

POP-Java: Parallélisme et distribution orienté objet

Beat Wolf, Pierre Kuonen, Thomas Dandekar

▶ **To cite this version:**

Beat Wolf, Pierre Kuonen, Thomas Dandekar. POP-Java: Parallélisme et distribution orienté objet. ComPAS 2014 : conférence en parallélisme, architecture et systèmes, Apr 2014, Neuchâtel, Suisse. hal-00988568

HAL Id: hal-00988568

https://inria.hal.science/hal-00988568

Submitted on 12 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ComPAS'2014 : Parallélisme / Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

POP-Java : Parallélisme et distribution orienté objet

Beat Wolf[1], Pierre Kuonen[1], Thomas Dandekar[2]
[1]iCoSys, Haute École Spécialisée de Suisse occidentale, Fribourg
[2]Biozentrum, Universität Würzburg

beat.wolf@hefr.ch, pierre.kuonen@hefr.ch, dandekar@biozentrum.uni-wuerzburg.de

Résumé
Cet article présente l'intégration du modèle de programmation POP pour *Parallel Object Programming*, dans le langage de programmation Java. Le modèle POP permet de créer des objets dans un environnement distribué et de les accéder d'une manière parallèle et transparente pour le programmeur. Ce travail se base sur les travaux déjà faits dans POP-C++, une implémentation du modèle POP en C++. À travers un exemple concret, les performances et fonctionnalités de POP-Java sont présentées et validées.

Mots-clés : Parallélisme, Distribution de calcul, Java, Langage de programmation OO

1. Introduction

Les processeurs sont proches d'atteindre leur limite de performance séquentielle et les futures technologies pour l'obtention de grandes puissances de calcul sont clairement la parallélisation et la distribution des calculs.
Durant les années 90, le parallélisme massif et la distribution des calculs étaient des sujets importants de recherche. Toutefois ils restaient principalement cantonnés dans le cercle très fermé du milieu du calcul de haute performance (HPC : High Performance Computing) parce qu'ils exigeaient l'utilisation de matériel spécifique. Aujourd'hui, en particulier grâce à la popularisation des architectures multi-coeurs et à l'augmentation croissante de la performance des réseaux tout un chacun peut disposer, pour des coûts relativement faibles, d'un petit superordinateur. Si le matériel a grandement évolué depuis les années 90, il n'est pas de même des outils de programmation pour ces environnements. Aujourd'hui encore, pour la distribution du calcul, c'est un standard défini au début des années 90 qui est le plus utilisé, à savoir MPI (Message Passing Interface) [2]. Depuis, d'autres outils de programmation sont apparus permettant de tirer profit des architectures multi-coeurs, comme OpenMP [3], mais ils ont été conçus de manière totalement indépendante des outils existants comme MPI. Si aujourd'hui tout un chacun peut disposer d'un environnement de calcul puissant alliant multi-coeur et multi-machine, la programmation d'un tel environnement reste une tâche difficile exigeant la connaissance et l'utilisation de plusieurs modèles et outils de programmation. Enfin, si des outils comme MPI ou OpenMP sont bien acceptés dans le milieu du HPC, ils le sont beaucoup moins hors de ce milieu ou la programmation de systèmes parallèles se fait encore beaucoup à l'aide de processus concurrents (*threads*). Disposer d'un modèle de programmation unique et de haut niveau d'abstraction permettant de tirer efficacement parti à la fois des architectures multi-coeur (parallélisme) et des architectures multi-machine (distribution) faciliterait grandement l'accès au

calcul de haute performance en dehors des milieux spécialisés. Dans ce papier nous présentons comment un modèle de programmation appelé POP pour *Parallel Object Programming*, permettant d'exploiter ces deux sources de performance, a été implémenté dans la langage Java. Les performances et la complexité du code obtenu sont comparées, sur un exemple concret du domaine de la bioinformatique, avec celles obtenues en utilisant les threads et RMI [10] de Java.

2. Le modèle POP

Le modèle de programmation POP, proposé par P. Kuonen et al. [15], est basé sur l'idée très simple que les objets sont des structures adaptées pour distribuer des données et du calcul dans des environnements distribués hétérogènes. Pour ce faire un nouveau type d'objet, appelé *objet parallèle* dans la nomenclature du modèle POP, a été introduit. Ces objets parallèles communiquent entre eux par appels de méthodes distantes. Afin de permettre le contrôle du parallélisme ainsi que l'ordre d'exécution des méthodes à l'intérieur d'un objet, six manières différentes d'appeler une méthode distante (appelées *sémantique* dans la terminologie POP) ont été définies. Une de ces sémantiques est la sémantique dite : *concurrente*. Elle permet, de manière naturelle et avec un niveau d'abstraction élevé, de créer des processus concurrents. La capacité du modèle POP d'exprimer de la concurrence est en fait un effet de bord d'une caractéristique, qui n'avait pas été initialement conçue précisément à cette fin. Toutefois, la conséquence est que l'on dispose maintenant, à l'intérieur d'un même modèle de programmation, de la possibilité d'exploiter à la fois le parallélisme (multi-coeur) et la distribution (multi-machine). Une description détaillée du modèle POP peut être trouvée dans l'article [16]. Nous en donnons ci-dessous une brève description suffisante pour comprendre le présent article.
Comme indiqué précédemment, le modèle POP est un modèle de programmation qui permet de créer des applications distribuées selon le paradigme de la programmation orientée objets. Il permet de créer des objets d'une manière transparente sur des machines distantes. Ces objets ne peuvent pas partager des données (les attributs publics sont interdits), ils communiquent donc par appels de méthodes distantes. Du point de vue de l'appelant de la méthode, l'appel distant peut être *synchrone* ou *asynchrone*. L'appel synchrone correspond à la situation classique ou l'appelant est bloqué durant l'exécution de la méthode distante. Dans le cas asynchrone, l'appelant n'est pas bloqué lors de l'appel et peut continuer son exécution en parallèle de l'exécution de la méthode distante. Du point de vue de l'objet appelé la méthode peut être *séquentielle*, *concurrente* ou *mutex*. Le premier cas correspond à la situation classique ou les différents appels s'exécutent dans l'ordre qu'ils arrivent sur l'objet. Les méthodes concurrentes peuvent être exécutées dès qu'elles arrivent sur l'objet. Ce sont elles qui permettent de créer des processus concurrents. Enfin les méthodes mutex ne peuvent être exécutées que lorsque toutes les méthodes précédemment arrivées sur l'objet ont été exécutées. Les six sémantiques d'appel de méthodes du modèle POP correspondent aux six combinaisons possibles de synchrone/asynchrone avec séquentielle/concurrent/mutex.
Le modèle POP a été implémenté comme une extension du langage C++ pour donner l'outil *POP-C++* [4]. Pour ce faire, six nouveaux mots clé ont été introduits dans C++. Le premier de ces mots clé est `parclass`. Il est utilisé à la place du mot clé `class` pour indiquer la déclaration d'une *classe parallèle*. Tout instance d'une classe parallèle est un objet parallèle qui s'exécutera à distance. Les cinq autres mots clé servent à indiquer la sémantique des méthodes d'une classe parallèle. Il s'agit de `sync` et `async` pour indiquer si la méthode est synchrone ou asynchrone et de `seq`, `conc` et `mutex` pour indiquer si la méthode est séquentielle, concurrente ou mutex. Finalement tout constructeur d'une classe parallèle peut être suivi d'une indication, introduite par `@od.`, permettant au programmeur de donner des indications sur les caractéristiques de

la machine à utiliser pour exécuter cet objet lorsque ce constructeur est utilisé pour instancier l'objet. Ces caractéristiques peuvent, par exemple, être la puissance de calcul désirée, la taille mémoire ou même l'adresse exacte de la machine.
Un exemple de déclaration d'une classe POP-C++ est donné dans le listing 1. Cette classe implémente une version très simple du type entier ne possédant que l'opération d'addition. Dans cet exemple, la classe parallèle possède deux constructeurs qui illustrent deux façons d'utiliser les indications permettant de donner les caractéristiques de la machine. Le premier constructeur indique, de manière statique, que l'objet doit être créé sur la machine locale (`localhost` = même machine que le créateur de l'objet). Le deuxième constructeur a un paramètre de type `char*`, qui permet de définir, de manière dynamique à l'exécution, l'adresse de la machine sur laquelle l'objet doit être créé.
Conformément à la définition du modèle POP, la méthode `get()` doit être synchrone car elle retourne une valeur. Par contre les méthodes `set()` et `add(Integer i)` peuvent être asynchrones car elles ne retournent pas de résultat. Il a été décidé, pour cet exemple, que l'ordre dans lequel les différents appels à `get()` sont exécutés n'était pas important ou, en d'autre termes, qu'une méthode `get()` pouvait être exécutée dès que l'appel était reçu et ceci indépendamment de l'ordre d'arrivée des autres appels. Elle est donc déclarée concurrente. Par contre, on désire que les appels à la méthode `set()` soient exécutés dans l'ordre où ils arrivent sur l'objet. La méthode a donc été déclarée séquentielle. Enfin la méthode `add(Integer i)` ne doit être exécutée que lorsque tous les appels de méthodes (quel que soit la méthode) précédemment arrivés sur l'objet ont été exécutés. Elle a donc été déclarée `mutex`. Ce dernier choix est quelque peu arbitraire et a comme but principal d'illustrer les différentes sémantiques du modèle POP. POP-C++ est disponible en open source à [4].

Listing 1 – Code POP-C++ implémentant une classe Integer accessible à distance

```
parclass Integer
{
  public:
        Integer()@{od.url("localhost");};
        Integer(char* url)@{od.url(url);};

        sync conc int Get();
        async seq void Set(int val);
        async mutex void Add(Integer &i);

  private:
        int value;
};
```

3. État de l'art

Java est un langage très populaire dans le milieu académique et industriel et il dispose d'un écosystème très riche. Depuis sa sortie en 1995, de nombreuses librairies et extensions ont été développées. Ceci sous la forme de projets comme Lombok [5] qui ajoutent des fonctionnalités au langage au travers des annotations ou de langages entièrement différents comme Scala [17] mais qui tournent dans la même machine virtuelle. Diverses librairies ou extensions du langage ont été développées pour la distribution de calcul au travers d'un réseau. Un premier exemple

est la fonctionnalité intégrée dans Java, à savoir le RMI (Remote Method Invocation) [10]. RMI offre la possibilité de rendre un objet Java accessible à distance. Une fois la connexion à cet objet faite au travers d'un proxy, les appels de méthodes se font comme sur un objet local. Bien que RMI résolve le problème de la distribution des objets à travers le réseau, il n'y existe aucune notion de parallélisme. Les outils traditionnels de Java, comme les threads doivent être utilisés à cet effet. Dans le modèle POP l'utilisation de threads est abstraite grâce à la sémantique des méthodes concurrentes. Plusieurs instances d'une ou de différentes méthodes ayant la sémantique concurrente peuvent s'exécuter en même temps.
Une autre approche pour la distribution d'objets Java à travers un réseau d'ordinateurs est l'outil ProActive [6]. ProActive offre une librairie complète pour distribuer des objets d'une manière transparente sur différentes machines. La différence majeure entre le modèle POP et le modèle ProActive réside dans le fait que ProActive est basé sur la notion d'*objets actifs*, ce qui n'est pas le cas du modèle POP. Il est impossible de dire lequel de ces choix est le plus judicieux. Il s'agit presque d'un choix *philosophique* fortement liés à la vision qu'à le concepteur du langage de ce qu'est un objet distant. Dans le cas du modèle POP, le choix ne pas se baser sur des objets actifs a été guidé par le fait que l'on désire faciliter l'accès à ce modèle en introduisant un minimum de notions nouvelles et en restant au plus proche du paradigme orienté objet classique. Or, que ce soit en C++ ou en Java, les objets ne sont pas actifs.
Signalons encore les approches Hadoop [18] et Akka [20] qui sont très populaires en ce moment. Hadoop est basé sur le modèle de calcul MapReduce [9] qui s'approche plus d'une approche fonctionnelle que d'une approche orientée objet. Akka adresse la problématique de la distribution de calcul au travers du modèle d'acteurs [11]. Ce modèle se base sur des acteurs distribués qui traitent et envoient des messages pour communiquer.

4. POP-Java

Pour intégrer le modèle POP dans le langage Java, il faut essentiellement introduire deux notions. La déclaration d'objets distants qui seront accédés d'une manière transparente à travers le réseau et la déclaration des sémantiques des différentes méthodes des objets distants. Pour ce faire, deux approches principales ont été identifiées. La première est l'ajout de mots clé dans le langage Java, comme cela est fait dans POP-C++. L'avantage de cette approche est de garantir un maximum de similarité avec le langage POP-C++, ce qui rend le transfert de connaissances entre les deux langages plus simple. La deuxième approche est d'utiliser au maximum les possibilités offertes par le langage Java, en particulier les annotations introduites dans Java 5. Cette approche a l'avantage de permettre de garder le code source compatible avec le langage Java et ainsi de rendre possible l'utilisation des environnements de programmation Java existants. Un premier prototype de POP-Java, qui utilisait l'approche avec les mots clé, a été réalisé par Valentin Clément [8]. Un parseur basé sur JavaCC a été créé pour analyser les nouveaux mots clé POP-Java et ensuite générer du code Java pur. Durant la phase de réalisation du prototype final de POP-Java, la décision d'utiliser des mots clé a été révisée pour, finalement, retenir l'approche basée sur les annotations. La raison principale de cette décision est de faciliter la prise en main du langage par des développeurs Java et de permettre une meilleure intégration de POP-Java dans l'écosystème Java.

4.1. Annotations POP-Java

Le tableau ci-dessous donne une vue synthétique des relations entre les annotations de POP-Java et les mots clé équivalents en POP-C++. Pour des questions de lisibilité du code, les cinq mots clé de POP-C++ qui décrivent la sémantique des méthodes (`sync`, `async`, `conc`, `seq`

et `mutex`), qui sont expliquées dans le chapitre 2, ont été remplacés par 6 annotations correspondant aux 6 combinaisons possibles de ces mots clé. L'indication `@od.` de POP-C++ a été remplacée avec deux annotations permettant de différencier entre le cas statique et dynamique. Le cas statique correspond à la situation où l'indication n'est pas paramétrée alors que la cas dynamique correspond à la situation où l'indication dépend d'un ou plusieurs paramètres du constructeur.

POP-C++	**POP-Java**
`parclass`	@POPClass
`sync conc`	@POPSyncConc
`sync seq`	@POPSyncSeq
`sync mutex`	@POPSyncMutex
`async conc`	@POPAsyncConc
`async seq`	@POPAsyncSeq
`async mutex`	@POPAsyncMutex
`od.url("localhost")`	@POPObjectDescription(url="localhost")
`od.url(url)`	@POPConfig(Type.URL)

Le listing 2 contient en POP-Java la même déclaration de la classe `Integer` donnée en POP-C++ dans le listing 1.

Listing 2 – Code POP-Java implémentant une classe Integer accessible à distance

```
@POPClass
public class Integer {
   private int value;

   @POPObjectDescription(url = "localhost")
   public Integer(){}

   public Integer(@POPConfig(Type.URL) String url){}

   @POPSyncConc
   public int get(){
      return value;
   }

   @POPAsyncSeq
   public void set(Integer i){
      value = i.get();
   }

   @POPAsyncMutex
   public void add(Integer i){
      value += i.get();
   }
}
```

Pour plus de détails sur les annotations POP-Java, leurs sémantiques et leurs utilisations, veuillez consulter le site web de POP-Java [8].

4.2. Mémoire

Comme indiqué précédemment le modèle POP est, à la base, un modèle à mémoire distribuée, les objets distants ne peuvent donc pas partager de données (pas d'attributs publics). La transmission d'information (ou données) entre objets distants ne peut se faire qu'au travers des paramètres des méthodes ou des valeurs retournées par ces méthodes. Pour ce faire une donnée complexe qui doit être transférée d'un objet distant à un autre doit être une instance d'une classe qui implémente l'interface `IPOPBase` fournie par l'environnement POP-Java. Cette interface définit les méthodes `serialize` et `deserialize` qui permettront d'empaqueter l'objet dans un message pour être envoyé à l'objet distant. Il est à signaler que pour garantir le même comportement d'un objet, exécuté localement ou à distance, chaque objet POP-Java est démarré et exécuté dans sa propre JVM (Java virtual machine). Le problème du ramasse miette pour les objets distribués est résolu grâce à des compteurs de références. Chaque objet possède un compteur qui compte le nombre de références vers lui. Un objet termine son exécution au moment où plus aucun autre objet ne le référence (compteur de référence nulle).

5. Implémentation

Le chapitre suivant décrit de quelle manière le langage POP-Java présenté dans le chapitre précédant a été implémenté.

5.1. Création d'objets à distance

Pour comprendre l'implémentation de POP-Java il faut dans un premier temps comprendre comment un objet POP-Java est créé à distance et comment il est accédé. Comme pour un objet Java classique, la création d'un objet POP-Java se fait au travers de l'appel à un de ses constructeurs. Cet appel est intercepté par l'environnement d'exécution de POP-Java et une connexion vers la machine à distance sur laquelle on désire faire tourner l'objet est alors ouverte. Comme présenté dans la section précédente, la machine sur laquelle l'objet parallèle est exécuté est déterminée, grâce à l'indication associée au constructeur utilisé pour construire l'objet. La connexion vers la machine distante qui exécutera l'objet est, dans la version actuelle de POP-Java, faite avec le protocole SSH, mais d'autres protocoles pourraient être utilisés. Sur la machine distante, un *Broker* POP-Java est lancé. C'est lui qui démarre l'objet distant. Sur la machine ou le constructeur de l'objet a été appelé, un objet *Interface* est créé. Cette interface se connecte sur l'objet distant au travers du Broker. À partir de ce moment, tout appel de méthode sur l'objet Interface est redirigé sur l'objet distant à travers le Broker. La figure 1 illustre ce processus. Il est important de noter que l'application voit et accède à l'objet distant comme s'il était exécuté localement dans la même machine virtuelle.

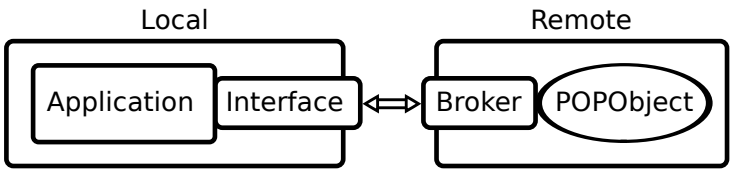


FIGURE 1 – Abstraction d'un objet distant à travers l'Interface et le Broker

Ce sont les classes Interface et Broker qui sont responsables de l'exécution des appels de méthodes. Ces appels sont exécutés par le Broker en assurant que les sémantiques définies par

le programmeur soient respectées. Les appels concurrents sont exécutés en utilisant un *FixedThreadPool* de Java, les méthodes séquentielles au travers d'un *SingleThreadExecutor* alors que les méthodes mutex sont, si aucune méthode est en cours d'exécution, exécutées depuis le thread principal du Broker. L'appel à une méthode d'un objet distant peut être fait par l'objet qui l'a créé ou par un objet qui a reçu sa référence. En effet, les références à des objets parallèles peuvent être passées en paramètre de méthodes comme n'importe quel objet de Java. Ceci contraste avec RMI, qui utilise une approche de type *publication/souscription* (publish/-subscribe). Chaque objet distant, après sa création, doit être publié sur le réseau au travers d'un registre RMI pour permettre sa découverte et son accès. C'est grâce à cette abstraction de la communication entre les objets distants au travers d'un objet Interface et Broker que la compatibilité entre le langage POP-Java et POP-C++ peut être assuré. En d'autres termes un programme POP-Java peut créer et accéder un objet POP-C++, et inversement.

5.2. Compilateur

Le langage Java ne permet pas de manipuler le code source d'une classe au moment de la compilation ou de l'exécution. Bien que des projets comme Lombok [5] le fassent, ils se basent sur des fonctionnalités non documentées, qui peuvent cesser de fonctionner à tout moment. Bien qu'il soit possible d'écrire, en Java pur, du code qui utilise le modèle POP, nous nous sommes mis comme objectif que la programmation de ces objets doit être la plus simple possible, et ne pas nécessiter l'écriture de codes répétitifs. C'est pour cette raison que nous avons opté pour la méthode suivante : les codes sources des classes POP-Java sont enregistrées dans des fichiers spécifiques à POP-Java, indiqués par leur extension `.pjava` (Contrairement au `.java` des fichiers Java classiques). Durant le processus de compilation, un parseur, basé sur JavaCC, convertit le code source des classes POP-Java en code Java pur. En particulier la création d'un objet POP-Java doit être interceptée, pour créer une instance d'une interface locale et du Broker à distance, comme illustré sur la figure 1. Chaque instance de `new` dans le code source, qui fait référence à une classe POP-Java, doit être remplacée par une instruction spécifique POP-Java qui retourne un objet Interface qui intercepte l'accès sur l'objet POP-Java et le redirige sur l'objet Broker à distance. La création de l'objet Interface se fait grâce à la librairie Javassist [7] qui permet la création d'un proxy autour d'une classe durant l'exécution de l'application. Le processus de compilation d'une application POP-Java est illustré sur la figure 2.

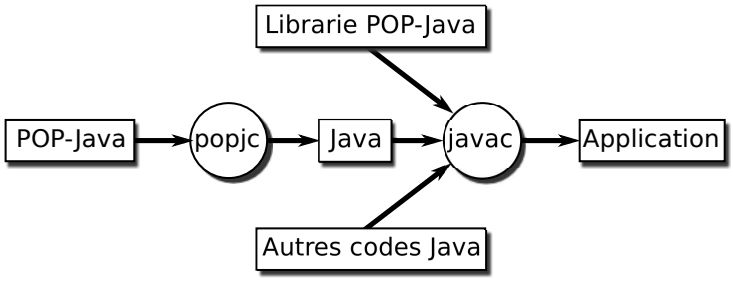


FIGURE 2 – Chaine de compilation de code POP-Java

Pour vérifier que le programmeur respecte les règles de programmation POP-Java, par exemple que toutes les méthodes publiques soient annotées avec les annotations POP-Java ou qu'il n'existe pas d'attributs publics, APT [1] (Java annotation processing tool) est utilisé. Depuis la version 6 de Java, cette fonctionnalité est intégrée nativement et permet de vérifier le code

source d'une classe au moment de la compilation. L'avantage d'utiliser cette fonctionnalité, au contraire d'intégrer cette détection d'erreurs dans le parseur POP-Java, est que les messages d'erreurs sont aussi affichés dans les environnements de développement Java, comme Eclipse [12].

5.3. Restrictions

Les restrictions de POP-Java par rapport à du code Java standard sont mineures. Les plus évidentes sont liées au modèle de mémoire de POP qui interdit d'avoir des attributs publics dans une classe POP-Java.

6. Validation

Dans ce chapitre, nous présentons un cas d'utilisation concret dans lequel POP-Java a été utilisé. À travers cet exemple nous comparons POP-Java et RMI sur un cas concret, du point de vue des performances et de la complexité du code source. Nous avons implémenté un aligneur de séquences ADN. L'alignement de séquences ADN contre une référence est un problème important en bioinformatique. Les techniques de séquençages d'ADN deviennent de plus en plus performantes et génèrent des quantités de données toujours plus grandes. Pour faire l'analyse de ces données dans un temps raisonnable la parallélisation est une approche intéressante. Durant le séquençage, l'ADN de l'organisme à analyser est coupé en petit morceaux. Ces morceaux sont ensuite digitalisés par le séquenceur et l'information sur l'ordre des séquences est perdue. Pour retrouver l'endroit le plus probable sur l'ADN de l'organisme pour chaque séquence digitalisée, on aligne ces séquences contre une séquence de référence. La séquence de référence est une séquence théorique censé représenter un spécimen moyen de cette espèce. La séquence de référence de l'humain a été générée grâce au *Human genome project* [13]. Un algorithme d'alignement a été distribué en utilisant les deux technologies RMI et POP-Java. C'est le même algorithme qui a été utilisé pour les deux implémentations. Les alignements produits sont donc identiques. Il est à signaler que POP-Java a été principalement conçu pour des applications avec un grand rapport calcul/communication. Ce n'est pas le cas de l'alignement de séquences d'ADN où de grandes quantités de données doivent être transmises, pour peu de calcul. Ce cas d'utilisation est donc parfait pour faire apparaître les éventuelles limites de POP-Java.

6.1. Algorithme

Il existe de nombreuses approches pour effectuer l'alignement d'ADN [14]. Dans notre cas, nous avons utilisé l'algorithme décrit dans l'article [19] dont nous avions réalisé une implémentation en Java pur. Cette implémentation utilise des threads et peut donc tirer parti de tous les coeurs d'une machine multi-coeurs. A partir de cette implémentation, nous avons dérivé deux autres implémentations. La première utilise RMI de Java pour distribuer le calcul sur plusieurs machines multi-coeurs. La seconde utilise POP-Java pour paralléliser le calcul sur les différents coeurs et distribuer le calcul sur plusieurs machines multi-coeurs. Elle ne fait donc pas appel à la technologie des threads. Ceci est possible grâce à l'utilisation des méthodes concurrentes du modèle POP. La structure générale de l'algorithme est la même pour les deux implémentations. Elle est basée sur le concept classique de producteur-consommateur. Un objet lecteur lit les séquences à aligner dans un fichier source, les envoie aux travailleurs qui se trouvent sur plusieurs machines mutli-coeurs différentes et un objet rédacteur récupère les résultats pour les écrire dans un fichier de résultats. La figure 3 illustre cette architecture.
L'implémentation en POP-Java a nécessité l'écriture de trois classes spécifiques à POP-Java. La

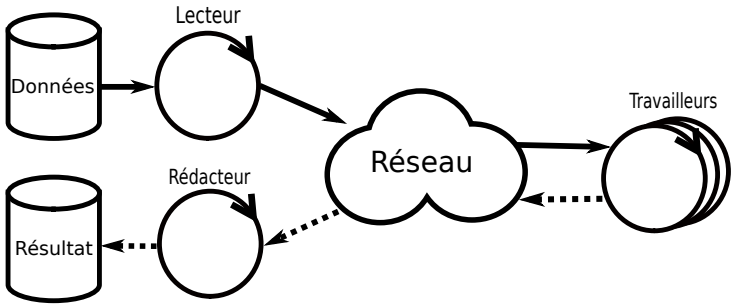


FIGURE 3 – Architecture générale d'algorithme d'alignement de séquences ADN

première qui contient la fonction principale de l'application (`main`) et les deux autres classes sont ceux qui décrivent les *travailleurs* et le *rédacteur*. La partie du code qui fait effectivement l'alignement a été extraite de l'implémentation de base et insérée dans des classes Java qui ne s'occupent pas de la parallélisation et de la distribution du calcul. La manière de programmer la distribution de l'algorithme dans les deux implémentations consiste principalement dans la création de *wrapper* qui sont responsables de faire les communications entre les différentes machines. Dans le cas de POP-Java, ces wrapper sont très petits et simples à écrire car le modèle POP rend la distribution des différents objets sur plusieurs machines très simple. Signalons qu'avec cette version il est possible, sans changement de code, d'adapter la répartition de la charge de travail en fonction de l'infrastructure disponible. Par exemple, on peut mettre plusieurs objets sur la même machine au cas où les différentes machines ne seraient pas toutes de la même puissance (environnement hétérogène). Dans le cas de RMI, le code à développer est plus conséquent. Comme mentionné préalablement, la distribution du travail dans la version RMI sur plusieurs coeurs d'une machine se fait au travers de threads, ce qui alourdit le code source. De plus, toute la gestion d'envoi et réception de données d'une manière efficace doit être implémentée à la main, une fonctionnalité intégré dans POP-Java avec les filles d'attentes des appels à distance.

6.2. Évaluation

Pour évaluer les deux implémentations, deux aspects ont été comparés. La complexité du code et la performance. Il est à préciser que notre objectif est de comparer, sur un nombre restreint de machines et de coeurs, le comportement de POP-Java par rapport à RMI et aux threads. Une étude plus poussée du passage à l'échelle de POP-Java sur un grand nombre de machines fera l'objet d'un travail ultérieur.

Comme l'implémentation originale en Java utilisait déjà une approche producteur-consommateur, les changements de code nécessaires pour la distribution du calcul devraient, en principe, être minimaux. La version écrite en POP-Java utilise plus de 50 classes Java représentant environ 12000 lignes de code. Seules trois de ces classes sont des classes POP-Java. Elles représentent un total d'environ 300 lignes de code, dont seulement un petit pourcentage est spécifique à POP-Java principalement constituées d'annotations. L'implémentation en RMI contient également plus de 50 classes, dont 5 sont spécifiques à la distribution du calcul représentant plus de 1200 lignes de code. A cela s'ajoute le fait que pour faire une parallélisation à plusieurs niveaux, c'est à dire, au niveau des différents coeurs (parallélisation) et au niveau des différentes machines (distribution), l'implémentation RMI nécessite la gestion de threads ce qui n'est pas le cas de POP-Java, grâce à l'utilisation de la sémantique concurrente. POP-Java est donc très compétitif au niveau de la complexité du code et permet d'écrire des applications parallèles et distribuées

plus rapidement et plus aisément qu'avec les technologies classiques des threads et de RMI. Pour évaluer les performances des deux implémentations, nous avons utilisé six machines à quatre coeurs cadencés à 3.4 GHz et contenant 8 GO de mémoire vive. L'hyperthreading a été désactivé. Ces machines sont connectées par un réseau de 1 Gb/s. Le set de données test est composé de 16 millions de séquences ADN générées artificiellement. Elles ont été alignées contre le chromosome humain 7 en utilisant entre une et six machines. Afin d'avoir des résultats cohérents et reproductibles, les objets *lecteur* et *rédacteur* sont exécutés sur une machine dédiée, le calcul lui-même étant fait sur d'autres machines (de 1 à 6 machines). Il est à noter que la version RMI a été fortement optimisée pour ce cas d'utilisation précis, afin d'avoir un point de comparaison valable pour l'évaluation des performances de la version POP-Java.

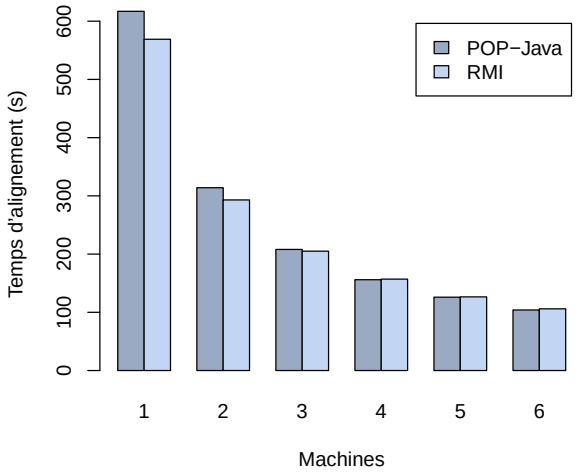


FIGURE 4 – Temps d'execution sur les machines

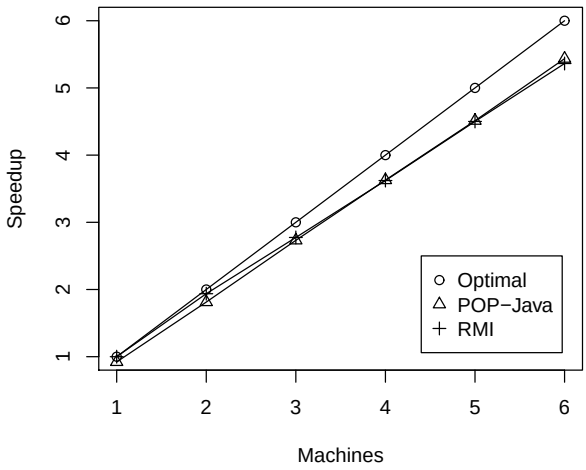


FIGURE 5 – Speedup sur six machines

La figure 4 montre le temps d'exécution de l'alignement avec les deux implémentations. Seul le temps de calcul pour l'algorithme lui-même a été mesuré, sans la partie séquentielle du chargement initial des données. La courbe d'accélération de la figure 5 est basée sur ces mêmes mesures. Le temps de référence pour le calcul de l'accélération est le temps d'exécution de la version RMI sur une seule machine. Globalement, on constate une légèrement meilleure performance pour la version RMI par rapport à la version POP-Java pour 1 à 4 machines. Sur une seule machine, l'implémentation RMI a un avantage d'environ 7% par rapport à POP-Java (617 secondes pour POP-Java, 569 secondes pour RMI). Sur six machines, cet avantage disparaît et les deux versions ont quasiment les mêmes performances avec 105 secondes contre 106 secondes. On voit une bonne accélération pour les deux implémentations, avec une accélération de 5.43 sur six machines pour l'implémentation POP-Java, ce qui correspond une efficacité de plus de 90%. L'implémentation utilisant RMI atteint une accélération de 5.36 sur six machines. Si l'on utilise comme valeur de référence, pour le calcul de l'accélération de la version POP-Java, le temps d'exécution sur une seule machine de la version POP-Java, on atteint une accélération de 5.89. Cela montre que, même pour une nombre restreint de machines, le passage à l'échelle est meilleur que l'implémentation RMI (5.36). On observe aussi qu'en augmentant le nombre de machines, les performances de l'implémentation en POP-Java se rapprochent et semblent même dépasser celle de RMI. Des tests sur un nombre plus grand de machines permettraient de s'assurer de la stabilité de cette tendance.

La légère perte de performance de la version POP-Java pour 1 à 4 machines s'explique par le fait que le modèle POP est un modèle de programmation utilisant un niveau d'abstraction élevé facilitant l'écriture du code. Une implémentation basée sur des concepts de plus bas niveau permet de faire des optimisations spécifiques au programme à exécuter. C'est en particulier le cas si l'on utilise des threads. Compte tenu du fait que POP-Java n'a pas été principalement conçu pour ce genre d'applications avec un rapport calcul/communication assez faible, nous sommes très satisfaits des résultats obtenus, d'autant plus que les tests de performances montrent que, déjà sur un nombre restreint de machines, le passage à l'échelle de la version POP-Java semble meilleur que celui de RMI.

7. Perspectives

Bien que POP-Java soit un prototype fonctionnel parfaitement utilisable pour développer des applications du monde réel, son développement est toujours en cours. Un des avantages majeurs de Java est sa portabilité grâce à l'utilisation de la machine virtuelle Java. Malheureusement, pour l'instant, cet avantage est perdu avec POP-Java car son environnement d'exécution est seulement supporté sur des plateformes à base d'UNIX comme Linux ou OSX. Nous espérons lever cette limitation dans une version future. L'utilisation d'annotations en POP-Java permet l'extension simple du langage, ce qui rend possible l'ajout de nouvelles fonctionnalités. Dans le futur, ces possibilités vont être analysées et, si nécessaire, améliorées. Les performances mesurées dans le cadre de ce travail sont plus qu'acceptables, toutefois notre intention est de continuer à les améliorer. En effet, POP-Java va être utilisé dans le cadre d'une thèse de doctorat pour améliorer les performances de calculs d'un outil commercial d'analyse de séquences d'ADN.

8. Conclusion

Dans cet article nous avons présenté la façon dont le modèle POP a pu être intégré dans le langage de programmation Java en utilisant la technique des annotations fournie par Java 5. L'outil ainsi créé, appelé POP-Java, facilite l'écriture d'applications parallèles et distribuées s'exécutant dans des infrastructures multi-machine et multi-coeur hétérogènes. Les performances de cet outil ont été comparées, sur une application d'alignement de séquences d'ADN, aux performances obtenues avec la même application développée à l'aide de threads et de RMI. POP-Java s'est montré plus adapté à l'écriture de cette application (code source plus court et plus simple) pour une performance équivalente voire meilleure. La dernière version du prototype POP-Java est disponible en open source à l'url : *http ://gridgroup.hefr.ch/popjava*

Bibliographie

1. Java apt, jsr 269. *https ://jcp.org/en/jsr/detail ?id=269*.
2. Mpi. *http ://www.mpi-forum.org/*.
3. Openmp. *http ://openmp.org/*.
4. Pop-c++ site web. *http ://gridgroup.hefr.ch/popc/*.
5. Project lombok. *http ://projectlombok.org*.
6. Baduel (L.), Baude (F.), Caromel (D.), Contes (A.), Huet (F.), Morel (M.) et Quilici (R.). – Programming, composing, deploying for the grid. *In : Grid Computing : Software Environments and Tools*. – Springer Verlag.
7. Chiba (S.). – Javassist - a reflection-based programming wizard for java. *In : International Business Machines Corp*, p. http ://www.javassist.

8. Clément (V.). – *POP-Java*. – Thèse, EIAFR, 2010.
9. Dean (J.) et Ghemawat (S.). – Mapreduce : simplified data processing on large clusters. *Commun. ACM*, vol. 51, n1, janvier 2008, pp. 107–113.
10. Grosso (W.). – *Java RMI*. – Sebastopol, "O'Reilly Media, Inc.", 2001.
11. Hewitt (C.), Bishop (P.) et Steiger (R.). – A universal modular actor formalism for artificial intelligence. *In : Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. pp. 235–245. – San Francisco, CA, USA, 1973.
12. Holzner (S.). – *Eclipse*. – Köln, O'Reilly Germany, 2004, 1. édition.
13. Lander (E. S.), Linton (L. M.), Birren (B.), Nusbaum (C.), Zody (M. C.), Baldwin (J.), Devon (K.), Dewar (K.), Doyle (M.), FitzHugh (W.) et al. – Initial sequencing and analysis of the human genome. *Nature*, vol. 409, n6822, 2001, pp. 860–921.
14. Li (H.) et Homer (N.). – A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, vol. 11, n5, 2010, pp. 473–483.
15. Nguyen (T. A.) et Kuonen (P.). – A model of dynamic parallel objects for metacomputing. *In : The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*.
16. Nguyen (T.-A.) et Kuonen (P.). – Programming the grid with pop-c ++. *Future Generation Computer Systems*, vol. 23, n1, Jan 2007, p. 23–30.
17. Odersky (M.), Spoon (L.) et Venners (B.). – *Programming in Scala -*. – Mountain View, Artima Inc, 2008.
18. White (T.). – *Hadoop - the Definitive Guide : MapReduce for the Cloud*. – Sebastopol, "O'Reilly Media, Inc.", 2012, 3. édition.
19. Wolf (B.) et Kuonen (P.). – A novel approach for heuristic pairwise dna sequence alignment. *In : Proceedings of the 2013 International Conference on Bioinformatics & Computational Biology*.
20. Wyatt (D.). – *Akka Concurrency -*. – California, Artima, Incorporated, 2013.