Projet Données Réparties 2021-2022

Basile Gros Raphaëlle Monville-Letu Christophe Pantel

1 Introduction

Le projet a pour but de réaliser un espace partagé de données typées. Cette approche est inspirée du modèle Linda (ou TSpaces). Dans ce modèle, les processus partagent un espace de tuples qu'ils peuvent manipuler à l'aide d'un jeu de primitives spécifiques. Nous allons implanter trois versions différentes, plusieurs applications et des tests unitaires pour vérifier le bon fonctionnement de nos programmes.

Résumons d'abord les différents aspects du sujet du projet.

1.1 Services fournis

Nous travaillons à partir des interfaces `Linda` et classes `Callback`, `AsynchronousCallback` qui sont figées et ne doivent en aucun cas être modifiées.

Les services qui doivent être implantés sont les suivants :

- `read` : renvoie un tuple compatible avec le motif si l'espace en contient un, sinon se bloque en attendant qu'un tuple compatible soit écrit ;
- `tryRead` : renvoie un tuple compatible avec le motif si l'espace en contient un, sinon renvoie `null` ;
- `readMany` : renvoie tous les tuples compatible avec le motif dans l'espace, éventuellement aucun ;
- `take` : renvoie un tuple compatible avec le motif si l'espace en contient un et supprime celui-ci de l'espace, sinon se bloque en attendant qu'un tuple compatible soit écrit ;
- `tryTake` : renvoie un tuple compatible avec le motif si l'espace en contient un et supprime celui-ci de l'espace, sinon renvoie `null` ;
- `takeMany` : renvoie et supprime tous les tuples compatible avec le motif dans l'espace, éventuellement aucun ;
- `write` : écrit un tuple dans l'espace, débloque éventuellement une lecture ou une prise bloquée en attente d'une écriture compatible, déclenche éventuellement une lecture ou une prise enregistrée pour une exécution asynchrone, une seule de ces actions bloquées ou asynchrones aura lieu ;
- `register` : enregistre une action asynchrone qui peut être une lecture ou une prise, cette action peut être immédiate, dans ce cas, si l'action est possible, elle est exécutée et supprimée, sinon elle est enregistrée pour être exécutée lors de futures écritures. Une action est supprimée dès qu'elle est exécutée. Une action est représentée par une classe qui implante l'interface `AsynchronousCallback`.

1.2 Rappels sur les tuples

Un tuple de valeurs est une séquence ordonnée de valeurs. Les valeurs sont de type quelconque (entier, booléen, objets, ... ou même tuple). Pour ce projet, une implantations des tuples est fournie par les enseignants. Cette classe `Tuple` repose sur la sérialisation des objets Java pour la communication à travers la technologie RMI dans la partie distribuée du projet. Elle fourni la méthode `match` qui permet de comparer deux tuples, ou plus précisément de déterminer si un tuple est compatible avec un motif représenté par un autre tuple. Un tuple motif (ou template) est un n-uplet dont certaines des composants peuvent des types représentés par les objets classes Java associés (mécanisme de réflexivité en Java), qui représentent « n'importe quelle valeur de ce type ».

1.3 Version en mémoire partagée

Il s'agit de réaliser une implantation de l'interface `Linda` qui s'exécute directement dans la même machine virtuelle que les codes utilisateurs. L'implantation doit être dans la classe `linda.shm.CentralizedLinda` avec un constructeur sans paramètre.

Dans ce projet, nous avons d'abord réalisé une implantation séquentielle de la version en mémoire partagée puis avons écrit une implantation concurrente. Par la suite nous avons comparé les performances entre les deux implantations.

1.4 Version Client/Serveur

Il s'agit de réaliser une implantation de l'interface `Linda` qui accède à un serveur distant qui centralise l'espace des tuples. La classe `linda.server.LindaClient` doit être une implantation de l'interface `Linda` qui permettra d'exécuter les services sur le serveur distant. Le constructeur doit prendre un unique paramètre chaîne qui est l'URI du serveur linda à utiliser (par exemple //localhost :4000/LindaServer).

Nous avons écrit le code de telle sorte que l'implantation du serveur réutilise en interne la version linda en mémoire partagée sans y toucher une virgule et nous pensons ne pas avoir tricher car nous n'avons pas modifié notre version centralisée séquentielle pour la construction de la version distribuée (sauf, bien sûr, pour corriger des erreurs que nous n'avions pas trouvées auparavant et pour construction la version centralisée concurrente.

1.5 Applications

Pour assurer le bon fonctionnement de nos programmes, nous devons réaliser une série d'applications. L'objectif sera, d'une part d'évaluer différents choix de conception et d'architecture pour les applications concurrentes, et d'autre part de permettre de comparer et valider les différentes versions du noyau lui-même.

L'archive fournie par les enseignants comporte quelques exemples simples et un exemple caractéristique complet d'une famille d'applications où l'utilisation du modèle Linda est pertinente : il s'agit d'un « tableau blanc » interactif, partagé par un ensemble d'utilisateurs. Le contenu du tableau blanc est conservé dans l'espace de tuples. Chaque utilisateur affiche sa copie du tableau blanc à partir des évolutions de l'espace de tuples et peut modifier le contenu du tableau blanc en modifiant l'espace de tuples.

Nous avons également compléter une application fournie par les enseignants, et développer une autre application spécifiée par les enseignants, qui se focalisent sur une autre catégorie d'applications pour laquelle le modèle Linda est adapté : les calculs intensifs sur des masses de données.

— Calcul des nombres premiers inférieurs à K (application développée complètement) ;
— Recherche approximative dans un fichier (application fournie par les enseignants et complétée par les étudiants).

2 Version en mémoire partagée

Comme indiqué dans le rapport préliminaire, nous avons d'abord réalisé une version peu concurrente (que nous appelons séquentielle) dans laquelle la concurrence découle uniquement des activités qui appellent les opérations de Linda, c'est-à-dire qu'il n'y a pas d'activités créée en interne pour accélérer la recherche de tuples dans l'espace de tuples. C'est une forme de lecteur/rédacteur avec une opération supplémentaire de prise qui supprime la valeur lue, et la possibilité d'enregistrer des événements asynchrones. La lecture peut se faire à plusieurs en parallèle, mais les autres opérations doivent être en exclusion.

Nous avons ensuite développée une version concurrente qui s'appuie sur le parallélisme de données : nous découpons l'espace des tuples en plusieurs morceaux et nous exécutons les opérations en parallèle sur les différents morceaux, que ce soit des lectures ou des prises. Cette nouvelle architecture permettra de faire plusieurs lectures en parallèle en utilisant un nombre d'activités maximum supérieur aux nombres de morceaux d'espace de tuples.

Nous avons développé la version faiblement parallèle sans avoir d'idées précises de la manière dont nous pourrions augmenter le parallélisme. Nous avons ensuite identifié les points communs que nous avons factorisé dans une classe abstraite `AbstractCentralizedLinda` en faisant apparaître les méthodes implantant les opérations quasi-séquentielles ou parallèles. Nous observons que toute la gestion de l'accès concurrent des utilisateurs à linda est identique entre les deux versions et que nous distinguons ensuite l'accès quasi séquentiel ou parallèle à l'implantation de l'espace des tuples.

2.1 Représentation de l'espace des tuples

Nous avons défini la classe `TupleSpace` qui contiendra l'intégralité de l'espace des tuples dans la version quasi-séquentielle, et un morceau de l'espace des tuples dans la version parallèle. Nous avons trois catégories de méthodes :

— Celles qui sont spécifiques à la version quasi-séquentielle et peuvent être utilisées pour la version parallèle de manière inefficace ;
— celles qui sont spécifiques à la version parallèle qui ont été dérivée des précédentes pour gérer plus simplement la coordination entre les différentes activités. Par contre, étant donné qu'elles contiennent des éléments de synchronisation, elles sont plus coûteuses que les versions précédentes ;
— celles qui sont communes aux deux versions car la parallélisation ne change pas la sémantique.

L'espace des tuples est définie comme une `ArrayList` de tuple. Nous avons choisi cette implantation des listes car elle est plus efficace dans le parcours des listes mais plus coûteuse lors de l'ajout d'éléments.

Pour les fonctions de lecture (`readOnce` et `readMany`), et d'écriture (`takeOnce` et `takeMany`) définies dans le fichier `TupleSpace.java`, on fournit un template/motif de tuple que l'on recherche dans l'espace des tuples.

Pour `readOnce`, nous exploitons un `iterator` sur l'espace des tuples qui va parcourir l'ensemble des éléments de l'espace jusqu'à trouver un tuple qui correspond au motif transmis en paramètre. Il renvoie ensuite le tuple recherché en en faisant un clone ou `null` si aucun ne satisfait le motif.

Pour `readMany`, nous utilisons une approche similaire sauf que l'on introduit une liste de tuples qui contiendra l'ensemble des tuples lus à renvoyer. Celle-ci peut être vide.

Pour `takeOnce` et `takeMany`, nous utilisons les mêmes approches et nous enlevons les éléments de l'espace des tuples avec la méthode `remove`.

La méthode `writeOnce` est très simple car nous n'avons juste qu'à rajouter un tuple à l'espace en utilisant la fonction `add`.

Ces opérations n'utilisent aucune primitive de synchronisation. C'est l'appelant qui devra assurer la gestion de la coordination.

Les appels utilisant chacun leur propre itérateur, ils peuvent s'exécuter en parallèle tant qu'ils ne modifient pas les données manipulées comme les opérations de lecture. Par contre, les opérations de prise et d'écriture doivent s'exécuter en exclusion. Cette exclusion est assurée par la classe `AbstractCentralizedLinda` vis-à-vis de l'utilisateur.

Lorsque nous avons implanté la version centralisée parallèle, il était nécessaire d'avoir plusieurs activités qui travaillent en parallèle sur les différents morceaux de l'espace des tuples et qui se coordonnent pour abandonner le parcours dès qu'une des activités a réussi. Cela concerne la lecture et la prise d'un tuple unique qui ne doivent lire ou prendre un tuple que dans un seul morceau de l'espace des tuples alors que plusieurs activités travailleront en parallèle sur les différents morceaux.

Nous avons donc implantés des variantes de `readOnce` et `takeOnce` qui utilisent le principe du Test And Set pour se coordonner. Nous pourrions utiliser ces versions de manière séquentielle également mais, nous avons choisi de garder les implantations précédentes moins coûteuses.

Nous avons implanté des opérations pour écrire et lire l'espace des tuples depuis un fichier en utilisant la sérialisation des tuples et des `ArrayList` mais nous n'avons finalement pas utilisé ces opérations.

2.2 Coordination de l'utilisation des services linda

Un même espace de tuples linda peut être utilisé par plusieurs clients simultanément. Il est donc nécessaire de coordonner ces utilisations. Étant donné la similitude avec le problème du lecteur/rédacteur, nous nous sommes inspirés des travaux pratiques pour implanter la coordination en linda sous la forme d'un moniteur.

Nous devons coordonner quatre familles d'opération : lecture, prise, écriture, enregistrement des actions asynchrones. Plusieurs opérations de la première famille peuvent s'exécuter simultanément car elles ne modifient pas les données. Les opérations des trois autres familles modifient les données partagées (l'espace des tuples, ou les actions asynchrones enregistrées). Elles doivent donc être exécutées en exclusion.

Nous utilisons un seul moniteur et quatre variables condition (une par famille d'opération). Nous donnons la priorité à l'enregistrement des actions asynchrones, puis à la lecture, à la prise et enfin à l'écriture. Celui-ci est implanté dans la classe `AbstractCentralizedLinda` dont héritent ensuite les implantations quasi séquentielle et concurrente.

Lors de l'enregistrement d'une action asynchrone, si celle-ci est immédiate, nous essayons de l'exécuter tout de suite, et nous ne l'enregistrons que si elle n'est pas possible. Nous l'enregistrons si elle n'est pas immédiate.

Les opérations asynchrones sont mémorisées dans deux listes (`ArrayList`) : celle des actions de lecture et celle des actions de prise.

Lors d'une écriture, nous commençons par consulter les listes d'actions asynchrones, si plusieurs lectures sont possibles, nous les exécutons et les supprimons, si plusieurs prises sont possibles, nous n'en exécutons qu'une et nous la supprimons. Nous enregistrons le tuple dans l'espace uniquement si nous n'avons pas déclenché d'action de prise. Une gestion précise du moniteur est nécessaire car les actions déclenchées peuvent faire appel de manière synchrone à linda et se bloquer dans le moniteur. Nous devons donc quitter le moniteur pour exécuter les actions déclenchées qui peuvent alors choisir de rentrer à nouveau dans le moniteur en exécutant des opérations linda.

Cette coordination est identique pour les versions quasi séquentielle et concurrente. Nous avons identifié les opérations suivantes qui sont appelées par les services coordonnés offerts aux utilisateurs de linda et qui ne demandent pas de coordination supplémentaire (elles font l'hypothèse qu'elles sont utilisées de manière coordonnée) :

- `readOnce` et `readMany` qui lisent un, respectivement plusieurs, tuples depuis l'espace des tuples de manière non bloquante, c'est-à-dire que la lecture renvoie `null` si aucun tuple n'est conforme au motif ;
- `takeOnce` et `takeMany` qui lisent puis suppriment un, respectivement plusieurs, tuples depuis l'espace des tuples de manière non bloquante, c'est-à-dire que la prise renvoie `null` si aucun tuple n'est conforme au motif.
- `writeOnce` qui écrit un tuple dans l'espace.

Dans la version initiale de notre projet, nous avions implanté le blocage des lectures et prises élémentaires lorsque l'espace de tuples ne contient pas le motif recherché en utilisant les variables conditions de lecture et de prise. Ceci n'est pas totalement satisfaisant car nous allons les réveiller régulièrement et relancer le parcours de l'intégralité de l'espace des tuples, ce qui est coûteux alors que nous pourrions observer les écritures avec un profil particulier, celui de l'opération de lecture ou d'écriture qui n'a pas réussi. Nous avons donc choisi de faire évoluer cette implantation pour nous appuyer sur la gestion des événements asynchrones et sur des variables conditions créées à la demande pour chaque opération de lecture ou prise qui a échoué. Nous avons défini la classe `WaiterCallback` qui mémorise le moniteur et crée une variable condition qui sera utilisée pour bloquer la lecture ou la prise en attente. Lorsque l'action est déclenchée, elle mémorise le tuple lu ou pris, puis elle libère l'opération initiale de lecture ou de prise. Cette approche va augmenter le nombre d'actions asynchrones en attente mais va éviter de parcourir de nombreuses fois l'espace des tuples.

Nous ne donnons pas les détails précis de chacune des opérations. Ces détails sont essentiels pour assurer une coordination correcte et éviter les interblocages qui nous avons détectés lors des tests élémentaires. Nous renvoyons le lecteur vers les sources des différentes classes pour observer ces détails.

Nous avons placé beaucoup de traces à chaque étape de l'exécution de manière à pouvoir comprendre les éventuels défaut à l'exécution. Nous avons ensuite mis en commentaire ces traces pour éviter qu'elles perturbent les mesures de performances. Nous n'avons pas trouvé de mécanisme simple pour désactiver les traces à la compilation sans impact sur l'exécution.

2.3 Implantation séquentielle

La classe `CentralizedSequentialLinda` correspond à l'implantation séquentielle du problème. Nous utilisons un seul objet `TupleSpace` initialisé dans le constructeur. Les opérations implantées utilisent directement les opérations du `TupleSpace` sans besoin de coordination supplémentaire.

2.4 Implantation concurrente

La classe `CentralizedtLinda` correspond à l'implantation parallèle du problème.

Nous avons choisi de mettre en place du parallélisme de données en découpant l'espace des tuples en plusieurs morceaux qui pourront être exploités en parallèle par plusieurs activités. Le nombre de morceau est un paramètre dans la construction des objets de la classe `CentralizedLinda` avec une valeur par défaut pour respectant l'interface imposée par le sujet.

Nous avons choisi de nous appuyer sur les services de gestion des tâches `ExecutorService` de nature `FixedThreadPool` pour créer un groupe d'activités qui seront réutilisées à chaque exécution parallèle. Nous limitons ce parallélisme à un nombre fixe d'activités par morceau d'espace, ceci pour éviter de surcharger la machine en créant un trop grand nombre d'activités. Le nombre d'activités par morceau est un paramètre dans la construction des objets de la classe `CentralizedLinda` avec une valeur par défaut pour respectant l'interface imposée par le sujet.

Lorsqu'une écriture doit être effectuée, nous choisissons d'écrire dans un des morceaux de manière équitable : nous considérons le tableau des morceaux comme un anneau et nous écrivons dans le morceau suivant à chaque étape. Ceci est équitable si nous ne faisons que des écritures et des lectures. Par contre, les prises peuvent totalement déséquilibrer la taille des morceaux. Nous aurions pu choisir d'écrire dans le plus petit morceau mais cela aurait un coût consistant à identifier le morceau le plus petit à chaque écriture, ou à chaque prise. Ceci pourrait être fait sans difficulté dans une version future et nous pourrions comparer les différentes versions selon les applications.

Lorsque nous effectuons des lectures ou des prises multiples, nous pouvons travailler en parallèle sur les différents morceaux sans se coordonner avant la fin des différentes activités. En effet, toutes les lectures et toutes les prises doivent être faites dans tous les morceaux. Nous devons juste faire en fin d'activités, l'union des résultats des différentes activités.

Nous utilisons pour cela le service de terminaison `ExecutorCompletionService` pour lancer les différentes activités, gérer leurs terminaisons, accéder à leurs résultats sous la forme de `Future`. Chaque opération de lecture ou de prise va créer son propre service de terminaison qui utilise le groupe de tâche partagé pour lancer les exécutions. Chaque exécution est représentée par un objet `ReadOnceCallable`, ou `ReadManyCallable`, ou `TakeOnceCallable`, ou `TakeManyCallable`, qui sont similaires à des objets `Runnable` qui renvoie un résultat. Lors de la création de ces objets, nous transmettons les paramètres de l'exécution, principalement le morceau d'espace concerné, le motif recherché, et les éléments de coordination quand cela est nécessaire (lecture et prise unique). Nous avons factorisé des éléments dans `AbstractOnceCallable` et `AbstractManyCallable` mais il nous semble qu'il serait possible de mieux factoriser.

Nous parcourons ensuite de manière bloquante les différentes activités auxquelles nous accédons par ordre de fin d'exécution, nous combinons les résultats et nous transmettons ceux-ci lorsque toutes les activités sont terminées.

Pour coordonner les opérations de lecture et prise unique, nous exploitons la classe `AtomicBoolean` qui permet de créer des objets partagés qui fonctionneront selon le principe du Test And Set atomique. Chaque activité d'une même requête partagera ce booléen. Lorsque celui-ci est à `true`, cela signifie que le tuple recherché a été trouvé par une autre activité, sinon il faut continuer à chercher. Nous utilisons l'opération `getAndSet` pour changer la valeur de ce dernier lorsque l'activité trouve le tuple recherché et ainsi alerter toutes les autres activités qui étaient aussi en train de le rechercher. Nous avons appliqué cette méthode sur la lecture (`readOnce` en ajoutant le paramètre `cancelled` au constructeur), et la prise (`takeOnce` en ajoutant le paramètre `cancelled` au constructeur) d'un tuple spécifique de l'espace des tuples. L'opération `writeOnce` n'en a pas besoin car elle travaille sur un seul morceau, donc sans parallélisme.

Le défaut de ces approches est que les programmes ne se terminent plus car il reste des activités possibles dans l'objet `ExecutorService`, même si toutes les activités sont finis. Il est alors nécessaire de demander à ce service de ce terminer. Nous avons ajouté un service `stop` dans une interface `ExtendedLinda` pour ne pas modifier l'interface imposée par le sujet. Nous ajoutons également une opération `getElapsedTime` pour mesurer le temps d'exécution et faire une analyse de performance.

Notons que nous n'avons parallélisé que la représentation et la gestion de l'espace des tuples. Une partie pourrait être parallélisée, la gestion des actions asynchrones enregistrées. En effet, nous utilisons deux listes qui mémorisent les actions de lecture et de prise. A chaque écriture, nous parcourons ces deux listes pour en extraire les actions qui peuvent être déclenchées, puis nous déclenchons les actions en dehors du moniteur. Le parcours des listes est effectué en exclusion mutuelle. Si la taille de ces listes devient importante, cela réduit significativement les performances. Nous pourrions paralléliser l'exploitation de ces listes de la même manière que l'espace des tuples, d'abord en découpant ces listes en plusieurs morceaux puis en les parcourant en parallèle en synchronisant les résultats comme nous venons de le décrire. L'approche serait exactement la même et pourrait être implantée sans difficulté particulière.

3 Version Client/Serveur

Nous devons ensuite implanter une version distribuée de linda s'appuyant sur la technologie RMI (Remote Method Invocation) qui implante le modèle RPC (Remote Procedure Call) en Java. Il s'agit d'une architecture Client/Serveur qui doit exploiter la version centralisée précédente au niveau du serveur si cela est possible sans modification de cette version pour s'adapter à cette partie (ce serait considéré comme de la triche).

3.1 Le client

La classe `LindaClient` implante l'interface `Linda` et offre les mêmes services que la version centralisée, que ce soit la quasi séquentielle ou la parallèle. Chaque opération de l'interface s'adresse par le protocole RMI à un objet serveur distant. Ceci nous a conduit aux différents choix suivants :

— Nous définissons une interface `RemoteLinda` qui offre les mêmes services que `Linda` mais est aussi de nature `Remote` pour se conformer aux exigences de la technologie RMI. Nous avons recopié le contenu de l'interface `Linda` pour ajouter à chaque opération l'exception `RemoteException` imposée par la technologie RMI. Il était en effet interdit de modifier directement l'interface `Linda`, de plus cela aura mis dans le monde RMI une version qui ne devait pas l'être, la version centralisée. Les deux interfaces ont un rôle similaire mais pas le même, l'une s'adresse à l'utilisateur de linda, l'autre est un intermédiaire interne à l'implantation distribuée. Nous aurions probablement pu automatiser ceci en utilisant les annotations et la réflexivité comme nous l'avons étudié dans le cours de métaprogrammation de l'UE Génie du Logiciel et des Systèmes (nous y avons étudié l'implantation de JUnit

pour automatiser la gestion des tests). Nous aurions aussi pu choisir de changer complètement les interfaces des opérations pour cette version interne.

— Nous devons gérer les actions asynchrones dans un monde distribué ce qui nous conduit à la question : où doivent s'exécuter les actions ? Deux réponses sont possibles, au sein du client, ou au sein du serveur. Il nous semble qu'il n'y a pas de meilleurs réponses, cela dépendra des opérations effectuées par les actions, soit elles communiquent principalement avec le reste de l'application et il est préférable qu'elles s'exécutent sur le client, soit elles communiquent principalement avec linda et il est préférable qu'elles s'exécutent sur le serveur. Nous avons choisi l'exécution sur le client. Il est donc nécessaire que les actions deviennent des objets distants pour le serveur qui lancera leur exécution. Nous avons donc défini :
  — une interface `RemoteCallback`,
  — une implantation associée `RemoteCallbackImpl` côté client,
  — et une passerelle `RemoteCallbackAdapter` qui sera enregistrée côté serveur et contactera l'objet distant `RemoteCallbackImpl` qui exécutera l'action de type `Callback` transmise par l'utilisateur de linda.

  Lorsque le client linda reçoit une demande d'enregistrement d'action :
  — il créé un objet local `RemoteCallbackImpl`,
  — le transmet au serveur `RemoteLindaImpl`,
  — celui-ci crée un objet `RemoteCallbackAdapter` qui appellera l'objet distant `RemoteCallbackImpl` quand l'action sera exécuté sur le serveur en lui transmettant le tuple identifié.

3.2 Le serveur

La classe `LindaServer` est indépendante de l'interface `Linda`. Elle n'offre pas directement les services linda à une utilisateur. Elle crée un objet `RemoteLindaImpl` qui implante l'interface `RemoteLinda` présenté dans la section précédente. Elle initialise un serveur de nom RMI (registry) et enregistre cet objet avec le nom imposé par le sujet `rmi://adresse_DNS_du_serveur/LindaClient`.

Pour implanter l'espace des tuples au sein de la classe `RemoteLindaImpl`, nous nous appuyons sur la version centralisée. Les différentes opérations de l'interface `RemoteLinda` appellent alors directement les opérations de la version centralisée exploitée.

4 Applications & Tests

4.1 Tests supplémentaires

Les enseignants avaient fourni quelques tests élémentaires `BasicTest1`, `BasicTest2`, `BasicTestCallback`, `BasicTestAsyncCallback`. Ceux-ci nous ont aidé à mettre au point notre implantation, mais, comme indiqué dans le sujet, ils n'étaient pas suffisants.

Nous avons ajouté des tests élémentaires plus simples :

— Avec une seule activité (paquetage : `linda.test.shm.synchronous` et `linda.test.shm.asynchronous`) :
  — Une écriture, puis une lecture ;
  — Une écriture, puis une prise ;
  — Plusieurs écritures, puis une lecture ;
  — Plusieurs écritures, puis une prise ;
  — Plusieurs écritures, puis plusieurs lectures élémentaires ;
  — Plusieurs écritures, puis plusieurs prises élémentaires ;
  — Plusieurs écritures, puis plusieurs lectures en une seule opération ;
  — Plusieurs écritures, puis plusieurs prises en une seule opération ;
  — Une écriture, puis enregistrement d'une lecture ;
  — Une écriture, puis enregistrement d'une prise ;
  — Plusieurs écritures, puis enregistrement d'une lecture ;
  — Plusieurs écritures, puis enregistrement d'une prise ;
  — Plusieurs écritures, puis enregistrement de plusieurs lectures élémentaires ;
  — Plusieurs écritures, puis enregistrement de plusieurs prises élémentaires ;
  — Enregistrement d'une lecture, puis une écriture ;
  — Enregistrement d'une prise, puis une écriture ;
  — Enregistrement d'une lecture, puis plusieurs écritures ;
  — Enregistrement d'une prise, plusieurs écritures ;
  — Enregistrement de plusieurs lectures élémentaires, puis plusieurs écritures ;
  — Enregistrement de plusieurs lectures élémentaires, puis plusieurs écritures ;
— Avec plusieurs activités (paquetage : `linda.test.shm.synchronous.concurrent` et `linda.test.shm.asynchronous.concurrent`) :
  — Une écriture et une lecture ;
  — Une écriture et une prise ;
  — Plusieurs écritures et une lecture ;
  — Plusieurs écritures et une prise ;
  — Plusieurs écritures et plusieurs lectures élémentaires ;
  — Plusieurs écritures et plusieurs prises élémentaires ;
  — Une écriture et enregistrement d'une lecture ;

— Une écriture et enregistrement d'une prise ;
— Plusieurs écritures et enregistrement d'une lecture ;
— Plusieurs écritures et enregistrement d'une prise ;
— Plusieurs écritures et enregistrement de plusieurs lectures élémentaires ;
— Plusieurs écritures et enregistrement de plusieurs prises élémentaires ;

Ces tests élémentaires nous ont permis de détecter certaines erreurs et de les corriger puis d'obtenir une plus grande confiance dans notre implantation et enfin de mesurer les performances de la version parallélisée par rapport à la version quasi séquentielle. Nous avons fait des mesures sur un espace de 65536 tuples avec des opérations de lecture et prise tuple par tuple et nous avons observé des améliorations d'un facteur entre 3 et 4 en découpant l'espace des tuples en 16 morceaux avec une activité par morceau. Il n'est pas utile d'avoir plus d'une activité par morceau car nous faisions une seule lecture. Cela montre l'intérêt du parallélisme de donnée pour notre exemple.

Nous avons ajouté un test plus complexe avec beaucoup d'accès parallèle `ComplexText` qui nous permet de vérifier que les opérations sont bien coordonnées. Celui-ci comporte une parallélisation sur une trentaine d'activités de lectures bloquantes suivi d'une écriture qui débloque une lecture qui écrit à son tour en cascade ; des actions asynchrones et des écritures en parallèle pour vérifier que `take` supprime bien le tuple envoyé.

Nous avons exécuté ces tests avec nos différentes versions : centralisée quasi séquentielle, centralisée concurrente, distribuée. Par contre, nous n'arrêtons pas systématiquement les versions concurrentes car cela demanderait une coordination plus complexe que l'objectif du test.

4.2 Calcul des nombres premiers inférieurs à K

Cet exemple se focalise sur la conception d'un algorithme parallèle. Il s'agit de produire l'ensemble des nombres premiers inférieurs à une borne donnée en paramètre, au lancement de l'application.

4.2.1 Séquentielle : Crible d'Ératosthène

Le principe du crible d'Érathosthène que nous avons implanté dans l'UE Technologies Objets consiste à construire un ensemble des nombres entiers naturels de 2 à K (car 1 est un nombre premier particulier) puis à répéter les opérations suivantes :

— prendre le minimum de cet ensemble ;
— extraire ce minimum qui est premier et ses multiples inférieurs à K qui ne sont pas premiers.

jusqu'à ne plus pouvoir extraire d'éléments.

Nous pourrions implanter cet algorithme de la même manière en utilisant l'espace de tuples comme moyen de stockage mais la recherche du minimum sera relativement coûteuse si l'espace des tuples ne nous offre pas ce service. Dans le cadre de l'UE Technologies Objets, nous avions utilisé un ensemble trié pour éviter ce problème.

Pour implanter ceci de manière séquentielle, nous proposons de :

— ajouter à l'espace de tuples tous les entiers naturels entre 2 et K ;
— enlever de l'espace de tuples tous les multiples des entiers naturels entre 2 et $\sqrt{K}$.

L'espace de tuples contiendra alors tous les nombres premiers entre 2 et K que nous pourrons consulter ensuite.

Notons que les demandes de suppression des multiples apparaîtrons autant de fois qu'il y a de multiplicateurs dans leur décomposition sous la forme de produits. Si nous utilisons la prise bloquante, nous serons bloqué lors de la seconde prise. Il est donc préférable d'utiliser la prise non bloquante, ou d'enregistrer une action de suppression qui ne sera éventuellement pas déclenchée.

4.2.2 Concurrente

Si nous voulons paralléliser cet algorithme, nous exécutons en parallèle la suppression des multiples : Une activité principale essaie de lire les tuples de 2 à $\sqrt{K}$ et crée pour chaque nombre qu'elle obtient ainsi une activité qui va se charger d'en enlever tous les multiples.

Pour savoir à quel moment nous avons fini, nous utilisons un groupe d'activités qui permet de bloquer l'activité principale jusqu'à que toutes les autres aient fini. Cela nous permet d'évaluer les performances de l'algorithme.

Au niveau des performances, pour un petit nombre d'activité et K de l'ordre de 10 000, la version séquentielle est plus rapide que la version concurrente, elle ne prends que quelques secondes au lieu de quelques dizaines de secondes.

Néanmoins, dès que nous passons à des ordres de grandeur supérieur, la tendance s'inverse : pour $K = 10^6$ et 1000 activités, la version concurrente fini en environ une minute alors que la version séquentielle n'est toujours pas finie plus d'une heure après.

4.3 Recherche approximative dans un fichier

L'archive contient une petite application de recherche d'un mot par distance Levenshtein. Cette application s'appuie sur le linda centralisé. Elle est constituée de deux activités, le `Manager` et le `Searcher`. Le manager dépose les données à explorer et la requête à traiter, l'activité de recherche lit une requête et parcourt les données. La requête est considérée comme traitée quand l'activité de recherche a fini. Telle que fournie, cette application suppose qu'une seule requête est déposée, et qu'une seule activité de recherche existe. Nous avons étendue cette application pour permettre :

1. d'avoir plusieurs activités de recherche (qui traitent concurremment la même requête) ;
2. d'avoir plusieurs activités manager (qui déposent plusieurs requêtes) ;

3. Le démarrage dynamique et l'arrêt arbitraire des chercheurs. Nous utilisons un attribut de classe atomique et un `Callback` en mode `take` pour récupérer le signal qu'un chercheur s'est arrêté. Quand l'attribut compteur tombe à zéro, nous savons qu'il n'y a plus de chercheurs disponibles et nous signalons aux managers de s'arrêter.
4. le retrait de la recherche par le manager après un certain délai (5 secondes) alors que les chercheurs n'ont pas fini. Plutôt que de réinitialiser les chercheur en changeant leurs paramètres, il a été plus simple de détruire les chercheurs assignés à une recherche avortée et de créer des chercheurs neufs à partir du constructeur.

Les activités communiquent entre elles à partir de `Callback` en mode `take` et de tuples avec les types `Code.Signal` et `SignalID` qui permettent de différencier l'ajout d'un chercheur et l'arrêt, voulu ou non, d'un chercheur, et l'arrêt d'un manager.

Faute de temps, nous n'avons pas pu vérifier toutes les modifications demandées, ni implémenter la version avec un linda en client/serveur avant la remise de la version officielle. Nous essaierons quand même de compléter cette application avant la séance de test.

4.4 Tableau blanc

Cette application fournie par les enseignants nous a permis de valider la version distribuée en ajoutant dynamiquement de nouveaux clients à l'espace des tuples distant.