Projet Réseaux de Kahn

Yu-Guan Hsieh & Téo Sanchez

Résumé du projet

Le but de ce projet est de réaliser différentes implémentations de réseaux de Kahn à partir d'une interface donnée sous forme monadique (ce qui nous permet de manipuler des langages fonctionnels purs avec des traits impératifs).

Les réseaux de Kahn (Kahn Process Networks en anglais), sont un modèle de calcul distribué entre plusieurs processus communiquant entre eux par des files. Ce réseau a un comportement déterministe si les processus qui le composent sont déterministes.

Les différentes implémentations sont :

1. Une implémentation utilisant la bibliothèque de threads d'OCaml (donnée dans l'énoncé)
2. Une implémentation utilisant la bibliothèque de thread Lwt. Cette implémentation a servi de référence car Lwt contient dèjà des fonctions caractéristiques de la communication inter-processus.
3. Un implémentation utilisant des processus Unix communiquant entre eux avec des pipes grâce à la bibliothèque Unix d'OCaml.
4. Une implémentation séquentielle où le parallélisme est simulé (inspiré de l'implémentation par continuation de l'article *A Poor Man's Concurrency Monad*).
5. Une implémentation distribuée sur le réseau utilisant des sockets de la bibliothèque Unix d'OCaml

Les différentes implémentations

Threads d'OCaml (fournie) : `kahn_th.ml`

Dans cette implémentation, les processus sont représentés par des fonctions de type `unit -> 'a`. C'est une promesse qui nous rendra une valeur de type `'a` une fois que la fonction `run` est exécutée. Chaque processus individuel vit dans son propre thread.

Threads de Lwt : `kahn_lwt.ml`

Ici, les processus sont des threads de type `'a Lwt.t`, et la plupart des fonctions requises sont déjà implémentés dans le module Lwt : return, bind, doco (`Lwt.join`), run, get (`Lwt_stream.next`). L'implémentation est triviale.

Processus lourd du module Unix d'OCaml : `kahn_proc.ml`

De manière analogue aux threads d'OCaml, les processus sont une fonction `unit -> 'a`. On utilise des pipes pour les canaux, couplés avec des mutex. Ces derniers préviennent des problèmes liés aux ressources partagées entre les processus, qui sont des portions de codes que l'on appellent zones critiques. Ils font parties des techniques dites d'exclusion mutuelles, n'autorisant l'accès à la zone critique par un seul processus à la fois.

Les fonctions get et put utilisent le module `Marshal` qui permet d'encoder n'importe quelle structures de données en séquences de bytes, afin d'être envoyés sur les canaux pour être ensuite décodés par le processus destinataire.

Enfin, la fonction `doco` qui prend une liste de processus et les exécute. Avec `Unix.waitpid`, le processus entier est bloquant: tous les processus de la liste doivent finir avant de passer à la prochaine étape. La fonction renvoie `unit` une fois que tous les processus de la liste ont été exécutées.

Implémentation séquentielle, parallélisme simulé : `kahn_seq.ml`

On distingue le "vrai" parallélisme (où un ordinateurs multi-cœurs lance des processus sur plusieurs de ses processeurs) du parallélisme à temps partagé, où les threads s'exécutent sur un même processeur. Dans ce dernier cas, le parallélisme est simulé. Cette idée est incarnée par deux implémentations possibles :

La première consiste à traduire directement le code Haskell issu de l'article *A Poor Man's Concurrency Monad* en OCaml à quelques nuances près, même si la structure reste la même. Elle utilise la méthode d'*entrelacement* (*interleaving* en anglais) c'est à dire que le processeur va exécuter le début d'un thread, avant de le suspendre pour donner la main à un autre thread. Afin de relancer les thread suspendus au même endroit, on doit avoir accès à son "futur" appelé généralement sa *continuation*. Cette implémentation monadique invoque des fonctions qui prennent une *continuation* comme premier arguement.

Ainsi, les processus sont divisés en tranches appelés actions, et ces mêmes actions renvoient leur futur qui sont elles mêmes des actions :

```
type action =
  | Stop    (* Fin du processus*)
  | Action of (unit -> action)
  | Doco of action list
```

Un processus est alors de type `('a -> action) -> action`.

Une autre implémentation possible pour cette section est d'utiliser un type `('a -> unit) -> unit` comme proposé dans l'énoncé. Cela requiert alors d'avoir

une structure de donnée globale qui stocke ce qu'il reste à faire après chaque étape de l'exécution d'un processus.

Nous avons choisi ici la première implémentation issue de l'article "A poor Man's Concurrency Monad".

Implémentation distribuée sur le réseau : `kahn_network.ml`

Déscription Cette implémentation a pour objectif de distribuer les processus sur plusieurs ordinateurs et communiquant à travers le réseau via des sockets. On utilise les sockets implémentés dans le module Unix d'OCaml et les données sont transféréss avec le module `Marshal`.

On distingue:

- Les processus qui sont du type `CSet.t -> 'a * CSet.t`, avec `CSet` une structure de donnée créée avec le module `Set` d'OCaml, et qui permet de stocker les canaux ouverts à un moment donné. Un processus prend l'ensemble des canaux ouverts et renvoie les canaux ouvers après l'exécution du processus.
- Un canal est défini par son numéro de port `port_num : int`, le nom de son ordinateur hôte `host : string`, et le champs `sock : (sock * sock_kind) option` qui défini le sens de la communication (qui est le producteur et qui est le consommateur) et stocke le type abstrait de canal (qui se compose de `out_channel` et `in_channel` d'OCaml) si la connection a été établie.
- Une file des ordinateurs disponibles `computer_queue`, créée à partir du fichier *network.config* où l'on doit écrire la liste des ordinateurs que l'on souhaite utiliser.

La fonction `new_channel` crée deux threads. L'un communique avec les producteurs et l'autre avec les consommateurs au travers de sockets. Les deux threads communiquent également entre eux par un pipe : Le premier thread lit les données dans les sockets entre lui et les producteurs et les stocke dans le pipe tandis que le second thread lit les données qui sont mis dans le pipe et les mets dans les sockets vers les consommateurs quand il reçoit une requête de sa part (`GET`: mettre une valeur dans le canal; `GETEND`: la fin de communication).

La fonction `send_processes` assure la distribution des processus sur les différents ordinateurs de `computer_queue` et établit les connections grâce aux sockets (à travers la fonction `easy_connect`). Puis dans la fonction `doco`, grâce à l'utilisation de la fonction `Unix.select`, on surveille que les processus fils s'exécutent, et dans le cas contraire, il faut redistribuer le processus.

On précise aussi que sur chaque ordinatuer (ou encore mieux, sur chaque noeud de réseau car un programme peut être lancé plusieurs fois sur une même machine)

il y a un thread qui se charge d'accepter les distributions de processus et de créer des nouveau threads dans lesquels ces processus seront exécutés. Le choix d'utiliser des processus légers autorise ainsi l'existence des variables partagées au sein d'un même exemplaire(?) de programme.

La fonction put prend un élément `v` à envoyer, et l'ensemble des canaux ouverts, et vérifie s'il existe un canal concret dans la socket (elle le crée sinon). Elle utilise `Marshal` pour envoyer la valeur dans le canal et renvoie `unit` et le nouvel ensemble des canaux ouverts. La fonction `get` procède de manière analogue sauf qu'elle renvoie une valeur.

Les fonctions `commu_with_send` et `commu_with_recv` gèrent le remplacement des producteurs et des consommateurs respectivement, en appelant un nouveau client quand la connection est rompue ou lorsque le processus lui renvoie le signal `PutEnd` ou `GetEnd` signifiant la fin de la communication.

Utilisation Cette implémentation ne fonctionne qu'avec OCaml >= 4.03.0 de façon génerale (dans des cas particuliers, comme le fichier `int_printer_network.ml`, ça pourrait marcher avec OCaml 4.02.0).

Il y a plusieurs options dans la ligne de commande ,

- `-wait` : doit être utilisé sur tous les ordinateurs du réseau à l'exception de l'ordinateur principal qui devra être lancé à la fin pour démarre le programme.
- `-port` : spécifie le port à écouter (1024 par défaut)
- `-config` : permet de spécifier le nom du fichier contenant la liste des ordinateurs à utiliser (par défaut, ce fichier est *network.config*).

Le fichier *network.config* se présente sous le format suivant :

```
Computer1 [port1]
Computer2 [port2]
...
```

Le même ordinateur peut apparaître plusieurs fois si les ports sont différents.

Améliorations possibles Lorsque la communication avec un ordinateur est interrompue de manière inopinée, le programme peut ne plus continuer correctement puisque le processus distribué interrompu sera redémarré depuis le début. Par exemple, avec l'exemple de base de la génération d'entier, si la fonction d'affichage sur la sortie standard est interrompu, elle sera redémarré et continuera d'afficher les entiers qu'il reçoit. Le résultat ne sera donc pas altéré. A l'inverse, s'il s'agit de la fonction qui génère les entiers, après l'arrêt de ce processus, le programme affichera les entiers depuis le début.

Pour règler ce problème, on a essayé de demander le processus fils de renvoyer son futur pour chaque `bind` exécutée. Il y a des modifications à faire, et surtout le type d'un processus ne resterait plus le même, mais ça n'a pas abouti à cause du module `Marshal` qui ne prend pas en charge le type abstrait (même si on n'a pas pu identifier d'où vient ce type abstrait en jeu empêchant notre code de fonctionner).

Afin que les communications à travers des canaux soient plus robustes, des questions sont à poser. Comment vérifier que le message envoyé a été bien reçu (et par la bonne personne)? Comment affirmer que les message reçus viennent effectivment du programme en question (et que les messages sont corrects)? Les idées comme des clés d'authentification peuvent être considérées.

Il y a encore un autre petit soucis dans notre réalisation de la version réseau: l'incompatibilité entre la fonction `Unix.select` et le module `Graphics` d'OCaml. Un bloc `try ... with ...` a été adopté face à cette gêne.

En OCaml avec le module `Arg` le parsing de la ligne de commande ne peut s'effectuer que dans un seul endroit, ce qui nous pose de difficulté car on a besoin que ça soit fait plusieurs fois (une fois pour le foncteur `Choose_impl`, une fois pour la fonction `run` dans l'implémentation de réseau et enfin une fois dans le programme utilisateur). Pour contourner ce problème, on a choisi d'utiliser la fonction `Arg.parse_argv` au lieu de `Arg.parse` et on modifie directement la valeur de `Sys.argv`. Il y a quelques défauts de cette solution:

1. Le tableau `Sys.argv` peut contenir des chaînes de caractères vides après le parsing. L'utilisateur de la bibliothèque doit les négliger.
2. Il nous manque un message complèt indiquant tous ces spécifications qui peut s'afficher quelque parts quand il y en a besoin (par exemple avec la commande `--help`).

Anecdote: En OCaml 4.03.0 et 4.04.0, dans le module `Arg` avec la fonction `Arg.parse`, pour une option de ligne de commande qui prend un seul argument (c'est ainsi le cas pour `-port` dans notre programme), le message d'erreur peut s'afficher trois fois si aucun argument est donné à cette option.

Exemples d'applications

Exemples basiques

- `put_get_test.ml` : un processus met l'entier 2 dans un canal et l'autre le lit et l'affiche. Faire `make put_get` pour générer ce programme
- `int_printer.ml` : Un processus génère une suite croissante infini d'entier, tandis que le second les récupère et les affiche dans la sortie standard.
- `alter_print.ml` : Deux processus écrivent et lisent alternativement une suite d'entier dans deux canaux.

Crible d'Ératosthène

Il s'agit d'un procédé permettant de trouver tous les nombres premiers inférieurs à un certain entier naturel donné. Le fichier `sieve_Erathostene.ml` contient l'algorithme décrit à la page 9 de l'article *Coroutines and Networks of Parallel Processes* de Gilles Kahn et David MacQueen. Le nombre de processus utilisés dans cet algorithme n'est pas borné et donc ne fonctionne pas bien avec l'implémentation distribuée par le réseau.

Tracé de l'ensemble de Mandelbrot

L'ensemble de Mandelbrot est une fractale définie comme l'ensemble des points c du plan complexe pour lesquels la suite des nombres complexes définie par récurrence par

z0 = 0 zn+1 = znˆ2 +c

est bornée.

Le fichier `Mandelbrot.ml` affiche l'ensemble sur [-2;2]x[-1.5;1.5]. L'image est divisée en plusieurs zones et le calcul de chaque zone est pris en charge par un processus. On peut spécifier la taille de l'image, le nombre de zones et le nombre d'itérations pour chaque point.

Pong

L'implémentation de pong n'utilise que la version réseau des KPN, un ordinateur lance le programme avec l'option `-wait` tandis que l'autre peut choisir les paramètres de jeu.

K-moyennes

Il s'agit d'un algorithme de partitionnement des données d'un problème d'optimisation combinatoire. Le fichier d'entrée contient sur chaque ligne un point dont les coordonnées sont séparées par des espaces. Le nombre de partitions `k`, le nombre d'itérations `i`, le nombre d'ouvriers `p` (de processus en parallèle), et le nombre de fois à exécuter l'algorithme `t` peuvent être données en arguments. Les centres des partitions sont ensuite calculées et écrits dans un fichier de sortie spécifié par l'option `-o`. En plus, si les entrées sont des points de dimension 2, on peut afficher le résultat en utilisant l'option `-plot`.

Divers: MapReduce

On a voulu implémenter le modèle de MapReduce dans le cadre de réseau de Kahn, mais c'est enfin abandonné dû à deux raisons principales:

1. **MapReduce est indéterministe, alors que KPN est déterministe**
   Le modèle MR est indéterministe dans le sens qu'il n'y a pas un ordre prédéfini des exécutions de tâches. Les résultats des ouvriers sont traités *immédiatement* par le patron dès qu'ils sont produits, ce qui est impossible dans un réseau de Kahn car on n'a pas de droit de tester si un canal est vide ou pas: une fois qu'on est bloqué, on est bloqué. Par conséquent, en modèlisant MapReduce par KPN, tout devient déterministe, ce qui montre une différence intrinsèque entre ces deux modèles.

2. **Le rôle de patron**
   Observons que dans le module `Functory`, le patron existe particulièrement pour effectuer un effet de bord, ce qui est gênant car quand on fait un `doco` dans les réseaux de Kahn, on aimerait bien que tous les processus soient purs (à éventuellement les opérations I/O près). Le fait que le patron peut effectuer un effet de bord impose qu'il vit dans le même ordinateur et le même processus Unix que le process qui a effectué la `doco`, ce que l'on n'a à priori pas de droit de controler au niveau d'un modèle abstrait tel que celui de réseau de Kahn.

Une petite mise en garde

Les deux points ci-dessus ne nous empêchent pas d'implémenter une interface MapReduce en utilisant notre bibliothèque de réseau de Kahn. Pourtant, ils nous font remarquer des nuances éventuelles entre le vrai MapReduce et un MapReduce qui est simulé par KPN.