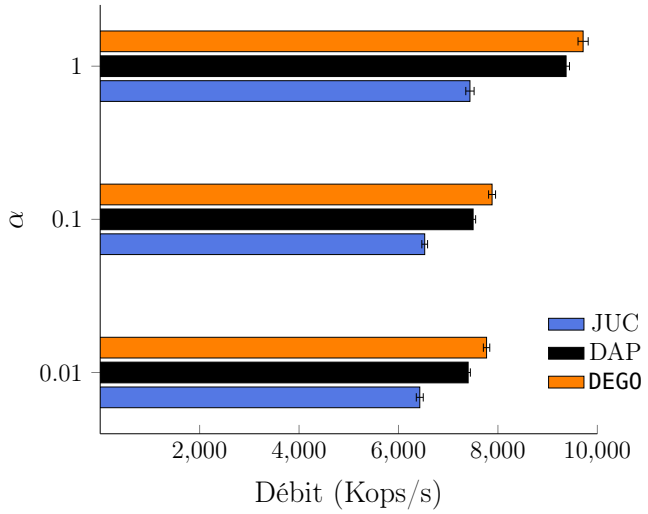

FIGURE 4.5 – Évolution des performances en faisant varier la distribution d'accès de 100000 utilisateurs. On compare les implémentations suivantes : (JUC) une mise en œuvre de référence basée sur `java.util.concurrent`, (`DEGO`) une autre utilisant des objets ajustés, et (DAP) une implémentation de type *share-noting*.

Localité

La Figure 4.5 examine l'influence de la distribution des utilisateurs sur les performances. Il y a 100000 utilisateurs du réseau social dans cette expérience. Lorsque la distribution est biaisée, une forte localité avantage `DEGO`. En effet, dans un tel cas, la contention devient le facteur dominant en termes de performance. À l'inverse, lorsque les utilisateurs sont choisis de manière uniformément aléatoire, l'efficacité du cache du processeur diminue, réduisant ainsi l'écart de performance avec JUC.

Dans cette application, l'ensemble des clés utilisateurs n'est pas complètement disjoints entre les threads : en effet, un utilisateur géré par un thread p peut tout à fait suivre un utilisateur géré par un autre thread q. Toutefois, dans la mise en œuvre DAP, ce n'est pas le cas. Si un utilisateur est géré par un thread p il ne peut suivre que les utilisateurs géré par p. En d'autres termes, les clés sont explicitement non partagées entre les threads. Ceci garantie que les accès soient à parallélisme disjoints et donc l'implémentation *share-nothing*. Ce choix de conception renforce la localité des données pour les objets `DEGO` par rapport aux objets DAP. Ainsi, on peut observer que le taux de *cache-miss* pour `DEGO` est inférieur à celui DAP

(d'environ 3%). Cela explique que DEGO soit plus performant que DAP dans la Figure 4.5. Notons qu'un tel écart se réduit (puis s'inverse) à mesure que le nombre d'utilisateurs diminue, du fait les données finissent par tenir dans le cache.

Résumé

Cette section évalue les performances de DEGO dans le cadre d'une application concrète de type réseau social. Le benchmark utilisé pour cette évaluation s'inspire de Retwis [69]. Les résultats montrent que les objets ajustés de la bibliothèque DEGO permettent d'obtenir des performances supérieures à celles de leurs homologues du JDK. Avec 80 threads exécutant des opérations concurrentes, le débit d'opérations est jusqu'à $1,7$ fois plus élevé.

4.3.5 Conclusion du chapitre

Au cours de ce chapitre, nous avons présenté une bibliothèque d'objets ajustée qui s'appelle DEGO. Cette bibliothèque met en œuvre les principes théoriques que nous avons introduits dans le Chapitre 3. Elle contient des collections partagées (e.g., *set*, *list* et *map*), ainsi que d'autres objets courants comme des compteurs et références. Nous avons présenté les logiques de construction de ces objets partagés.

Dans la seconde partie du chapitre, les objets proposés dans DEGO ont été évalués. Nous utilisons pour ce faire deux types de benchmarks. D'abord des micro-benchmarks proches de la suite Synchrobench proposée par Gramoli [21]. Par ailleurs, nous avons aussi utilisé une application de type réseau social, inspirée de Retwis [69].

Nos résultats montrent que les objets ajustés améliorent la performance des applications concurrentes lorsque la situation est propice Ainsi, avec les micro-benchmarks nous obtenons une accélération de la latence des opérations pouvant aller jusqu'à deux ordres de grandeur par rapport à la mise en œuvre de référence reposant sur le JDK. Dans l'application de type réseau social, le gain maximal est autour de 1.7x.

Chapitre 5

Conclusion

Cette thèse a motivé, présenté et évalué **le principe d'ajustement**. L'idée centrale de ce principe est de spécialiser un objet partagé. Nous avons vu qu'ajuster un objet revient à restreindre son interface (exprimé par un sous-typage) et/ou à rendre asymétrique son utilisation. L'objectif d'un ajustement est l'obtention de meilleures performances dans le cadre d'un usage applicatif spécifique.

Ci-après, nous faisons un état des lieux de ce travail (Section 5.1). Puis, nous verrons les limites de cette approche ainsi que les difficultés que nous avons rencontrées dans sa réalisation (Section 5.2). Enfin, nous clôturerons cette thèse par un aperçu des travaux futurs et questions qu'elle soulève (Section 5.3).

5.1 Résumé

Contexte de motivation

Les infrastructures informatiques modernes reposent sur des machines multiprocesseurs et multicœurs. La conception de programmes parallèles capables de passer à l'échelle sur ces architectures constitue un réel défi pour tout développeur. Pour aider à leurs réalisations, des bibliothèques d'objets partagés existent dans de nombreux langages de programmation. Toutefois, ces bibliothèques sont génériques dans le sens où les abstractions qu'elles proposent ont une interface étendue afin de couvrir leurs nombreux cas d'usage. Le point de départ de cette thèse est la question de savoir s'il est intéressant de spécialiser ces objets afin de

les rendre plus performants.

Analyse des usages

Afin de répondre à la question ci-dessus, nous avons conduit une étude approfondie sur l'utilisation des objets partagés dans les logiciels modernes. Pour ce faire, nous avons réalisé une analyse sur 50 projets de la fondation Apache. Cette analyse a mis en évidence une augmentation progressive de l'utilisation des objets partagés, liée à l'accroissement du parallélisme matériel. Bien que leur proportion reste faible dans les codes source, ces objets sont utilisés dans les fichiers les plus souvent modifiés par les développeurs. Ceci souligne la centralité des questions de concurrence dans les programmes actuels.

Une analyse fine des méthodes utilisées révèle que seule une fraction des fonctionnalités des objets partagés est couramment sollicitée, certaines méthodes se distinguant par leur usage fréquent. Par ailleurs, il apparaît que certaines de ces méthodes sont fréquemment appelées sans que leur valeur de retour soit exploitée. Ces observations suggèrent des possibilités d'optimisation en spécialisant les objets partagés à leurs usages spécifiques dans un programme donné, avec à la clé des gains potentiels en termes de performance. De fait, cette approche existe déjà dans des projets de grande envergure. En effet, certains développeurs experts ont recours à l'usage d'objets ad hoc dans des situations spécifiques.

Canevas théorique

Fort de cette étude, nous avons proposé dans un premier temps de définir cette spécialisation, dite **ajustement**. Pour formaliser et analyser les objets ajustés, nous avons introduit un outil conceptuel : le **graphe d'indistinguabilité**. Ce graphe permet de caractériser la capacité d'un objet partagé à différencier plusieurs entrelacements d'opérations. Il se construit à partir du type de l'objet et des permissions d'accès des différents threads. Nous avons vu que la connexité, la densité ainsi que le nombre de labels par arêtes de ce graphe sont des indicateurs de la capacité de l'objet à passer à l'échelle (scalabilité). En un mot, un graphe dense rend l'objet plus scalable. Ajuster un objet revient à restreindre son interface (exprimé sous la forme d'un sous-typage) et/ou à rendre asymétrique son utilisation

par les threads. Un ajustement se traduit par une augmentation du nombre d'arcs dans les graphes d'indistinguabilité de l'objet. Une fois ce principe d'ajustement défini formellement, nous avons illustré différents types d'ajustements possibles.

Bibliothèque `DEGO` et évaluation

La seconde partie de cette thèse met en pratique les résultats ci-dessus. Dans cette partie, nous présentons la bibliothèque d'objets ajustés `DEGO`. Cette bibliothèque inclut des collections partagées (e.g., *set*, *list* et *map*), ainsi que d'autres objets comme des compteurs, files et références pour de multiples cas d'usage. Elle couvre autour de 11000 lignes de Java.

La thèse se poursuit avec une démonstration de l'intérêt pratique des objets de `DEGO`. Pour ce faire, nous utilisons deux types de benchmarks. D'abord, les objets ajustés sont validés au travers d'une série de micro-benchmarks. Ces benchmarks, proches de la suite Synchrobench [21], visent à observer leurs comportements sous différentes charges de travail. Ainsi nous analysons ces objets en cas de forte contention, avec un workload mixte, ou encore en variant la taille du *working set*. Les performances obtenues sont comparées à celles des objets partagés du Java Development Kit (JDK), que l'on trouve dans la bibliothèque `java.util.concurrent`. Les résultats obtenus montrent que les objets ajustés surpassent systématiquement leurs homologues du JDK.

Nous évaluons ensuite les performances des objets ajustés disponibles dans `DEGO` pour le développement d'une application moderne. Cette application a été entièrement écrite par nos soins afin d'effectuer une analyse fine des points de contention dans un cadre réaliste. Elle simule un réseau social similaire à Twitter, permettant ainsi de tester les objets ajustés dans un environnement concurrent à grande échelle. Cette application est inspirée du benchmark Retwis [69]. Les résultats montrent de nouveau que les objets ajustés offrent des performances supérieures aux objets du JDK.

Dans l'ensemble, ces expériences soulignent la pertinence de la bibliothèque d'objets ajustés `DEGO`, notamment pour des contextes applicatifs où une gestion efficace de la concurrence est déterminante.

5.2 Difficultés rencontrées et limites

Cette thèse développe une approche générale visant à améliorer les performances des programmes parallèles. Elle couvre à la fois des aspects théoriques et pratiques. Une des difficultés rencontrée est le passage de la théorie vers la pratique. Ainsi, bien que certains objets ajustés semblent prometteurs sur le papier, il y a de nombreux défis d'implémentation. Ci-après, nous soulignons certaines de ces difficultés, puis nous discutons des limites de notre approche.

Une mise en œuvre complexe

La première difficulté est que les performances initialement observées ne sont pas toujours conformes aux attentes. Par exemple, ce fut le cas avec la classe `QueueMASP`. Pour rappel, la méthode `poll` de cette classe ne nécessite aucun mécanisme de synchronisation explicite (a contrario de `ConcurrentLinkedQueue`). Or, nous avons constaté au départ que l'ajout d'éléments dans la file ajustée était plus lent, alors même que le code exécuté est identique. Après analyse, nous avons découvert que cette différence était causée par un phénomène de *false sharing*. En effet, comme l'opération `poll` est plus efficace dans la file ajustée, le pointeur de la tête de la file est mis à jour plus fréquemment, générant ainsi des conflits de cache entre threads. Pour résoudre ce problème dans la version actuelle, nous utilisons l'annotation `@Contended`, qui permet de faire du *padding* mémoire, réservant ainsi la ligne de cache pour cette variable. Une fois cette modification apportée, les performances obtenues sont nettement meilleures, comme vu en Section 4.2.

Un autre défi rencontré concernait la gestion de l'empreinte mémoire. Par exemple, bien que le *padding* ait résolu le problème de *false sharing*, il a également augmenté l'empreinte mémoire de la file ajustée. Cela a eu pour effet d'entraîner des lectures plus fréquentes depuis la mémoire principale, au détriment du cache. Avec un grand nombre d'utilisateurs dans l'application de réseau social, ce surcoût en écriture a nullifié les gains liés à l'ajustement. Un problème similaire s'est aussi produit lors de l'utilisation d'objets ajustés basés sur une segmentation. Lorsque le nombre de threads et d'utilisateurs dans le système augmentait, le nombre de segments créés devenait de plus en plus important. Ceci a eu pour effet d'ampli-

fier les coûts liés à la gestion mémoire. Or, cette amplification des coûts n'a été apparente que lorsque nous avons effectué des scénarios à grande échelle.

Limites de notre approche

Les difficultés ci-dessus illustrent certaines des limites du principe d'ajustement. En premier lieu, l'existence d'une construction théorique pour implémenter un objet avec peu (voir pas) de conflits n'assure pas en pratique de son utilité avec le matériel actuel. C'est par exemple le cas pour certains des résultats énoncés dans le Chapitre 3 qui repose sur des constructions génériques (dites univerelles). Un autre point est que le modèle théorique néglige la hierarchie mémoire et son impact sur les performances. En particulier, un ajustement peut conduire à une amplification mémoire, qui se traduit par des *cache hits* moins fréquents. Avec les architectures actuelles, il y a un ordre de grandeur entre le temps d'accès au cache CPU et la mémoire principale. Ainsi, comme l'illustrent les exemples précédents, cette amplification mémoire peut annuler tout gain lié à une moindre contention avec les objets ajustés.

5.3 Travaux futurs

Cette étude ouvre des perspectives prometteuses, tant sur le plan théorique que pratique, pour approfondir et élargir les travaux présentés. Nous en élaborons certaines en particulier ci-dessous.

Mieux comprendre les usages

Notre étude de l'usage des objets partagés dans les programmes parallèles a été réalisée de manière statique, sur des bases de code. En allant plus loin, on pourrait envisager une analyse dynamique des applications. Cette dernière permettrait de capturer l'usage précis des objets au cours de l'exécution, et voir lesquels d'entre eux sont clés pour la performance.

Mieux comprendre la scalabilité

Sur le plan théorique, il serait pertinent d'explorer plus en détail la relation entre la structure du graphe d'indistinguabilité et la capacité d'un objet à supporter une mise à l'échelle efficace. Une compréhension plus fine de cette relation pourrait offrir des critères plus précis pour caractériser les objets ajustés en termes de scalabilité. Conjointement, d'un point de vue pratique, une avancée notable consisterait à automatiser la génération du graphe d'indistinguabilité. Ceci facilitant la comparaison entre objets pour déterminer lequel offre la meilleure scalabilité dans un contexte donné.

Étendre les critères de correction

L'étude actuelle porte essentiellement sur des objets linéarisables et sans-attente. Un prolongement naturel consisterait à étendre nos résultats à des modèles de cohérence plus faibles et/ou à des conditions de progrès moins strictes. Cela permettrait d'élargir l'applicabilité des concepts développés à un éventail plus large d'objets partagés et de programmes.

Injecter automatiquement des objets ajustés

Enfin, on pourrait aussi travailler sur la substitution automatique d'objets partagés par des versions ajustées. Cela permettrait de manière simple d'optimiser un programme parallèle, rendant notre approche plus accessible aux développeurs.

Bibliographie

[1] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability : Relaxed consistency for improved concurrency. In C. Lu, T. Masuzawa, and M. Mosbah, editors, *Principles of Distributed Systems*, pages 395–410, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17653-1.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory : definitions, implementation, and programming. *Distrib. Comput.*, 9(1) :37–49, mar 1995. ISSN 0178-2770. doi : 10.1007/BF01784241. URL https://doi.org/10.1007/BF01784241.

[3] J. Aspnes, H. Attiya, and K. Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1), mar 2012. ISSN 0004-5411. doi : 10.1145/2108242.2108244. URL https://doi.org/10.1145/2108242.2108244.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1) :124–142, jan 1995. ISSN 0004-5411. doi : 10.1145/200836.200869. URL https://doi.org/10.1145/200836.200869.

[5] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 69–78, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586069. doi : 10.1145/1583991.1584015. URL https://doi.org/10.1145/1583991.1584015.

[6] A. Bieniusa. Antidotedb, 2017. URL https://www.antidotedb.eu/.

[7] G. E. Blelloch and B. M. Maggs. *Parallel algorithms*, page 25. Chapman & Hall/CRC, 2 edition, 2010. ISBN 9781584888208.

[8] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. Numa policies and their relation to memory architecture. *ACM SIGOPS Operating Systems Review*, 25(Special Issue) :212–221, 1991.

[9] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense ! (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 363–372, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916549. doi : 10.1145/197917.198126. URL https://doi.org/10.1145/197917.198126.

[10] M. S. M. Bouksiaa, F. Trahay, A. Lescouet, G. Voron, R. Dulong, A. Guermouche, E. Brunet, and G. Thomas. Using differential execution analysis to identify thread interference. *IEEE Transactions on Parallel and Distributed Systems*, 30(12) :2866–2878, 2019. doi : 10.1109/TPDS.2019.2927481.

[11] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule : Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4), Jan. 2015. ISSN 0734-2071. doi : 10.1145/2699681. URL https://doi.org/10.1145/2699681.

[12] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 9780080573076.

[13] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 33–48, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi : 10.1145/2517349.2522714. URL https://doi.org/10.1145/2517349.2522714.

[14] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, page 37–48, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581139454. doi : 10.1145/1029873.1029879. URL https://doi.org/10.1145/1029873.1029879.

[15] E. W. Dijkstra. *Cooperating sequential processes*, page 65–138. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 0387954015.

[16] P. Fatourou, F. Fich, and E. Ruppert. A tight time lower bound for space-optimal implementations of multi-writer snapshots. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 259–268, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136749. doi : 10.1145/780542.780582. URL https://doi.org/10.1145/780542.780582.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, Apr. 1985. ISSN 0004-5411. doi : 10.1145/3149.214121. URL http://doi.acm.org/10.1145/3149.214121.

[18] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960, 1972. doi : 10.1109/TC.1972.5009071.

[19] M. Forsell, S. Nikula, J. Roivainen, V. Leppänen, and J. Träff. Performance and programmability comparison of the thick control flow architecture and current multicore processors. *The Journal of Supercomputing*, 78 :1–32, 02 2022. doi : 10.1007/s11227-021-03985-0.

[20] M. P. Forum. Mpi : A message-passing interface standard. Technical report, -, USA, 1994.

[21] V. Gramoli. More than you ever wanted to know about synchronization : synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming, PPoPP 2015, page 1–10, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332057. doi : 10.1145/2688500.2688501. URL https://doi.org/10.1145/2688500.2688501.

[22] H. Guiroux, R. Lachaize, and V. Quéma. Multicore locks : the case is not closed yet. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 649–662, USA, 2016. USENIX Association. ISBN 9781931971300.

[23] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

[24] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13 (1) :124–149, Jan. 1991. ISSN 0164-0925. doi : 10.1145/114005.102808. URL http://doi.acm.org/10.1145/114005.102808.

[25] M. Herlihy. Asynchronous consensus impossibility. In *Encyclopedia of Algorithms*, pages 152–155. Springer, Boston, MA, 2016. doi : 10.1007/978-1-4939-2864-4_36. URL https://doi.org/10.1007/978-1-4939-2864-4_36.

[26] M. Herlihy and J. E. B. Moss. Transactional memory : architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2) : 289–300, may 1993. ISSN 0163-5964. doi : 10.1145/173682.165164. URL https://doi.org/10.1145/173682.165164.

[27] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123973375.

[28] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123973375.

[29] M. P. Herlihy and J. M. Wing. Linearizability : a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3) :463–492, jul 1990. ISSN 0164-0925. doi : 10.1145/78969.78972. URL https://doi.org/10.1145/78969.78972.

[30] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[31] M. Hill. A case for direct-mapped caches. *Computer*, 21(12) :25–40, 1988. doi : 10.1109/2.16187.

[32] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, oct 1969. ISSN 0001-0782. doi : 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

[33] C. A. R. Hoare. Monitors : an operating system structuring concept. *Commun. ACM*, 17(10) :549–557, Oct. 1974. ISSN 0001-0782. doi : 10.1145/355620.361161. URL https://doi.org/10.1145/355620.361161.

[34] T. Ishida, Y. Sasaki, and Y. Fukuhara. Use of procedural programming languages for controlling production systems. In *[1991] Proceedings. The Seventh IEEE Conference on Artificial Intelligence Application*, volume i, pages 71–75, 1991. doi : 10.1109/CAIA.1991.120848.

[35] A. Israeli and A. Shirazi. The time complexity of updating snapshot memories. *Inf. Process. Lett.*, 65(1) :33–40, Jan. 1998. ISSN 0020-0190. doi : 10.1016/S0020-0190(97)00189-0. URL https://doi.org/10.1016/S0020-0190(97)00189-0.

[36] S. K. J. Held, J. Bautista. From a few cores to many – a tera-scale computing research overview. *Intel White Paper*, 2006. URL https://www.intel.sa/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-tera-scale-research-paper.pdf.

[37] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations (preliminary version). In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 257–266, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897918002. doi : 10.1145/248052.248105. URL https://doi.org/10.1145/248052.248105.

[38] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing*, STOC, 1997.

[39] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. *ACM SIGARCH Computer Architecture News*, 13(3) :276–283, 1985.

[40] P. Khanchandani, J. Schäppi, Y. Wang, and R. Wattenhofer. On consensus number 1 objects. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 875–882, 2021. doi : 10.1109/ICPADS53394.2021.00115.

[41] W. Kim and F. Lochovsky. *Object-oriented Concepts, Databases and Applications*. ACM Press frontier series. ACM Press, 1989. ISBN 9780201144109. URL https://books.google.bg/books?id=MLImAAAAMAAJ.

[42] Kogan and Herlihy. The future(s) of shared data structures. In M. M. Halldórsson and S. Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 30–39. ACM, 2014. doi : 10.1145/2611462.2611496. URL https://doi.org/10.1145/2611462.2611496.

[43] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 19–30, New York, NY, USA, 2010. Association for Computing Machinery. ISBN

9781605584799. doi : 10.1145/1706299.1706304. URL https://doi.org/10.1145/1706299.1706304.

[44] N. Koval, A. Fedorov, M. Sokolova, D. Tsitelov, and D. Alistarh. Lincheck : A practical framework for testing concurrent data structures on jvm. In *Computer Aided Verification : 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part I*, page 156–169, Berlin, Heidelberg, 2023. Springer-Verlag. ISBN 978-3-031-37705-1. doi : 10.1007/978-3-031-37706-8_8. URL https://doi.org/10.1007/978-3-031-37706-8_8.

[45] P. Kuznetsov. *Synchronization using failure detectors*. PhD thesis, EPFL, 01 2005.

[46] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9) :690–691, 1979. doi : 10.1109/TC.1979.1675439.

[47] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9) :690–691, Sept. 1979. ISSN 0018-9340. doi : 10.1109/TC.1979.1675439. URL https://doi.org/10.1109/TC.1979.1675439.

[48] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2) : 133–169, may 1998. ISSN 0734-2071. doi : 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

[49] L. Leslie. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978. ISSN 0001-0782. doi : 10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563.

[50] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[51] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA : A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[52] LinkedIn. Concurrentli, 2017. URL https://github.com/linkedin/concurrentli.

[53] R. J. Lipton. Reduction : a method of proving properties of parallel programs. *Commun. ACM*, 18(12) :717–721, dec 1975. ISSN 0001-0782. doi : 10.1145/361227.361234. URL https://doi.org/10.1145/361227.361234.

[54] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16 :1811–1841, 1994.

[55] J. D. C. Little and S. C. Graves. *Little's Law*, pages 81–100. Springer US, Boston, MA, 2008. ISBN 978-0-387-73699-0. doi : 10.1007/978-0-387-73699-0_5. URL https://doi.org/10.1007/978-0-387-73699-0_5.

[56] G. Liu and X. Liu. The complexity of weak consistency. In J. Chen and P. Lu, editors, *Frontiers in Algorithmics*, pages 224–237, Cham, 2018. Springer International Publishing. ISBN 978-3-319-78455-7.

[57] G. Liu and X. Liu. The complexity of weak consistency. In J. Chen and P. Lu, editors, *Frontiers in Algorithmics*, pages 224–237, Cham, 2018. Springer International Publishing. ISBN 978-3-319-78455-7.

[58] D. Michel ;, S. Christoph ;, and B. F. A. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2) :9–21, feb 1988. ISSN 0018-9162. doi : 10.1109/2.15. URL https://doi.org/10.1109/2.15.

[59] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[60] S. A. Myers, A. Sharma, P. Gupta, and J. Lin. Information network or social network ? the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, page 493–498, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327459. doi : 10.1145/2567948.2576939. URL `https://doi.org/10.1145/2567948.2576939`.

[61] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[62] J. K. P.K. Veenstra, F.P.M. Beenker. Testing of random access memories : theory and practice. *IEE Proceedings G (Electronic Circuits and Systems)*, 135 :24–28(4), February 1988. ISSN 0143-7089. URL `https://digital-library.theiet.org/content/journals/10.1049/ip-g-1.1988.0004`.

[63] C. Ribeiro, M. Castro, V. Marangozova-Martin, J.-F. Méhaut, H. Freitas, and C. Martins. Investigating the impact of cpu and memory affinity on multi-core platforms : A case study of numerical scientific multithreaded benchmarks. In -, 11 2011.

[64] C. Ribeiro, M. Castro, V. Marangozova-Martin, J.-F. Méhaut, H. Freitas, and C. Martins. Evaluating cpu and memory affinity for numerical scientific multithreaded benchmarks on multi-cores. *IADIS International Journal on Computer Science and Information Systems*, 7 :79–93, 11 2012.

[65] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *2009 21st International Symposium on Computer Architecture and High Performance Computing*, pages 59–66, 2009. doi : 10.1109/SBAC-PAD.2009.16.

[66] A. Rinberg and I. Keidar. Intermediate value linearizability : A quantitative correctness criterion, 2020.

[67] E. Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4) :1156–1168, 2000. doi : 10.1137/S0097539797329439. URL https://doi.org/10.1137/S0097539797329439.

[68] S. Sanfilippo. Redis, 2009. URL https://redis.io/.

[69] S. Sanfilippo. http ://retwis.antirez.com, 2009. URL https://github.com/antirez/retwis.

[70] B. Schling. *The Boost C++ Libraries*. XML Press, 2011. ISBN 0982219199.

[71] C. Schweimer, C. Gfrerer, F. Lugstein, D. Pape, J. A. Velimsky, R. Elsässer, and B. C. Geiger. Generating simple directed social network graphs for information spreading. In *Proceedings of the ACM Web Conference 2022*. ACM, apr 2022. doi : 10.1145/3485447.3512194.

[72] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9) : 1175–1185, Sept. 1990. ISSN 0018-9340. doi : 10.1109/12.57058. URL https://doi.org/10.1109/12.57058.

[73] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24550-3.

[74] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5) :172–182, dec 1995. ISSN 0163-5980. doi : 10.1145/224057.224070. URL https://doi.org/10.1145/224057.224070.

[75] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1) :40–44, jan 2009. ISSN 0001-0782. doi : 10.1145/1435417.1435432. URL https://doi.org/10.1145/1435417.1435432.

[76] W. E. Weihl. Commutativity-based concurrency control for abstract data types. In *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*, page 205–214, Washington, DC, USA, 1988. IEEE Computer Society Press. ISBN 0818608420.

[77] A. W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th annual international symposium on Computer architecture*, pages 244–252, 1987.

[78] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna : A kvs for any scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018. doi : 10.1109/ICDE.2018.00044.

[79] J. Zhang. Performance comparative analysis on garbage first garbage collector and z garbage collector. In *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, pages 733–740, 2021. doi : 10.1109/ICFTIC54370.2021.9647167.

Annexe A

Modèle du système

Nous considérons le modèle de calcul distribué standard à mémoire partagée [24]. Ci-après, nous introduisons les éléments de ce modèle de calcul, ainsi que des notions clés pour énoncer nos résultats.

Objet

Un objet est défini par un *type de données*. Le type de données modélise les états possibles de l'objet, les opérations permettant d'y accéder, ainsi que les valeurs de réponse associées à ces opérations. Formellement, un type de données est un automate $A = (\mathcal{S}, s_0, \mathcal{C}, \mathcal{V}, \tau)$ où $\mathcal{S}$ représente l'ensemble des états de A, $s_0 \in \mathcal{S}$ son état initial, $\mathcal{C}$ les opérations de A, $\mathcal{V}$ les réponses, et $\tau : \mathcal{S} \times \mathcal{C} \to \mathcal{S} \times \mathcal{V}$ définit la relation de transition.

Une opération c est dite *totale* si $\mathcal{S} \times c$ est dans le domaine de τ. L'opération c est *déterministe* si la restriction de τ à $\mathcal{S} \times c$ est une fonction. Dans ce qui suit, nous supposons que toutes les opérations sont totales et déterministes.

Les champs $.\mathsf{st}$ et $.\mathsf{val}$ extraient respectivement les composantes d'état et de valeur de réponse d'une opération, c'est-à-dire que pour un état s et une opération c, $\tau(s, c) = (\tau(s, c).\mathsf{st}, \tau(s, c).\mathsf{val})$. La fonction τ^+ est définie par l'application répétée de τ, c'est-à-dire que pour une séquence d'opérations $\sigma = \langle c_1, \ldots, c_{n \geq 1} \rangle$ et

un état s, nous avons :

$$\tau^+(s,\sigma) \triangleq \begin{cases} \tau(s,c_1) & \text{si } n=1, \\ \tau^+(\tau(s,c_1).\mathsf{st}, \langle c_2,\ldots,c_n\rangle) & \text{sinon.} \end{cases}$$

Logique de Hoare

Nous utilisons les notations de Hoare pour spécifier les objets. Une opération c est définie comme un triplet $[P]c[Q]$, où P et Q sont respectivement les préconditions et postconditions de c, c'est-à-dire des prédicats sur l'état de l'objet. Lorsque c est exécutée dans un état qui satisfait P et qu'à la fin de l'exécution de c, Q est satisfaite, le triplet $[P]c[Q]$ est vrai. En revanche, lorsque c est appliquée à un état qui ne satisfait pas P (ou Q), alors l'opération c n'altère pas l'objet et retourne la valeur spéciale $\perp$.

Ci-dessous, nous utilisons ces notations pour spécifier un compteur avec les opérations `increment`, `get` et `reset`. Comme d'habitude, s représente l'état de l'objet et s' est le nouvel état après l'application de l'opération. Nous notons $\mathbf{r}$ la valeur de retour de l'opération.

- $[true]\ inc()\ [s' = s+1 \wedge \mathbf{r} = s']$
- $[true]\ get()\ [\mathbf{r} = s]$
- $[true]\ reset()\ [s' = 0]$

Principe de substitution de Liskov

Un type de données S est un *sous-type* de T lorsque les conditions suivantes sont remplies. **(1)** Il existe une fonction d'abstraction qui mappe un état (valide) du sous-type à un état du super-type ; **(2)** S préserve les opérations de T. Cela signifie que si $[P]c_S[Q]$ de S correspond à l'opération $[P']c_T[Q']$ de T, alors il est vrai que : **(2a)** Soit c_S et c_T ont toutes les deux un résultat, soit aucune des deux n'en a. **(2b)** *règle de précondition* : c_S peut être appelée au moins dans tout état requis par c_T ($P' \Rightarrow P$). **(2c)** *règle de postcondition* : la postcondition de c_S est plus forte que la postcondition de c_T ($Q \Rightarrow Q'$). De plus, **(3)** une règle concernant la spécification de T portant sur ses propriétés historiques est maintenue avec S. Dans [54], Liskov et Wing définissent un principe de substitution qui permet de

remplacer un type par un autre. Ce principe exige que si une propriété $\mathcal{P}(O)$ est vraie pour tout objet O de type T, alors $\mathcal{P}(O')$ est vraie pour tout objet O' de type S.

Histoires

Nous considérons un modèle à temps global ainsi qu'un ensemble fini de processus, appelés *threads*. Chaque thread peut subir un arrêt (crash) dans son calcul. Une histoire est une séquence d'invocations et de réponses d'opérations par les threads sur un ou plusieurs objets. Nous notons $c \rightsquigarrow_h d$ lorsque l'opération c précède d dans l'histoire h. Cet ordre est appelé *causalité* (ou « précédence »). Les opérations non reliées par $\rightsquigarrow_H$ sont dites *concurrentes*. Dixit Herlihy [24], les histoires possèdent diverses propriétés selon la manière dont les invocations et les réponses sont entrelacées. Pour être complet, nous rappelons ces propriétés ci-après : *(i)* Une histoire h est *complète* si chaque invocation a une réponse correspondante. *(ii)* Une histoire *séquentielle* h est une séquence non entrelacée d'invocations et de réponses correspondantes, possiblement terminée par une invocation sans réponse. Lorsqu'une histoire h n'est pas séquentielle, nous disons qu'elle est *concurrente*. *(iii)* Une histoire h est *bien formée* si (i) la restriction $h|p$ est séquentielle pour chaque processus client p, (ii) chaque opération c est invoquée au plus une fois dans h, et (iii) pour chaque réponse $\mathsf{res}_i(c, v)$, il existe une invocation $\mathsf{inv}_i(c)$ qui la précède dans h. *(iv)* Une histoire bien formée h est *légale* si, pour chaque objet O, $h|O$ est à la fois complète et séquentielle, et en notant $\langle c_1, \ldots, c_{n\geq 1}\rangle$ la séquence des opérations apparaissant dans $h|O$, si une valeur de réponse apparaît pour une opération c_k dans $h|O$, elle est égale à $\tau^+(s_0, \langle c_1, \ldots, c_k\rangle).\mathsf{val}$.

Linéarisabilité

Deux histoires h et h' sont *équivalents* si elles contiennent le même ensemble d'événements. Soit un objet O et une histoire h de O, h est *linéarisable* [24] si elle peut être étendue (en ajoutant zéro ou plusieurs réponses) pour former une histoire complète h' équivalente à une histoire légale et séquentielle l de S tel que $<_{h'} \subseteq <_l$. Dans un tel cas, l'histoire l est appelée une *linéarisation* de h. Les histoires d'un objet O sont toutes les histoires linéarisables qui peuvent être construites avec les

opérations de S.

Progrès sans attente (Wait-freedom)

Dans un système distribué, il est souhaitable qu'un thread puisse terminer ses tâches indépendamment de l'activité des autres threads. Cette propriété est appelée le progrès *sans-attente* (en anglais, *wait-freedom*).

Tâche distribuée

Notons n le nombre de threads dans le système. Une tâche distribuée Δ est définie par un ensemble $\mathcal{I}$ de n-vecteurs d'entrées, un ensemble $\mathcal{O}$ de n-vecteurs de sorties et une fonction Δ de $\mathcal{I}$ vers $2^{\mathcal{O}}$. Si la valeur d'entrée d'un thread p dans $I \in \mathcal{I}$ est $\perp$, nous disons que p ne participe pas au vecteur d'entrée I. De même, si $O[p]$ vaut $\perp$, le thread p ne décide pas dans O. Pour chaque tâche distribuée $\Delta = (\Delta, \mathcal{I}, \mathcal{O})$, nous exigeons que : *(i)* un thread puisse ne pas décider $((\forall p : O'[p] \in \{O[p], \perp\} \wedge (I, O) \in \Delta) \Rightarrow (I, O') \in \Delta)$, et que *(ii)* un thread qui ne participe pas ne décide pas $((I[p] = \perp \wedge (I, O) \in \Delta) \Rightarrow O[p] = \perp)$.

Annexe B

Preuves

B.1 Graphe d'indistinguabilite

Lemma 1. *Considérons un objet partagé O de type T, un état s de T, et un ensemble B d'opérations. Si c est labellisant dans $\mathcal{G}_T(B, s)$, alors pour toute permutation $x \in B$, la valeur de la réponse de c dans x correspond à la réponse obtenue en appliquant c à l'état s.*

Démonstration. Considérons une permutation x de B, et soit v la valeur de la réponse de c dans x. Prenons également y, une permutation de B qui commence par c. Puisque c est labellisant dans $\mathcal{G}_T(B, s)$, il existe un arc (x, y) dans $\mathcal{G}_T(B, s)$ avec un label contenant c. Par conséquent, la réponse de c dans x est identique à celle dans y. Étant donné que c est la première opération dans y, sa réponse est $\tau(s, c).\mathsf{val}$. □

Theorem 1. *Considérons un type de données lisible $T \notin CN_1$. Alors, $CN(T) = \max\{k : \exists l \geq 2 \cdot T \in \mathcal{D}(k, l)\}$*

Démonstration. Le théorème est démontré en deux étapes successives.

Tout d'abord, nous montrons que si un type T peut résoudre consensus parmi k threads, alors il doit exister un état s et un ensemble d'opérations B, tel que $|B| = k$, et de sorte que le graphe de conflits $\mathcal{G}(B, s)$ possède au moins deux composantes connexes. Cela implique que le nombre de consensus de T est inférieur ou égale à $\max\{k : \exists l \geq 2 \cdot T \in \mathcal{D}(k, l)\}$.

Ensuite, nous prouvons que le consensus binaire peut être résolu parmi n threads, sous l'hypothèse que, pour un ensemble d'opérations B avec $|B| = n$ et un état s, $\mathcal{G}(B, s)$ contient au moins deux composantes connexes. Par conséquent, le nombre de consensus de T est supérieur ou égale à $\max\{k : \exists k \geq 2 \cdot T \in \mathcal{D}(k, l)\}$.

($\leq$)

Soit un algorithme $\mathcal{A}$ résolvant le consensus binaire entre n threads, utilisant un nombre quelconque d'objets de type T et de registres. Notre raisonnement s'appuie sur l'argument classique reposant sur la valence introduit dans FLP par Fischer et al. [17]. En résumé, ce raisonnement se déroule comme suit : nous considérons toutes les exécutions de $\mathcal{A}$. Dans une exécution, les états collectifs des threads et de la mémoire partagée à un instant donné sont appelés une *configuration*. Les configurations de toutes les exécutions de $\mathcal{A}$ sont connectées pour former un graphe orienté. Il existe une arête e étiquetée avec p entre les configurations $\mathcal{C}$ et $\mathcal{D}$ lorsque la prochaine opération du thread p dans $\mathcal{C}$ mène à la configuration $\mathcal{D}$. Si c est la prochaine opération appelée par un thread dans la configuration $\mathcal{C}$, alors $\mathcal{C}.c$ est la configuration atteinte après son application. Une configuration $\mathcal{C}$ est 0-valente (respectivement 1-valente) lorsqu'un processus peut décider 0 (resp. 1) dans une configuration succédant à $\mathcal{C}$. La configuration $\mathcal{C}$ est *monovalente* si elle est soit 0-valente soit 1-valente ; sinon, la configuration est dite *bivalente*. $\mathcal{C}$ est *critique* lorsqu'elle est bivalente et que toute configuration immédiatement après $\mathcal{C}$ est monovalente.

En appliquant un raisonnement classique [25], on peut montrer qu'il doit exister une configuration critique $\mathcal{C}$. De plus, dans la configuration $\mathcal{C}$, tous les threads sont sur le point d'accéder au même objet, disons O, de type T. Soit s l'état de l'objet O dans $\mathcal{C}$. Notons B le multiensemble des n opérations que les threads sont sur le point d'exécuter.

Pour $u \in \{0, 1\}$, il existe $x_u \in \mathit{perm}(B)$ tel que pour tout z non vide, $z \sqsubseteq x_u$, z mène à une configuration u-valente depuis $\mathcal{C}$. En effet, étant donné que $\mathcal{C}$ est bivalente, il existe deux opérations $c_0, c_1 \in B$ amenant s respectivement à une configuration 0-valente et 1-valente. Ainsi, toute permutation de B débutant par c_0 (resp., c_1) mène à une configuration 0-valente (resp., 1-valente). Nous choisissons

x_0 et x_1 parmi ces permutations.

Ensuite, nous observons que si $\mathcal{G}(B, s)$ a une seule composante connexe, alors x_0 et x_1 ont la même valence. Si $[x_0] = [x_1]$, alors il existe des permutations successives $y_1, \ldots, y_{k\geq 2}$ telles que *(i)* $y_1 = x_0$ et $y_k = x_1$, et *(ii)* pour chaque $i \in [1, k]$, $y_i \sim c_i, sy_{i+1}$ pour une certaine opération c_i. Comme $y_i \sim c_i, sy_{i+1}$, l'opération c_i retourne la même valeur de réponse dans y_i et y_{i+1}. De plus, il existe un état commun après c_i dans y_i et y_{i+1}. Soient z_i et z_{i+1} les préfixes correspondants, et soit p le thread sur le point d'exécuter c_i depuis la configuration $\mathcal{C}$. Tant $\mathcal{D} = \mathcal{C}.z_i$ que $\mathcal{D} = \mathcal{C}.z_{i+1}$ sont indistinguables pour p. En conséquence, toute exécution en solo de p commençant depuis $\mathcal{D}$ est également applicable à $\mathcal{D}'$ (et vice-versa). Ainsi, elles ont la même valence. Par induction, nous en déduisons que x_0 et x_1 ont la même valence, ce qui constitue une contradiction.

($\geq$)

Le résultat est obtenu par des réductions successives. Nous nous appuyons sur deux variantes du problème de consensus binaire, à savoir le consensus faible et le consensus par équipe. Dans le consensus faible, la condition de terminaison est remplacée par une propriété plus faible : il existe une exécution où la valeur 0 est décidée et une autre où la valeur 1 est décidée. Le consensus par équipe divise les threads en deux équipes (connues à l'avance). La propriété d'accord du consensus est alors remplacée par : si tous les threads d'une équipe ont la même valeur d'entrée, alors aucun thread ne décide d'une valeur différente. À partir du consensus faible, on peut obtenir le consensus par équipe, puis à partir du consensus par équipe, le consensus (binaire) lui-même. Ces deux réductions sont des résultats bien établis dans la littérature [45]. Nous montrons que sous l'hypothèse de l'existence de deux composantes, le consensus faible est solvable.

Supposons que pour un multiensemble d'opérations B avec $|B| = n$ et un état s, $\mathcal{G}(B, s)$ contienne au moins deux composantes connexes. Pour résoudre le consensus faible, nous utilisons un seul objet partagé O de type T. L'algorithme fonctionne comme suit.

CONSTRUCTION 1. *Initialement, l'objet O est placé dans l'état s. Selon le résultat de [9], il est possible de faire cette hypothèse. En suivant un ordre (canonique) $<$*

sur B, on associe à chaque thread p une opération unique (bien sûr, autorisée par $O.m$) $c_p \in B$. De même, chaque classe $[x]$ dans $\mathcal{G}(B, s)$ est associée à une valeur $d([x]) \in \{0, 1\}$. Lors de la proposition de la valeur u au consensus, le thread p appelle c_p, renvoyant un certain résultat r de cet appel. Ensuite, il lit l'objet O, renvoyant un état s'. Il doit exister $B' \subseteq B$ et $x \in perm(B')$ tels que c_p retourne r dans x et que l'état s' suive c_p dans x. Une classe unique $[z]$ dans $\mathcal{G}(B, s)$ satisfait $z[0] = x[0]$. Le thread p retourne $d([z])$.

Établissons maintenant la validité de la construction précédente. À cette fin, considérons un thread p qui prend une décision lors d'une exécution. Notons l la linéarisation des opérations sur O au cours de cette exécution. Il existe une permutation x des opérations dans B telle que *(i)* l soit un préfixe de x, et *(ii)* si c_p retourne r dans l, alors c_p retourne nécessairement r dans $\tau(s, x)$. D'après *(i)* et le Lemma 1, la première opération dans l, disons c_q, est aussi la première opération dans x. Cette opération identifie la composante, notée $[z]$, à laquelle x appartient. Par *(ii)*, c_p retourne r dans $\tau(s, x)$. Ainsi, p calcule $[z]$ dans la construction et retourne $d([z])$. Cela prouve l'accord.

Pour $u \in \{0, 1\}$, il existe une composante $[z]_u$ avec $d([z]_u) = u$. D'après ce qui précède, pour une permutation $x_u \in [z]_u$, si $x_u[0]$ correspond à la linéarization d'une exécution, alors u est décidée. On considère donc une histoire séquentielle où les opérations des threads sont jouées dans l'ordre x_u. Ceci montre que la validité faible est respectée. □

Pour compléter le résultat précédent, nous proposons ci-dessous une cartographie exhaustive des objets lisibles de la classe CN_1.

Rappelons tout d'abord qu'un objet T est dit "sans histoire" (*historyless* en anglais) lorsque toute paire d'opérations d'écriture (c, d) est recouvrante, à savoir l'état après c est le même qu'après d puis c, et réciproquement [37]. Il est bien établi que les objets de ce type appartiennent à la classe CN_1. De plus, cette classification s'applique également si l'ensemble des opérations d'écriture sont commutatives.

Deux opérations sont qualifiées de *faiblement commutatives* si, quel que soit l'état de l'objet, les appliquer dans un ordre quelconque aboutit au même état final, et une de ces deux opérations n'a pas conscience de l'existence de l'autre opération. À titre d'exemple, une opération de lecture et une opération d'écriture

sont faiblement commutatives. Cela est également vrai pour un incrément et un fetch-and-add.

Un objet T est considéré comme *permissif* lorsque toutes les paires d'opérations d'écriture le concernant sont soit recouvrantes, soit faiblement commutatives. Le résultat suivant établit que les objets lisibles dans CN_1 sont précisément les objets permissifs. Ce résultat a été trouvé concurremment par Khanchandani et al. [40, Théorème 1].

Lemma 2. *Un objet lisible T appartient à CN_1 si, et seulement si, T est permissif.*

Démonstration. ($\Rightarrow$) (Par contradiction.) Supposons que T ne soit pas permissif. Cela implique qu'il existe au moins une paire d'opérations qui n'est ni recouvrante, ni faiblement commutative. Autrement dit, cela signifie qu'il existe un triplet (s, c, d), où c et d sont deux opérations distinctes, tel que :

$$
\begin{array}{rl}
 & \neg \ \vee \ (\tau(s,c) = \tau(s,dc)) \\
 & \quad \vee \ (\tau(s,d) = \tau(s,cd)) \\
 & \quad \vee \ \wedge \ (\tau(s,cd).\mathsf{st} = \tau(s,dc).\mathsf{st}) \\
 & \qquad \wedge \ \vee \ (\tau(s,c).\mathsf{val} = \tau(s,dc).\mathsf{val}) \\
 & \qquad \quad \vee \ (\tau(s.d).\mathsf{val} = \tau(s.cd).\mathsf{val}) \\
= & \wedge \ \vee \ (\tau(s,c).\mathsf{st} \neq \tau(s,dc).\mathsf{st}) \\
 & \quad \vee \ (\tau(s,c).\mathsf{val} \neq \tau(s,dc).\mathsf{val}) \\
 & \wedge \ \vee \ (\tau(s,d).\mathsf{st} \neq \tau(s,cd).\mathsf{st}) \\
 & \quad \vee \ (\tau(s,d).\mathsf{val} \neq \tau(s,cd).\mathsf{val}) \\
 & \wedge \ \vee \ (\tau(s,cd).\mathsf{st} \neq \tau(s,dc).\mathsf{st}) \\
 & \quad \vee \ \wedge \ (\tau(s,c).\mathsf{val} \neq \tau(s,dc).\mathsf{val}) \\
 & \qquad \wedge \ (\tau(s.d).\mathsf{val} \neq \tau(s.cd).\mathsf{val})
\end{array}
$$

Nous allons démontrer que, dans ce contexte, il est possible de résoudre consensus pour deux threads, notés p et q :

Tout d'abord, supposons que les conditions suivantes soient vérifiées : *(i)* $\tau(s,c).\mathsf{val} \neq \tau(s,dc).\mathsf{val}$ et que *(ii)* $\tau(s,d).\mathsf{val} \neq \tau(s,cd).\mathsf{val}$. Pour résoudre consensus dans ce cas, nous utilisons un registre par thread et un objet O de type T, initialement dans l'état s. Le thread p (respectivement q) écrit sa proposition dans son registre, puis applique l'opération c (respectivement d) à l'objet O. Après avoir effectué

ces opérations, chaque thread peut déterminer si l'opération qu'il a appliquée est linéarisée avant ou après l'autre opération. Si p observe que c est linéarisé avant d, il décide de sa propre proposition. Dans le cas contraire, il choisit la proposition de l'autre thread.

Dans le cas où $\tau(s, cd).\mathsf{st} \neq \tau(s, dc).\mathsf{st}$, la construction est légèrement différente. En plus d'écrire dans le registre et d'appliquer les opérations, chaque thread doit également lire l'état de l'objet après l'application de son opération. Cette lecture supplémentaire permet de déterminer l'ordre de linéarisation des opérations.

($\Leftarrow$) Supposons qu'il existe un protocole de consensus pour deux threads implémenté à partir d'objets permissifs et de registres atomiques de lecture/écriture. Comme dans la preuve de Théorème 1, soit $\mathcal{C}$ une configuration bivalente où les deux threads sont prêts à accéder au même objet permissif O. Le thread p est sur le point d'exécuter une opération c qui mène à un état (disons) 0-valent, tandis que l'opération d de l'autre thread q conduit à un état 1-valent.

Premièrement, considérons qu'une opération écrase l'autre : Soit $\mathcal{D}$ la configuration 0-valente si c est appliqué puis d, et soit $\mathcal{D}'$ la configuration 1-valente lorsque d est exécuté en premier. Comme d écrase la valeur écrite par c, $\mathcal{D}$ et $\mathcal{D}'$ ne diffèrent que dans l'état interne du thread p. Ainsi, q prend la même décision en partant de $\mathcal{D}$ et de $\mathcal{D}'$, ce qui est impossible puisque les deux configurations ont une valence différente.

Ensuite, supposons que les deux opérations commutent faiblement : Soit s l'état de l'objet O dans la configuration $\mathcal{C}$. Puisque O est permissif, nous pouvons supposer, sans perte de généralité, que $\tau(s, dc).\mathsf{val} = \tau(s, d).\mathsf{val}$ et $\tau(s, cd).\mathsf{st} = \tau(s, dc).\mathsf{st}$. Soit $\mathcal{D}$ la configuration 0-valente si c est exécutée puis d, et soit $\mathcal{D}'$ la configuration 1-valente si les deux opérations sont exécutées dans l'ordre inverse. D'après ce qui précède, $\mathcal{D}$ et $\mathcal{D}'$ ne sont pas distinguables pour q. Par conséquent, toute exécution solo par q décide de la même valeur depuis $\mathcal{D}$ ou $\mathcal{D}'$, ce qui constitue une contradiction.

□

Dans la partie ($\leq$) de la démonstration du Théorème 1, nous nous appuyons sur le fait qu'un thread puisse lire l'état d'un objet, ce qui implique que l'objet soit *long-lived*. Toutefois, si nous nous limitons aux objets *one-shot*, le raisonnement

devient plus simple, car l'état final après une permutation des opérations n'a plus d'importance. Dans un tel cas, la relation d'équivalence sur l'ensemble des opérations B ne dépend que des valeurs de retour obtenues lors des appels. La partie ($\geq$) de la démonstration reste inchangée. Pour la partie ($\leq$), le thread n'a pas besoin de lire l'état de l'objet. Il utilise simplement la valeur de retour de l'opération pour identifier à quelle composante appartient l'exécution. Cela permet d'obtenir la caractérisation du Lemma 2 en s'appuyant sur l'observation fondamentale suivante : si $CN(T) = 1$, alors pour tout état s et pour tout ensemble d'opérations B tel que $|B| = 2$, le graphe $\mathcal{G}(B, s)$ ne contient qu'une seule composante. Par conséquent, l'objet T doit être permissif.

B.2 Prédire la mise à l'échelle

Proposition 1. Supposons que l'objet O soit one-shot. Il existe une implémentation sans conflits de O si, et seulement si, B est labellisant pour tout $\mathcal{G}_T(B, s)$ dans Γ_O.

Démonstration. ($\Leftarrow$) Pour implémenter O de manière sans conflits, nous procédons ainsi : chaque thread maintient une copie locale de l'objet, initialement dans l'état s. Lorsqu'un thread appelle une opération (au plus une fois, puisque l'objet est one-shot), il applique l'opération à sa copie locale, puis renvoie la valeur de retour correspondante. Il est évident que cette implémentation est à la fois sans attente (*wait-free*) et sans conflits (*conflict-free*). La linéarisation découle du Lemma 1 et du fait que l'ensemble B est labellisant dans $\mathcal{G}_T(B, s)$ pour tout ensemble B. Plus précisément, considérons une exécution ρ dans laquelle les threads effectuent les opérations contenues dans B. Soit x une permutation de B respectant l'ordre temporel dans ρ. Par hypothèse, B est labellisant dans $\mathcal{G}_T(B, s)$. En appliquant le Lemma 1, la valeur de retour d'une opération $c \in B$ est $\tau(s, c).\mathsf{val}$. Il en résulte donc que x est une linéarisation de l'exécution ρ.

($\Rightarrow$) Supposons maintenant que $\mathcal{I}$ soit une implémentation sans conflits et sans attente pour O. Considérons un ensemble B d'opérations de taille $|\mathcal{P}|$, et deux permutations $x, y \in \mathit{perm}(B)$. Soit ρ_x (respectivement, ρ_y) une exécution de $\mathcal{I}$ dans laquelle tous les threads effectuent les opérations dans B selon l'ordre

donné par x (resp. y). Cela est possible puisque B doit respecter les contraintes de $O.m$ et que $\mathcal{I}$ est sans attente.

Considérons maintenant une opération $c \in B$ et une autre opération d qui précède c dans x. Puisque $\mathcal{I}$ est sans conflits, c ne lit jamais un registre écrit par d. Par conséquent, la valeur de retour de c dans x est $\tau(s, c).\mathsf{val}$. La même observation s'applique pour la permutation y. De plus, comme les opérations n'écrivent pas dans un registre partagé, l'état de l'objet reste identique après ρ_x et ρ_y. Cela implique que l'opération c labellise la paire (x, y) dans $\mathcal{G}_T(B, s)$. □

Proposition 2. Il existe une implémentation sans conflits de O si, et seulement si, B est labellisant pour tout $\mathcal{G}_T(B, s)$ dans Γ_O, avec $|B| = 2$.

Démonstration. Nous allons examiner chaque côté de l'équivalence de manière détaillée.

($\Leftarrow$) L'implémentation proposée est exactement la même que dans le cas one-shot. Autrement dit, chaque thread conserve une copie locale de l'objet. Lorsqu'une opération c est exécutée, celle-ci est appliquée à la copie locale et la valeur de retour correspondante est renvoyée. Supposons maintenant que ρ soit une exécution de cette implémentation et que h soit l'histoire induite par ρ. Comme cet algorithme est sans attente (*wait-free*) nous pouvons supposer, sans perte de généralité, que h est une histoire complète. Prenons ensuite une linéarisation de $(ops(h), \ll_h)$, où $ops(h)$ dénote les opérations dans h et $\ll_h$ l'ordre temps-réel dans h. Soit l l'histoire séquentielle où les opérations sont exécutées dans l'ordre de cette linéarisation. Nous devons maintenant démontrer que toutes les opérations dans l produisent exactement les mêmes valeurs de retour que dans h. Cela établira que h est effectivement linéarisable. Pour ce faire, nous parcourons chaque opération, thread par thread, en tenant compte de l'ordre dans lequel elles apparaissent dans l. Chaque opération est « validée » en la déplaçant en tête de la séquence, de manière similaire à l'exécution de ρ. Plus précisément, considérons un thread p. Soit c la dernière opération non validée exécutée par p, et d l'opération immédiatement précédant c dans l. Si d a également été exécutée par p, alors nous avons terminé : le début de la séquence contient toutes les opérations de p exécutées avant c, ce qui correspond exactement à l'exécution de ρ. Sinon, c et d peuvent être échangées tout en conservant les mêmes valeurs de retour partout. En effet, par hypothèse,

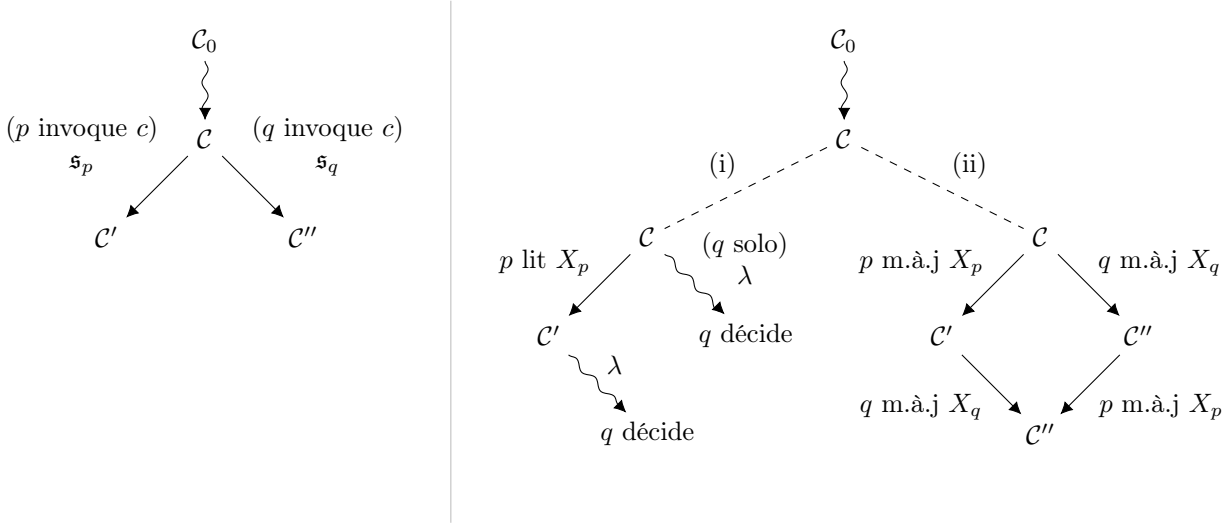

FIGURE B.1 – Illustration du Lemma 3.

c et d produisent les mêmes valeurs de retour avant l'échange, et cet échange ne modifie pas l'état de l'objet.

($\Rightarrow$) Supposons maintenant un graphe d'indistiguabilité $\mathcal{G}_T(B, s)$ dans Γ_O, où $B = \{c, d\}$ est un ensemble d'opérations. Nous allons prouver que, si une implémentation sans conflits $\mathcal{I}$ de O existe, alors B est labellisant dans $\mathcal{G}_T(B, s)$. Nous atteignons l'état s en réalisant d'abord une exécution séquentielle appropriée ρ_0. Cela est possible, car chaque opération peut être exécutée par au moins un thread. Ensuite, considérons que deux threads, p et q, sont sur le point d'exécuter respectivement les opérations c et d (avec $c \in O.m[p]$ et $d \in O.m[q]$). Prenons une continuation où p exécute d'abord c, suivi de l'exécution de d par q. Comme l'implémentation $\mathcal{I}$ est sans conflits, la valeur de retour de d reste identique à celle qui aurait été obtenue si d avait été exécutée en premier. Ainsi, l'opération d est labellisante dans $\mathcal{G}_T(B, s)$. En utilisant un raisonnement symétrique, c est également labellisante. De plus, l'état de l'objet après l'exécution de cd est exactement le même que celui après l'exécution de dc.

□

Lemma 3. *Si c a un pouvoir de consensus, alors il n'existe pas d'implémentation sans conflits de mise à jour de c.*

Démonstration. Soit T le type de données associé à l'opération c. Supposons que c a une implémentation sans conflits de mise à jour, notée $\mathcal{I}$. Soit $\mathcal{A}$ un algorithme résolvant le consensus binaire à l'aide de l'opération c sur des objets de type T, et supposons que tous ces objets soient implémentés avec $\mathcal{I}$. Nous allons construire une exécution ρ de $\mathcal{A}$ dans laquelle deux threads corrects, p et q, ne terminent jamais.

La construction de ρ est illustrée dans la Figure B.1. Partant d'une configuration initiale bivalente $\mathcal{C}_0$, l'exécution ρ atteint une configuration critique bivalente $\mathcal{C}$ en alternant les pas de calculs de p et q. Notons que, si une telle configuraiton n'est jamais atteinte, alors ρ ne décide jamais, ce qui est bien sûr impossible.

Dans la configuration $\mathcal{C}$, les threads p et q sont sur le point d'exécuter le pas de calcul $\mathfrak{s}_p$ et $\mathfrak{s}_q$ sur les registres X_p et X_q, respectivement. Ces pas de calcul sont réalisés dans le cadre d'appels à l'opération c sur le même objet de type T (voir la partie gauche de la Figure B.1). Si ce n'était pas le cas, nous pourrions appliquer directement l'argument classique d'impossibilité du consensus en mémoire partagée basée sur les lectures/écritures atomiques.

Maintenant, en fonction de la nature de ces pas de calculs, plusieurs cas se présentent (partie droite de la Figure B.1) : *(i)* Si $\mathfrak{s}_p$ est une lecture sur X_p, nous considérons une continuation solo λ de q à partir de s, où q parvient à une décision. Cette même continuation λ est applicable à partir de $\mathcal{C}'' = \mathcal{C}.\mathfrak{s}_p$, menant à la même décision, ce qui crée une contradiction car $\mathcal{C}$ est bivalente. *(ii)* Si $\mathfrak{s}_q$ est une lecture, le raisonnement précédent s'applique de manière symétrique. *(iii)* Supposons maintenant que les deux pas de calcul soient des mises à jour des registres. Puisque $\mathcal{I}$ est sans conflits de mise à jour, ces deux étapes ne peuvent pas porter sur le même registre. Ainsi, ces deux opérations sont commutatives. Par conséquent, la valence de la configuration $\mathcal{C}.\mathfrak{s}_p.\mathfrak{s}_q$ est la même que celle de $\mathcal{C}.\mathfrak{s}_q.\mathfrak{s}_p$, ce qui contredit le fait que $\mathcal{C}$ est critique. □

Proposition 3. L'opération c est implémentable sans conflits de mise à jour si c se déplace à gauche dans chaque $\mathcal{G}_T(B, s)$ de Γ_O. Lorsque c possède un pouvoir de consensus, cette condition est également nécessaire.

Démonstration. Nous démontrons d'abord que la condition est suffisante. Supposons que c soit un *left-mover*. On va considérer l'implémentation suivante :

CONSTRUCTION 2. *Les threads partagent une horloge globale. Chaque thread possède également une copie de l'objet, un journal et une horloge locale. Les entrées du journal sont des paires de la forme $E = (d, \theta)$, où d est une opération et θ son horodatage (un entier).*

*Lors de l'appel d'une opération c, le thread lit d'abord l'horloge globale et stocke sa valeur dans la variable t. Puis : **(1)** Si c est un left-mover, le thread ajoute une entrée $E = (c, t)$ à son journal, applique c à l'état local et stocke la valeur de réponse dans la variable v. Sinon, **(2)** si c n'est pas un left-mover, le thread ajoute une entrée $E = (c, \perp)$ à son journal. Puis **(2a)** le thread lit et incrémente l'horloge globale, et exécute une opération compare-and-swap pour modifier l'horodatage de E avec la valeur de temps obtenue. Ensuite, **(2b)** le thread récupère l'horodatage assigné à c (par lui-même ou un autre thread) et le stocke dans la variable t. Le thread **(2c)** parcourt les journaux pour rechercher les opérations en attente, c'est-à-dire les opérations dont l'horodatage est encore $\perp$. Pour chaque opération en attente, il exécute l'étape 2a pour lui assigner un horodatage. Ce mécanisme d'aide est répété pour chaque autre thread dans l'ordre de son journal jusqu'à ce qu'un horodatage supérieur à t soit assigné. Ensuite, **(2d)** le thread identifie dans les journaux toutes les opérations comprises entre son horloge locale et t. Les opérations non encore appliquées sont appliquées à la copie locale dans l'ordre de leurs horodatages. Lors de cette application, la valeur de réponse de l'opération c est stockée dans la variable v. Enfin, **(3)**, l'horloge locale est mise à jour à t avant que le contenu de la variable v soit retourné comme résultat de l'opération.*

La construction ci-dessus est sans blocage car, lors de l'aide à l'étape 2c, un thread poursuit jusqu'à assigner un horodatage supérieur à la variable t. Nous prouvons ensuite que cette construction est également linéarisable.

Pour ce faire, considérons une exécution λ de la construction ci-dessus et l'histoire correspondante h. Puisque la construction est sans blocage, nous pouvons toujours étendre λ pour compléter toute opération en attente. Ainsi, sans perte de généralité, nous supposerons que h est complet. Dans l'exécution λ, un *left-mover* c est *vu* par une opération d lorsque c est appliqué au moment où la réponse de d est calculée, c'est-à-dire lorsque le thread qui fait l'appel exécute l'étape 2d.

Soit L la séquence des opérations qui ne sont pas des *left-movers* dans h,

ordonnée selon leurs horodatages. À partir de L, nous créons une séquence L' de la manière suivante : Pour chaque opération $c \in L$, nous ajoutons à L' tous les *left-movers* dans h vus par c et n'y figurant pas encore. Ensuite, nous ajoutons l'opération c à L'. Ce processus est répété pour toutes les opérations dans L dans l'ordre de la séquence ($<_L$). À partir de L', nous obtenons ensuite $\hat{L}$ en réordonnant les *left-movers* dans L' selon le temps réel. Plus précisément, nous initialisons $\hat{L}$ avec L'. Puis, pour deux *left-movers* c et d dans $\hat{L}$, si $c \rightsquigarrow_h d$ mais $d <_{\hat{L}} c$, les opérations c et d sont échangées dans $\hat{L}$. Soit l l'histoire séquentielle induit par $\hat{L}$. Nous montrons ci-dessous que l est une linéarisation de h.

Dans la construction ci-dessus, l'état local est toujours défini de manière unique par la séquence des opérations appliquées jusqu'ici. Nous pouvons formuler les deux invariants suivants concernant la construction : *(INV1)* si c précède d et que d a un horodatage, alors c a un horodatage inférieur à celui de d ; et *(INV2)* si c a un horodatage inférieur à celui de d et que d n'est pas un *left-mover*, alors d voit c. La preuve de ces invariants est simple et donc omise.

Premièrement, considérons deux opérations $c \rightsquigarrow_h d$. En raison de l'INV1, c précède d dans L, ce qui implique $c \rightsquigarrow_l d$, comme requis.

Deuxièmement, nous montrons que la valeur de retour d'une opération c dans l est la même que dans h. Cela est trivial si c est un *left-mover*. En effet, quel que soit l'état local, l'opération renvoie toujours la même réponse, celle de l'état initial (voir la Section 3.2.5). Supposons maintenant que c n'est pas un *left-mover*.

Soit p le thread exécutant c dans h. Si c voit d, alors par construction d précède c dans L'. C'est également le cas dans $\hat{L}$, sauf si d a été échangé avec une opération e après d. Dans ce cas, $e \rightsquigarrow_h d$. Cependant, dans un tel cas, par INV2, c voit e et doit donc précéder c dans L', ce qui est une contradiction.

Ensuite, nous prouvons que chaque opération avant c dans $\hat{L}$ est effectivement vue par c. Considérons une opération d précédant c dans $\hat{L}$. Il y a deux cas à considérer : *(i)* d précède c dans L', et *(ii)* c précède d dans L'. Si (i) est vrai, alors par construction, c voit d, comme requis. Sinon (ii), d est échangée avec une opération $e <_{L'} c$ car $d \rightsquigarrow_h e$. Par INV2, parce que c voit e, il doit aussi voir d, ce qui contredit le fait que d est après c dans L'.

D'après ce qui précède, nous savons que les opérations avant c dans $\hat{L}$ sont celles que c voit. Nous utilisons cela pour prouver que la valeur de retour dans l

est la même que dans h.

Soit x la séquence des opérations appliquées avant c pour calculer sa valeur de réponse (étape 2d). Soit y égal à $\hat{L}_{|<c}$, c'est-à-dire toutes les opérations avant c dans $\hat{L}$. Nous avons établi que x et y contiennent exactement les mêmes opérations. Selon INV2, les opérations qui ne sont pas des *left-movers* sont dans le même ordre dans x et y. Ainsi, les deux séquences ne peuvent différer qu'en ce qui concerne les *left-movers*. Soit d le premier *left-mover* dans x mal positionné par rapport à y. Nous avançons d dans x en l'échangeant succéssivement avec l'opération juste avant lui jusqu'à ce qu'il atteigne sa position dans y. Soit $\hat{x}$ la séquence d'opérations résultante. Comme d est un *left-mover*, $\tau(s_0, x).\mathsf{st} = \tau(s_0, \hat{x}).\mathsf{st}$ (voir la Section 3.2.5). Ainsi, $\tau(s_0, x.c).\mathsf{val} = \tau(s_0, \hat{x}.c).\mathsf{val}$, comme requis.

Nous en concluons que la valeur de retour de c dans h et dans l est la même.

À l'inverse, supposons que c a le pouvoir de consensus. D'après le Lemma 3, il n'existe pas d'implémentation telle que c est sans conflits de mise à jour. Ainsi, par vacuité, le fait que c soit *left-mover* est nécessaire. □

Proposition 4. L'opération c est implémentable de manière invisible si c se déplace à droite dans chaque $\mathcal{G}_T(B, s)$ de Γ_O.

Démonstration. Pour implémenter convenablement l'objet, les threads emploient une file non-bornée *arr* qui est sans attente. Cette liste permet de concaténer une valeur (*offer*), connaître l'index de son dernier élément (*last*), ainsi que l'élément à un certain index (*get*). Outre *arr*, chaque thread maintient une copie locale de l'objet ainsi qu'un pointeur *pnt*, lequel indique l'index de la dernière opération dans *arr* qui a été appliquée à la copie locale. On exécute une opération c ainsi : Si c n'est pas un *right-mover*, le thread annonce c en l'ajoutant à la fin de *arr* puis il applique toutes les opérations (non encore appliquées) avant c à sa copie locale. Sinon, le thread n'annonce pas l'opération c. Á la place, il applique localement toutes les opérations non encore appliquées dans *arr* (en s'arrêtant à la dernière opération observée au moment de l'invocation). Dans tous les cas, le thread applique ensuite localement l'opération c et retourne sa réponse comme résultat de l'invocation. Au cours de ce calcul, le pointeur *pnt* est mis à jour de façon appropriée.

La construction ci-dessus garantit que les *right-movers* sont invisibles et que toute invocation de c est sans attente (puisque les objets utilisés sont également

sans attente et que les threads font un nombre de pas de calcul fini à chaque invocation). Prenons maintenant une exécution ρ et son histoire induite h. Comme précédemment, on supposera que h complète.

On linéarise les opérations en fonctions de leur accés à la variable partagée arr. En détail, le point de linéarisation des opérations non *right-movers* de h se produit lorsqu'elles sont ajoutées à arr (avec *offer*), tandis que celui des *right-movers* survient au moment de l'observation du dernier élément dans arr (avec *last*). On notera l une telle linéarisation. Par construction, on observe que $<_h \subseteq <_l$.

On montre désormais que les histoires l et h sont équivalentes. Soit c une opération. On observe que toutes les opérations appliquées au moment du calcul de la réponse de c dans ρ sont avant c dans l. On déplace succéssivement toutes les opérations avant c dans l, et non appliquées au moment du calcul de sa réponse dans ρ, après c (dans un ordre quelconque). Ces opérations sont nécessairement des right-movers. Or, par définition, déplacer un *right mover* à droite d'une opération ne change en rien le résultat de cette opération. Soit l_c l'histoire séquentielle ainsi obtenue. Le résultat de c dans l_c est le même que celui dans l. De plus, par construction, il est identique à celui de c dans h.

De ce qui précède, nous pouvons conclure que la construction est linéarisable. □

B.3 Objets ajustés

Proposition 5. Considérons que l'objet O ajuste l'objet O'. Alors, toute tâche distribuée pouvant être résolue avec O (en conjonction avec des registres) peut également être résolue avec O'.

Démonstration. Soit Δ une tâche distribuée. Considérons un algorithme $\mathcal{A}$ qui résout Δ à l'aide d'instances de O et de registres. Nous obtenons l'algorithme $\mathcal{A}'$ en remplaçant chaque instance de O par une instance de O'. L'algorithme $\mathcal{A}'$ est valide puisque, comme O ajuste O', *(i)* toute méthode existant dans O existe dans O' du fait que $O'.T$ est un sous-type (restreint) de $O.T$, et *(ii)* pour tout thread p, si p peut appeler la méthode f de O, alors il peut aussi le faire avec O' (car $O.m \subseteq O'.m$). Considérons maintenant une exécution ρ de $\mathcal{A}$. Soit h

l'histoire associée à ρ. Désignons par l une linéarisation de h. D'après le principe de substitution de Liskov, puisque $O'.T$ est un sous-type de $O.T$, pour toute instance x de $O'.T$, l'histoire $l|x$ reste une histoire séquentielle valide pour $O.T$. (Á savoir, les opérations retournent les mêmes valeurs de retours.) Ainsi, l'histoire h est aussi une histoire de l'algorithme $\mathcal{A}$. Par conséquent, Δ est résolue dans ρ. □

Proposition 6. Supposons que O ajuste O'. Soit s un état commun à O et O', et B un ensemble d'opérations tel que B respecte à la fois $O.m$ et $O'.m$. Alors, $G_{O'.T}(B, s) \subseteq G_{O.T}(B, s)$.

Démonstration. De manière évidente, tous les sommets de $G_{O.T}(B, s)$ apparaissent également dans le graphe $G_{O'.T}(B, s)$, Prenons une permutation x de B, et considérons l'histoire séquentielle h obtenu en appliquant x depuis l'état s selon $O.T$. En vertu du principe de substitution de Liskov, cette même histoire h est aussi valable pour x depuis s selon $O'.T$. Ainsi, pour chaque arête (x, y) reliant deux permutations x et y dans $G_{O.T}(B, s)$, il existe également une arête (x, y) dans $G_{O'.T}(B, s)$, prouvant que $G_{O.T}(B, s)$ est bien inclus dans $G_{O'.T}(B, s)$. □

ECOLE
DOCTORALE

Titre : Les objets ajustés : Une approche bien fondée et efficace de la programmation concurrente

Mots clés : Mémoire partagé ; JAVA ; Objets concurrents

Résumé : Depuis le milieu des années 2000, la hausse de la fréquence des transistors a ralenti, limitée par leur capacité à dissiper la chaleur. Pour maintenir les performances, les concepteurs ont adopté des architectures multi-cœurs, rendant le parallélisme indispensable à l'exploitation des capacités de calcul. Cependant, écrire des programmes parallèles et concurrents reste complexe, nécessitant la gestion des entrelacements de threads, du réordonnancement des tâches et du modèle mémoire sous-jacent.
Pour aider les développeurs, des bibliothèques comme java.util.concurrent et Boost.LockFree proposent des objets concurrents. Conçues pour couvrir de nombreux cas d'utilisation, elles offrent des interfaces riches et complexes, souvent basées sur des mécanismes tels que les verrous, barrières ou primitives non bloquantes (e.g., compare-and-swap), impactant directement les performances.
Pour de meilleures performances, les développeurs créent parfois des versions ad hoc de ces objets, que nous appelons objets ajustés. Cette thèse vise à poser les bases théoriques et pratiques des objets ajustés.
Nous débutons par un état de l'art sur la programmation parallèle et concurrente, suivi d'une analyse de plusieurs systèmes de gestion de données pour observer l'utilisation effective des objets concurrents.
Ensuite, nous proposons une première définition formelle des objets ajustés, introduisant un outil mesurant la parallélisabilité d'un objet : le graphe d'indistinguabilité.
Nous présentons ensuite DEGO, une bibliothèque d'objets ajustés offrant des versions optimisées de structures concurrentes classiques comme les dictionnaires, ensembles, files et compteurs, adaptées à des contextes spécifiques (par exemple, lorsque les écritures sont commutatives).
Enfin, nous évaluons DEGO en comparant ses performances à celles de 'java.util.concurrent' à l'aide de micro-benchmarks et d'une application de type réseau social.

Title : Adjusted Objects : A Principled and Efficient Approach to Concurrent Programming

Keywords : Shared Memory ; JAVA ; Concurrent Object

Abstract : Since the mid-2000s, the increase in transistor frequency has slowed down, limited by their ability to dissipate heat. To maintain performance, designers have adopted multi-core architectures, making parallelism essential for fully utilizing computing power. However, writing parallel and concurrent programs remains complex, requiring the management of thread interleavings, task reordering, and the underlying memory model.
To assist developers, libraries such as 'java.util.concurrent' and Boost.LockFree provide concurrent objects. Designed to cover a wide range of use cases, these libraries often feature rich and complex interfaces, frequently relying on mechanisms like locks, barriers, and non-blocking primitives (e.g., compare-and-swap), which directly impact performance.
For better performance, developers sometimes create ad hoc versions of these concurrent objects, referred to as adjusted objects. This thesis aims to establish the theoretical and practical foundations of adjusted objects.
We begin with a review of parallel and concurrent programming support, followed by an analysis of several data management systems to examine how developers use concurrent object interfaces in practice.
Next, we propose the first formal definition of adjusted objects, introducing a new tool to measure an object's parallelizability : the indistinguishability graph.
We then present DEGO, a library of adjusted objects offering optimized versions of common concurrent structures, such as dictionaries, sets, queues, and counters, tailored for specific contexts (e.g., when all write operations are commutative).
Finally, we evaluate DEGO by comparing its performance against 'java.util.concurrent' using micro-benchmarks and a social network-style application.

Institut Polytechnique de Paris
91120 Palaiseau, France