Functory : Une bibliothèque de calcul distribué pour Objective Caml

Jean-Christophe Filliâtre[1,2,3] & K. Kalyanasundaram[3]

1: CNRS / LRI UMR 8623 F-91405 Orsay
2: Université Paris Sud F-91405 Orsay
3: INRIA Saclay – Île-de-France F-91893 Orsay
filliatr@lri.fr, kalyan.krishnamani@inria.fr

Résumé

Cet article présente Functory, une bibliothèque de calcul distribué pour Objective Caml. Les principales caractéristiques de cette bibliothèque sont (1) une interface polymorphe, (2) plusieurs réalisations correspondant à des contextes d'utilisation différents et (3) un mécanisme de tolérance aux pannes. Cet article détaille la conception et la réalisation de Functory et montre son potentiel sur de nombreux exemples.

1. Introduction

Cet article présente Functory, une bibliothèque de calcul distribué pour Objective Caml. Initialement, ce travail a été motivé par des besoins de calcul au sein de notre équipe de recherche, ProVal. Nos applications en vérification déductive de programmes incluent notamment la validation de très nombreuses formules logiques par des démonstrateurs automatiques variés [7]. Nos moyens de calcul consistent en quelques machines très puissantes (typiquement 8 ou 16 cœurs) et plusieurs ordinateurs de bureau (typiquement 2 cœurs). Aucune bibliothèque ne nous permettait de tirer facilement partie de cette infrastructure de calcul dans notre langage préféré. C'est pourquoi nous avons conçu et réalisé la bibliothèque Functory qui est le sujet de cet article. Cette bibliothèque est réalisée pour OCaml mais pourrait facilement être adaptée pour tout autre langage fonctionnel.

Functory n'est pas une bibliothèque qui aide l'utilisateur à paralléliser ses calculs. Son rôle consiste plutôt à offrir des facilités pour distribuer, de manière sûre, des calculs déjà identifiés comme indépendants. En particulier, Functory offre plusieurs interfaces génériques pour distribuer des calculs sur différents cœurs d'une même machine ou sur un réseau de machines. Ceci correspond exactement au contexte qui a motivé la construction de cette bibliothèque mais aussi, très probablement, à celui de nombreuses autres applications. Les principales caractéristiques de Functory sont les suivantes :

- *généricité* : les traits fonctionnels que sont l'ordre supérieur et le polymorphisme sont exploités pour fournir un maximum de généricité ;
- *simplicité* : on passe d'un calcul séquentiel à un calcul multi-cœurs puis à un calcul en réseau en ne modifiant que quelques lignes dans le code ;
- *distribution et tolérance aux pannes* : l'intégralité de la gestion de la distribution et de la tolérance aux pannes est prise en charge par la bibliothèque.

Bien que Functory ait été écrite avec un soucis de généralité, elle ne vise pas les fermes de calcul où de grandes quantités de données sont manipulées, en particulier parce qu'il n'y a pas, pour l'instant, de

Ce travail a été réalisé en partie dans le cadre du projet U3CAT (*Unification of Critical C Code Analysis Techniques*, ANR-08-SEGI-021).

moyen d'assurer la localité des données. Elle s'adresse plutôt aux équipes de recherche qui souhaitent exploiter rapidement des capacités de calcul existantes allant d'un simple ordinateur de bureau à un réseau de machines. Le reste de cette introduction décrit notre approche du calcul distribué dans le contexte d'un langage fonctionnel.

Calcul distribué. Initialement inspirée par la bibliothèque MapReduce[1] de Google [6], notre approche lui emprunte beaucoup de vocabulaire. La bibliothèque Functory est centrée autour de la notion de *tâche*. Les tâches représentent les calculs atomiques pouvant être réalisés de manière indépendante. Elles sont traitées par un *patron* et des *ouvriers* (respectivement *master* et *workers* en anglais). Les ouvriers représentent les capacités de calcul qui effectuent les tâches. Ils sont matérialisés par un ou plusieurs programmes, s'exécutant en parallèle sur une ou plusieurs machines. Le rôle du patron consiste à distribuer les tâches auprès des ouvriers et à récolter les résultats. Il est matérialisé par un unique programme s'exécutant de manière séquentielle.

Une partie importante du travail de Functory consiste en la transmission des tâches et de leurs résultats. Ceci implique leur sérialisation (en anglais *marshalling*) à travers le réseau, sur un parc potentiellement hétérogène en termes d'architectures et de systèmes d'exploitation. La taille du mot et l'*endianness* peuvent notamment varier[2]. Un autre aspect essentiel du calcul distribué, et de Functory en particulier, est la *tolérance aux pannes*. Les ouvriers peuvent ainsi être arrêtés, relancés, temporairement stoppés ou inatteignables à cause de problèmes liés au réseau, sans jamais compromettre le résultat final du calcul.

Une approche fonctionnelle. Nous avons essayé de tirer partie des spécificités de la programmation fonctionnelle pour proposer une interface la plus générique possible. L'une des idées principales de Functory est que chaque ouvrier est une fonction potentiellement polymorphe

$$\mathsf{worker}\colon\ \alpha \to \beta$$

où α dénote le type des tâches et β le type des résultats. Le patron est pour sa part une fonction à laquelle on passe d'une part une fonction pour traiter les résultats et d'autre part la liste des tâches initiales :

$$\mathsf{master}\colon\ (\alpha \to \beta \to \alpha\ \mathsf{list}) \to \alpha\ \mathsf{list} \to \mathsf{unit}$$

Cette fonction passée en argument à master est appliquée dès qu'un nouveau résultat est disponible. Elle peut produire de nouvelles tâches (d'où son type de retour α list), qui s'ajoutent alors à la liste des tâches à effectuer. Le processus complet s'achève lorsque toutes les tâches ont été effectuées.

Notre bibliothèque tire partie des capacités de sérialisation d'OCaml autant que possible. Ainsi lorsque le patron et les ouvriers sont matérialisés par le même exécutable, fonctions et valeurs polymorphes peuvent être sérialisées, ce qui permet de réaliser facilement le schéma ci-dessus. Il n'est cependant pas toujours possible d'utiliser le même programme pour le patron et les ouvriers. Dans ce cas, on peut tout de même continuer à sérialiser des valeurs polymorphes lorsque la version d'OCaml utilisée est la même pour tous. Sinon, on ne peut plus que transmettre des chaînes de caractères entre les différents acteurs. La bibliothèque Functory s'adapte à toutes ces situations en proposant plusieurs interfaces.

Cet article s'organise ainsi. La section 2 présente l'interface de la bibliothèque. Son utilisation est alors illustrée sur des exemples dans la section 3. La section 4 donne des détails techniques concernant la réalisation de Functory. Enfin la section 5 montre le potentiel de cette bibliothèque sur des tests

[1] Ironiquement, l'approche de Google a été elle-même inspirée par la programmation fonctionnelle.
[2] Les auteurs se refusent à utiliser le mot « boutisme » pour désigner l'*endianness*.

expérimentaux. Functory est librement distribuée à l'adresse `http://functory.lri.fr/`. Un rapport plus détaillé que le présent article, en anglais, est également disponible sur ce site.

2. Interface

Cette section décrit l'interface de la bibliothèque Functory. La fonctionnalité primitive est une fonction compute réalisant l'idée principale évoquée dans l'introduction.

```
val compute :
  worker:(α → β) →
  master:(α × γ → β → (α × γ) list) →
  (α × γ) list → unit
```

Les tâches sont des paires, de type $\alpha \times \gamma$, où la première composante sera transmise à l'ouvrier et la seconde conservée par le patron. La fonction worker doit être pure[3] et sera exécutée en parallèle par tous les ouvriers. La fonction master, au contraire, peut être impure et sera exécutée uniquement au sein d'un unique processus séquentiel. Cette fonction master accumule typiquement les résultats renvoyés par les ouvriers dans une structure locale. Elle peut en outre produire de nouvelles tâches, sous la forme d'une liste de type $(\alpha \times \gamma)$ list, qui sont alors ajoutées aux tâches restant à effectuer. Le troisième argument de compute est la liste des tâches initiales, qui déclenchent le calcul. La fonction compute rend la main lorsque toutes les tâches ont été effectuées. Il n'y a pas de résultat renvoyé, mais uniquement des effets de bord de la fonction master.

À titre d'illustration, voici comme écrire un équivalent distribué de la fonction Array.map, qui applique une fonction f à tous les éléments d'un tableau a.

```
let array_map f a =
  let n = Array.length a in
  let b = if n = 0 then [——] else Array.create n (f a.(0)) in
  let tasks = ref [] in for i = 1 to n - 1 do tasks := (a.(i), i) :: !tasks done;
  compute ~worker:f ~master:(fun (_,i) bi → b.(i) ← bi; []) !tasks;
  b
```

La liste des tâches, tasks, contient les couples (a.(i), i) pour tous les indices i du tableau. La fonction worker se réduit à la fonction f. La fonction master reçoit un résultat bi correspondant à un indice i et le stocke dans le tableau résultat b. La fin de cette section décrit plusieurs telles fonctions d'ordre supérieur fournies par Functory, toutes dérivées de la fonction compute.

En réalité, la bibliothèque Functory fournit *cinq* mises en œuvre différentes de la fonction compute, correspondant à cinq contextes d'utilisation différents. Les deux premiers sont les plus simples.

1. **Exécution purement séquentielle :** Elle permet d'obtenir un code de référence, pour mesurer des performances ou mettre au point son programme.
2. **Plusieurs cœurs sur une même machine :** Il s'agit là de distribuer le calcul sur une unique machine, uniquement en créant des processus fils.

Les trois contextes suivants correspondent à une distribution du calcul sur un réseau de machines.

3. **Patron et ouvriers sont matérialisés par un même exécutable :** Cette mise en œuvre exploite la capacité d'OCaml à sérialiser fonctions et valeurs polymorphes de manière portable. Selon que le programme est exécuté comme le patron ou comme un ouvrier, les arguments pertinents de la fonction compute sont utilisés.

[3] On entend ici *observationnellement pure* mais on autorise la levée d'exceptions pour signaler l'échec du calcul.

4. **Patron et ouvriers sont matérialisés par différents programmes, compilés avec la même version d'OCaml :** Il n'est plus possible de sérialiser des fonctions mais on peut encore sérialiser des valeurs polymorphes. En conséquence, la fonction compute est présentée sous la forme de deux fonctions, servant respectivement à réaliser le patron et les ouvriers :

   val Worker.compute : $(\alpha \rightarrow \beta) \rightarrow$ unit
   val Master.compute : $(\alpha \times \gamma \rightarrow \beta \rightarrow (\alpha \times \gamma)$ list$) \rightarrow (\alpha \times \gamma)$ list $\rightarrow$ unit

5. **Patron et ouvriers sont matérialisés par différents programmes, qui ne sont même pas compilés avec la même version d'OCaml :** Il n'est plus possible d'utiliser la sérialisation d'OCaml et la fonction compute est présentée sous la forme de deux fonctions ne manipulant plus que des chaînes de caractères :

   val Worker.compute : (string $\rightarrow$ string) $\rightarrow$ unit
   val Master.compute : (string $\times \gamma \rightarrow$ string $\rightarrow$ (string $\times \gamma$) list) $\rightarrow$
   (string $\times \gamma$) list $\rightarrow$ unit

La bibliothèque Functory est donc organisée en trois modules : Sequential pour le calcul purement séquentiel ; Cores pour le calcul distribué sur plusieurs cœurs d'une même machine ; et enfin Network pour le calcul en réseau. Ce dernier module comporte trois sous-modules, appelés respectivement Same, Poly and Mono, correspondant aux situations 3, 4 et 5 ci-dessus.

Fonctions dérivées. Dans de nombreuses situations, la mise en œuvre la plus simple du parallélisme consiste à appliquer une opération sur une liste, le traitement de chaque élément pouvant être réalisé en parallèle. Functory fournit plusieurs fonctions d'ordre supérieur offrant du calcul parallèle sur des listes, toutes dérivées de la fonction compute. En particulier, elles sont disponibles dans les cinq modules décrits ci-dessus.

L'opération la plus naturelle est celle consistant à appliquer une fonction à tous les éléments d'une liste, analogue à la fonction array_map décrite plus haut, c'est-à-dire

val map : $(\alpha \rightarrow \beta) \rightarrow \alpha$ list $\rightarrow \beta$ list

Plus subtilement, on peut combiner une fonction f:$\alpha \rightarrow \beta$ avec une fonction fold:$\gamma \rightarrow \beta \rightarrow \gamma$ pour calculer, à partir d'une liste l et d'un accumulateur initial a, la valeur finale

$$\text{fold ... (fold (fold } a \text{ (f } x_1\text{)) (f } x_2\text{)) ... (f } x_n\text{)} \tag{1}$$

pour une certaine permutation non spécifiée $[x_1, x_2, ..., x_n]$ de la liste l. On peut alors distinguer deux cas, selon que l'opération fold est beaucoup moins coûteuse que l'opération f, et peut être effectuée localement par le patron, ou qu'au contraire elle peut être coûteuse et a donc intérêt à être effectuée en parallèle des opérations f. La bibliothèque fournit donc deux fonctions, correspondant à ces deux cas de figure :

val map_{local,remote}_fold : f:$(\alpha \rightarrow \beta) \rightarrow$ fold:$(\gamma \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma \rightarrow \alpha$ list $\rightarrow \gamma$

Dans le cas de map_remote_fold, une seule opération fold peut être effectuée à la fois (possiblement en parallèle d'opérations f), comme le montre l'équation (1). Il existe cependant des situations dans lesquelles plusieurs opérations fold peuvent être effectuées en parallèle, dès que des résultats intermédiaires de f sont disponibles. C'est le cas notamment lorsque l'opération fold est associative (ce qui implique que les types β et γ sont égaux). Lorsque l'opération fold est de plus commutative, on peut effectuer encore plus d'opérations fold en parallèle. Notre interface fournit donc deux autres fonctions pour ces cas particuliers :

val map_fold_{a,ac} : f:($\alpha \to \beta$) $\to$ fold:($\beta \to \beta \to \beta$) $\to \beta \to \alpha$ list $\to \beta$

Ces cinq fonctions se dérivent facilement de la fonction compute. Voici par exemple comment réaliser la fonction map_fold_a. On commence par introduire un type somme pour distinguer les tâches f des tâches fold, c'est-à-dire **type** (α, β) map_or_fold = Map **of** α — Fold **of** β. Le code de l'ouvrier se résumé à distinguer les deux types de tâches.

```
let worker = function Map x → f x — Fold (x, y) → fold x y
```

Le code du patron est plus compliqué. En effet, l'opération fold étant seulement associative, on ne peut calculer d'expression analogue à (1) que pour des éléments *consécutifs* de la liste initiale l. Le patron conserve donc, dans une table de hachage locale, tous les intervalles $[x_i, \ldots, x_j]$ de l pour lesquels la valeur

$$\mathsf{f}\ x_i \oplus \cdots \oplus \mathsf{f}\ x_j$$

a déjà été calculée (en notant ici fold par $\oplus$), ainsi que cette valeur. Cete table est doublement indexée, par i et par j. Initialement cette table est vide et la liste des tâches ne contient que des tâches de type Map, une par élément de l. De manière générale, chaque tâche est associée au couple (i, j) dénotant le segment qu'elle représente. Lorsque le patron reçoit un résultat r pour un segment $[x_i, \ldots, x_j]$, trois cas se présentent. Soit il existe dans la table un résultat pour un segment de la forme $[x_k, \ldots, x_{i-1}]$, que l'on extrait grâce à la clé $i-1$, et on produit alors une nouvelle tâche Fold pour calculer la valeur correspondant à $[x_k, \ldots, x_j]$, après avoir supprimé de la table les entrées pour k et $i-1$. Soit il existe dans la table un résultat pour un segment de la forme $[x_{j+1}, \ldots, x_k]$, que l'on traite de la même manière. (Si les deux situations se présentent en même temps, on choisit arbitrairement, car il y a de toutes façons deux tâches Fold à effectuer.) Soit enfin il n'y a ni entrée pour $i-1$ ni entrée pour $j+1$ dans la table, auquel cas on se contente d'ajouter la valeur r pour les deux clés i et j. Lorsque toutes les tâches ont été effectuées, la table doit contenir exactement une valeur, associée aux clés 1 et n, sauf dans le cas $n = 0$ où aucun calcul n'est nécessaire. Au final, la fonction map_fold_a prend la forme suivante, où on a seulement omis le code de fusion des résultats expliqué ci-dessus.

```
let map_fold_a ~f ~fold acc l =
  let tasks = let i = ref 0 in List.map (fun x → incr i; Map x, (!i, !i)) l in
  let results = Hashtbl.create 17 in
  let merge i j r = ...traiter le résultat r du segment [i,j]... in
  let worker = function Map x → f x — Fold (x, y) → fold x y in
  let master x r = match x with Map _,(i,_) → merge i i r — Fold _,(i,j) → merge i j r in
  compute ~worker ~master tasks;
  try let _,_,r = Hashtbl.find results 1 in r with Not_found → acc
```

On peut évidemment dériver d'autres fonctions du même genre, comme par exemple une variante où seule la fonction fold est significative. L'utilisateur de Functory ne devrait pas avoir de mal à les dériver lui-même de la fonction compute ou des fonctions du type map_fold ci-dessus.

3. Études de cas

Cette section présente plusieurs études de cas que nous avons réalisées avec Functory. On se concentre ici sur l'utilisation de la bibliothèque ; les résultats expérimentaux sont présentés plus loin (section 5). Le code source de tous les exemples ci-dessus est contenu dans la distribution de Functory, dans le sous-répertoire `tests`.

3.1. Multiplication de matrices

Comme premier exemple, on considère la multiplication de deux matrices a et b, de tailles respectives n × p et p × m. Le résultat sera stocké dans une matrice c de taille n × m. On suppose que a, b et c sont contenues dans des variables globales. En supposant en outre a organisée en lignes et b en colonnes, une multiplication séquentielle s'écrirait ainsi

```
for i = 0 to n-1 do
  for j = 0 to m-1 do
    for k = 0 to p-1 do
      c.(i).(j) ← c.(i).(j) + a.(i).(k) × b.(j).(k) (* b en colonnes *)
    done
  done
done
```

l'addition et la multiplication des coefficients étant notées respectivement + et ×. La complexité de ce code est clairement $O(n \times m \times p)$,

Une manière évidente de distribuer ce calcul consiste à faire de la boucle interne sur k une tâche élémentaire. On construit alors une liste de n × m tâches, de la manière suivante :

```
let tasks =
  let l = ref [] in
  for i = 0 to n-1 do for j = 0 to m-1 do
    tasks := ((a.(i), b.(j)), (i,j)) :: !tasks
  done done;
  !l
```

Chaque tâche est une paire formée d'une part d'une ligne a.(i) et d'une colonne b.(j), et d'autre part d'une position (i,j). L'ouvrier reçoit la première composante et calcule le produit scalaire.

```
let worker (ai, bj) =
  let c = ref 0 in
  for k = 0 to p-1 do c := !c + ai.(k) × bj.(k) done;
  !c
```

La patron est une fonction d'une ligne, qui reçoit le résultat r renvoyé par l'ouvrier et remplit la matrice c, en fonction de la position contenue dans la seconde composante de la tâche. Aucune nouvelle tâche n'est produite.

```
let master (_, (i,j)) r = c.(i).(j) ← r; []
```

Au final, l'ensemble du calcul est lancé par un appel à la fonction compute.

```
let () = compute ~worker ~master !tasks
```

où les variables α, β et γ du type de compute sont respectivement instanciées par coeff array × coeff array, coeff et int × int.

Utiliser la version séquentielle de Functory est aussi simple que d'ajouter **open** Sequential au début du code. Le calcul est alors similaire à la multiplication usuelle donnée plus haut. Cela peut être néanmoins utile pour vérifier la correction du code avant de chercher à distribuer le calcul.

Supposons maintenant que l'on veuille utiliser une machine 4 cœurs pour effectuer le calcul. Il suffit alors de remplacer la ligne **open** Sequential par les deux lignes suivantes.

```
open Cores
let () = set_number_of_cores 4
```

Le reste du code est inchangé.

Supposons enfin que l'on souhaite utiliser plutôt des machines présentes sur le réseau, par exemple deux machines appelées `orcus` et `belzebuth` offrant respectivement 4 et 8 cœurs. Il suffit de remplacer les deux lignes de code ci-dessus par les suivantes.

```
open Network
let () = declare_workers ~n:4 "orcus"
let () = declare_workers ~n:8 "belzebuth"
open Same
```

On utilise ici le sous-module Same de Network, qui permet d'utiliser le même exécutable pour le patron et les ouvriers. Ce module fournit toujours une fonction compute de même signature que dans les modules Sequential et Cores, et le reste du code est toujours inchangé. Le patron et les ouvriers sont distingués à l'exécution par la présence de la variable d'environnement `WORKER`.

Si on a besoin d'écrire deux programmes différents pour le patron et les ouvriers, pour des raisons d'incompatibilité de binaires ou tout autre raison, l'interface de Functory permet de le faire. Si les deux programmes sont compilés avec la même version d'OCaml, on utilise le module Poly. Commençons par l'ouvrier. Son code prend la forme suivante.

```
open Network.Poly
let worker (ai, bj) = ...
let () = Worker.compute worker ()
```

La fonction Worker.compute entre dans une boucle qui attend les tâches envoyées par le patron et renvoie les résultats calculés par la fonction worker. Le code du patron, quant à lui, est quasiment le même qu'auparavant. On commence par remplacer Same par Poly.

```
open Network
let () = declare_workers ~n:4 "orcus"
let () = declare_workers ~n:8 "belzebuth"
open Poly
```

La construction des tâches et la fonction master sont inchangées.

```
let tasks = ...
let master (_, (i,j)) r = ...
```

Enfin on lance le calcul avec la fonction Master.compute, qui ne prend plus de fonction worker en argument.

```
let () = Master.compute ~master tasks
```

Quand le patron et les ouvriers sont compilés avec des versions différentes d'OCaml, la bibliothèque fournit une interface ne permettant plus que l'échange de chaînes de caractères. En conséquence, il faut convertir les tâches et les résultats depuis et vers des chaînes, dans les deux programmes. L'ouvrier modifié prend alors la forme suivante.

```
open Mono
let worker (ai, bj) = ...
```

```
let worker_string s = string_of_coeff (worker (task_of_string s))
let () = Worker.compute worker_string ()
```

Le patron est modifié de la même façon. On remplace Poly par Mono et on encode/décode les tâches et résultats.

```
let tasks = ... string_of_task ...
let master (_, (i,j)) r = c.(i).(j) ← coeff_of_string r; []
```

Les quatre fonctions de conversion string_of_{task,coeff} et {task,coeff}_of_string sont à la charge de l'utilisateur.

3.2. Autres études de cas

Nous présentons ici trois autres études de cas, plus rapidement.

N-reines. Il s'agit du problème classique consistant à calculer le nombre de façons de disposer N reines sur un échiquier $N \times N$ sans que deux d'entre elles soient en prise. On utilise un algorithme standard, consistant à positionner une reine sur chaque ligne de l'échiquier, en partant de la première ligne. Il est facile de distribuer le calcul : on considère toutes les façons de placer les reines des D premières lignes et on effectue le reste du calcul en parallèle. Pour $D = 1$ on obtient ainsi N tâches ; pour $D = 2$ on obtient $N^2 - 3N + 2$ tâches ; et ainsi de suite. Chaque tâche est composée de quelques entiers et son résultat est un entier donnant le nombre de solutions pour cette tâche. On utilise la fonction map_local_fold où f effectue la recherche et fold somme les résultats intermédiaires.

L'ensemble de Mandelbrot. Dessiner l'ensemble de Mandelbrot est un autre exemple de calcul aisément distribuable. Il s'agit de l'ensemble des points c du plan complexe telle que la suite définie par $z_0 = 0$ et $z_{n+1} = z_n^2 + c$ reste de module borné. Pour le dessiner, on cherche le plus petit n tel que $|z_n| > 2$, ce qui assure alors la divergence, en s'arrêtant après un nombre maximal d'itérations fixé (par exemple 200). Il est clair que la couleur de chaque point peut être calculée indépendamment. Supposons donnée une région du plan complexe à dessiner, ainsi que la taille $\mathsf{w} \times \mathsf{h}$ en pixels de l'image finale. On se donne un nombre de tâches $\mathsf{t} \geq 1$ comme paramètre. Il est immédiat de découper l'image en t sous-images, par exemple en bandes horizontales (si $\mathsf{h} \geq \mathsf{t}$) ou plus généralement en blocs rectangulaires. Chaque tâche est constituée de quatre flottants définissant la région à dessiner, ainsi que deux entiers donnant la taille de l'image correspondante. Le résultat est une matrice de pixels, de taille $(\mathsf{w} \times \mathsf{h})/\mathsf{t}$. Ainsi, dessiner une image 800×600 en utilisant 20 tâches donnera 20 sous-images de 176 000 octets chacune, en supposant chaque pixel représenté par quatre octets.

Démonstrateurs automatiques. Le dernier exemple correspond à celui mentionné dans l'introduction. Il s'agit ici de vérifier la validité de 80 obligations de preuve (OP), issues de la plate-forme Why [7], à l'aide de quatre démonstrateurs automatique de la famille SMT (à savoir Alt-Ergo, Simplify, Z3 et CVC3). Chaque OP est vérifiée avec chaque démonstrateur, ce qui fait un total de 320 tâches. Les OP sont contenues dans des fichiers accessibles par NFS et une tâche est donc un nom de fichier et un nom de démonstrateur. Le démonstrateur est appelé comme un programme externe, avec un temps d'exécution maximal. Le résultat d'une tâche est la réponse du démonstrateur (valide, abandon, temps limite atteint, erreur pendant l'exécution) et son temps de calcul. Le patron collecte les résultats et les présente au final sous forme d'une table synthétique.

4. Détails techniques

Cette section décrit la réalisation des différents modules de la bibliothèque Functory introduits section 2, exception faire du module Sequential, dont la réalisation est immédiate. Les deux modules Cores et Network contiennent une boucle principale analogue, de la forme suivante :

tant que tâches à faire ∨ tâches en cours
  tant que tâches à faire ∧ ouvriers disponibles
    affecter une tâche à un ouvrier
  attendre la fin d'une tâche
    ajouter les nouvelles tâches renvoyées par master

La différence se situe dans la technologie utilisée pour affecter une tâche à un ouvrier, pour attendre la fin d'une tâche et enfin pour ajouter de la tolérance aux pannes.

4.1. Cores

Le module Cores permet de distribuer le calcul sur une unique machine, typiquement en exploitant plusieurs cœurs. Comme illustré section 3, la fonction set_number_of_cores permet de spécifier le nombre de cœurs à utiliser. Ce nombre peut être différent du nombre effectif de cœurs de la machine. Il peut même être strictement plus grand. En fait, ce nombre indique tout simplement combien de tâches peuvent être effectuées simultanément.

Le module Cores est réalisé à l'aide de processus Unix, en utilisant la fonction Unix.fork de la bibliothèque standard d'OCaml. Plus précisément, un compteur indique le nombre d'ouvriers disponibles et l'affectation d'une tâche à un ouvrier se fait en créant un nouveau sous-processus avec fork. À la fin du calcul, le résultat de la tâche est transmis au processus père par sérialisation dans un tube. La répartition des tâches sur les différents cœurs (physiques) de la machine, le cas échéant, est laissée au système. Il se peut donc que deux tâches se retrouvent être exécutées sur le même cœur, y compris dans le cas où le nombre de cœurs déclarés est inférieur ou égal au nombre de cœurs effectifs.

4.2. Network

Le module Network permet de distribuer le calcul sur un réseau de machines. Comme illustré section 3, la fonction declare_workers: n:int → string → unit permet de déclarer l'ensemble des machines du réseau sur lesquels se trouvent des ouvriers, en spécifiant un nombre d'ouvriers par machine (qui ne coïncide pas nécessairement avec un nombre effectif de cœurs). Le module Network est basé sur une architecture client/serveur utilisant TCP/IP, où chaque ouvrier est un serveur et où le patron est (ironiquement) le client de chaque ouvrier. Le patron est un programme purement séquentiel. En particulier, il ne peut administrer les tâches pendant l'exécution de la fonction master. Ce n'est pas un problème en pratique, car on peut supposer que la fonction master s'exécute rapidement. L'ouvrier, en revanche, crée un nouveau processus avec fork pour exécuter la tâche qui lui est confiée et peut donc continuer à communiquer avec le patron pendant son calcul.

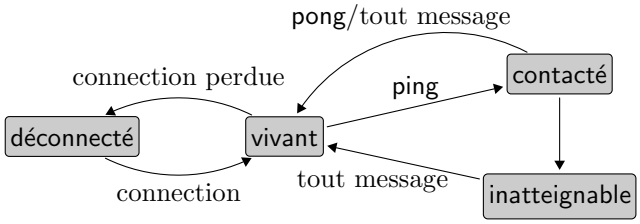
Protocole. Le protocole utilisé comporte sept messages différents. Les quatre messages possibles envoyés par le patron à un ouvrier sont : Assign(id:int, f:string, x:string) pour affecter une nouvelle tâche id à l'ouvrier, sous la forme de deux chaînes f et x dont le sens dépend du contexte ; Kill(id:int) pour demander l'interruption de la tâche id ; Stop pour demander l'arrêt définitif de l'ouvrier ; et enfin Ping pour vérifier que l'ouvrier est toujours réactif. Inversement, les trois messages possibles envoyés par un ouvrier au patron sont : Pong en réponse à un message Ping ; Completed(id:int, s:string) pour renvoyer le résultat s de la tâche id ; et enfin Aborted(id:int) pour signifier l'échec de la tâche id, soit en réponse à Kill, soit à cause d'une erreur lors de l'exécution de la fonction worker.

Ce protocole est de telle sorte que patron et ouvriers peuvent être exécutés sur des plate-formes complètement différentes, vis-à-vis de l'*endianness*, de la version d'OCaml et du système d'exploitation. Dans le sous-module Same, les arguments f et x du message Assign désignent respectivement la sérialisation d'une fonction et de son argument, et l'argument s du message Completed la sérialisation du résultat. Dans les sous-module Poly et Mono, en revanche, la partie f du message Assign n'est plus pertinente car la fonction worker est maintenant locale à l'ouvrier. Les chaînes x et s restent pertinentes ; pour Poly, ce sont des sérialisations de valeurs OCaml et pour Mono de simples chaînes de caractères.

4.2.1. Tolérance aux pannes

L'une des principales difficultés dans un environnement de calcul distribué est la tolérance aux pannes. C'est l'un des principaux atouts de la bibliothèque Functory. La tolérance aux pannes de Functory est limitée aux ouvriers ; la tolérance aux pannes du patron est laissée à l'utilisateur, par exemple sous la forme d'une sauvegarde régulière de l'état du patron. Les pannes des ouvriers sont de deux sortes : un ouvrier peut être stoppé et éventuellement plus tard redémarré ; ou un ouvrier peut être temporairement ou définitivement inaccessible sur le réseau. Dans tous les cas, on souhaite que le calcul parvienne à son terme, dès lors que cela reste possible, et idéalement en utilisant au mieux les ressources disponibles.

Pour assurer cette tolérance aux pannes, le patron maintient en permanence l'état de chaque ouvrier. Cet état est contrôlé par deux délais T_1 et T_2, paramétrés par l'utilisateur, et par l'ensemble des messages échangés. Il y a quatre états possibles pour un ouvrier : déconnecté signifie qu'il n'y a pas de connection TCP en cours entre le patron et l'ouvrier ; vivant signifie qu'un message de l'ouvrier a été reçu par le patron il y a moins de T_1 secondes ; contacté signifie que l'ouvrier n'a pas envoyé de message depuis plus de T_1 secondes et que le patron lui a envoyé un message Ping depuis moins de T_2 secondes ; enfin inatteignable signifie que l'ouvrier n'a toujours pas répondu au message Ping (depuis plus de T_2 secondes). Dès qu'un message est reçu en provenance d'un ouvrier, son état est mis à jour. On a donc l'automate suivant pour les états d'un ouvrier donné.



La tolérance aux pannes est réalisée en exploitant l'état des ouvriers de la manière suivante. Les tâches ne sont envoyées qu'à des ouvriers se trouvant dans l'état vivant ou contacté. D'autre part, dès qu'un ouvrier en train d'effectuer une tâche t passe dans l'état déconnecté ou inatteignable, la tâche t est reprogrammée, ce qui signifie qu'elle est ajoutée de nouveau à l'ensemble des tâches restant à effectuer. Lorsque le patron reçoit un résultat pour une tâche t, il déprogramme la tâche t si elle avait été reprogrammée et indique à tout ouvrier ayant déjà entrepris de recalculer t de cesser son travail (avec le message Kill).

Enfin, il est important de signaler que notre bibliothèque est également robuste vis-à-vis des exceptions levées par la fonction worker. Le cas échéant, un message Aborted est envoyé au patron et la tâche est reprogrammée. C'est la responsabilité de l'utilisateur de rattraper et de traiter ces exceptions si nécessaire.

5. Résultats expérimentaux

Cette section présente quelques résultats expérimentaux obtenus avec les quatre programmes décrits section 3.

N-reines. La table suivante montre les temps d'exécution pour différentes valeurs de N et pour chacun des trois modules Sequential, Cores et Network. L'objectif est ici de mesurer le facteur par rapport à l'exécution séquentielle. En conséquence, tous les calculs sont effectués sur la même machine, un Intel Xeon 8 cœurs 3.2 GHz sous Linux Debian. L'exécution séquentielle utilise un unique cœur. La version multi-cœurs utilise les 8 cœurs de la machine. La version réseau utilise également 8 ouvriers locaux à la machine, le patron s'exécutant sur une machine distante pour induire des communications réseaux réalistes. La première colonne donne la valeur de N et la seconde le nombre de tâches. Les trois colonnes suivantes donnent les temps d'exécution, en secondes, et le facteur d'accélération entre parenthèses.

N	D	#tâches	Sequential	Cores	Network
16	1	16	15.2	2.04 (7.45×)	2.35 (6.47×)
	2	210	15.2	2.01 (7.56×)	21.80 (0.69×)
17	1	17	107.0	17.20 (6.22×)	16.20 (6.60×)
	2	240	107.0	14.00 (7.64×)	24.90 (4.30×)
18	1	18	787.0	123.00 (6.40×)	125.00 (6.30×)
	2	272	787.0	103.00 (7.64×)	124.00 (6.34×)
19	1	19	6120.0	937.00 (6.53×)	940.00 (6.51×)
	2	306	6130.0	796.00 (7.70×)	819.00 (7.48×)

Il apparaît clairement que les modules Cores et Network permettent d'obtenir un facteur d'accélération significatif, qui atteint presque 8, c'est-à-dire le nombre de cœurs utilisés. Il apparaît également que le module Network donne de meilleurs résultats lorsque le temps de calcul domine largement le temps de communication. Les deux cas extrêmes correspondent aux deuxième et dernière lignes : dans la deuxième ligne, le temps de communication domine et représente en fait plus de 91% du temps total ; dans la dernière, au contraire, il ne représente plus que 4.6% du temps total. D'une manière générale, le module Network n'est intéressant que si le temps de calcul de chaque tâche est significatif.

L'ensemble de Mandelbrot. Dans cette expérience, on dessine le fragment rectangulaire de l'ensemble de Mandelbrot compris entre les points $(-1.1, 0.2)$ et $(-0.8, 0.4)$, sous la forme d'une image de résolution $9\,000 \times 6\,000$. On utilise la même machine que dans l'expérience précédente. Un calcul purement séquentiel prend 29,4 secondes. Les résultats pour les modules Cores et Network sont présentés figure 1. Les meilleurs résultats sont obtenus avec le module Cores, où la communication n'intervient que localement, entre processus. Il y a deux différences significatives avec l'expérience précédente des N-reines. D'un côté, le nombre de tâches peut être contrôlé beaucoup plus facilement. Expérimentalement, on constate que le nombre optimal de tâches est de l'ordre de 30. D'un autre côté, en revanche, le résultat de chaque calcul est une image, et non plus un simple entier. En conséquence, les coûts de communication sont bien plus grands. Dans cette expérience particulière, la taille totale des résultats dépasse les 200 Mo.

Multiplication de matrices Cette expérience a été inspirée par le concours de programmation associé à la conférence PASCO 2010 [4]. L'un des problèmes consistait à multiplier deux matrices de taille 100×100, c'est-à-dire $\mathsf{n} = \mathsf{p} = \mathsf{m} = 100$ avec les notations de la section 3.1. Les coefficients sont entiers mais peuvent avoir des milliers de chiffres et GMP [2] est utilisé ici pour les calculs relatifs aux coefficients.

#cœurs	#tâches	Cores	Network
2	10	15.8 (1.86×)	20.3 (1.45×)
	30	15.7 (1.87×)	18.7 (1.57×)
	100	16.1 (1.83×)	19.8 (1.48×)
	1000	19.6 (1.50×)	38.6 (0.76×)
4	10	9.50 (3.09×)	14.4 (2.04×)
	30	8.26 (3.56×)	11.4 (2.58×)
	100	8.37 (3.51×)	11.4 (2.58×)
	1000	10.6 (2.77×)	20.5 (1.43×)
8	10	9.40 (3.13×)	12.6 (2.33×)
	30	4.24 (6.93×)	7.6 (3.87×)
	100	4.38 (6.71×)	7.5 (3.92×)
	1000	6.86 (4.29×)	11.3 (2.60×)

FIG. 1 – Résultats pour l'ensemble de Mandelbrot.

On compare les performances de deux programmes distribués utilisant Functory. Dans le premier, appelé mm1, chaque tâche consiste en le calcul d'un unique coefficient de la matrice résultat, exactement comme décrit section 3.1. Dans le second, appelé mm2, chaque tâche consiste en le calcul de toute une ligne de la matrice résultat. En conséquence, le nombre total de tâches est $\mathsf{n} \times \mathsf{m} = 10,000$ pour mm1 et seulement $\mathsf{n} = 100$ pour mm2. Les résultats expérimentaux, toujours en secondes et toujours sur la même machine, sont les suivants.

	mm1 (10,000 tâches)	mm2 (100 tâches)
Sequential	20.3	20.2
Cores (2 cœurs)	22.7 (0.89×)	11.3 (1.79×)
(4 cœurs)	12.3 (1.65×)	6.1 (3.31×)
(6 cœurs)	8.6 (2.36×)	4.3 (4.70×)
(8 cœurs)	8.0 (2.54×)	3.5 (5.77×)

On exclut les résultats pour la configuration en réseau, car ils n'apportent aucune amélioration par rapport à l'exécution séquentielle. La raison en est que les coûts de communication dominent largement les coûts de calcul, donnant un temps d'exécution total supérieur à 30 secondes. Que ce soit pour mm1 ou mm2, le nombre total de coefficients transmis est en effet en $O(\mathsf{n} \times \mathsf{m} \times \mathsf{p})$, ce qui représente ici des milliards d'octets. Dans une réalisation moins naïve, les ouvriers pourrait lire les deux matrices une seule fois, par exemple depuis un fichier, et le patron n'enverrait uniquement que des indices de lignes et de colonnes. Cela réduirait le volume des communications à seulement $O(\mathsf{n} \times \mathsf{m})$.

Démonstrateur automatiques. Comme expliqué plus haut, cette expérience consiste à vérifier la validité de 80 formules logiques à l'aide de quatre démonstrateurs automatiques de la famille SMT. Chacune de ces 320 tâches est exécutée dans une limite de temps fixée à une minute. La plate-forme de test est ici constituée de trois machines en réseau, de 4, 8 et 8 cœurs respectivement. Le tableau ci-dessus indique le temps total passé dans chaque démonstrateur automatique, cumulé pour chacune des réponses possibles.

démonstrateur	valide	abandon	temps limite	erreur
Alt-ergo	406.0	3.0	11 400.0	0.0
Simplify	0.5	0.4	1 200.0	222.0
Z3	80.7	0.0	1 800.0	1 695.0
CVC3	303.0	82.7	4 200.0	659.0

Ce qui nous intéresse ici est en particulier le temps total d'exécution séquentielle cumulée, qui dépasse les 6 heures de calcul. Cependant, en utilisant Functory et notre petit réseau de trois machines, le temps total de calcul n'a pas dépassé 22 minutes et 30 secondes, ce qui représente une accélération d'un facteur 16. On est encore loin du facteur optimal 20 (on utilise ici 20 cœurs de puissance équivalente), en particulier parce que certains démonstrateurs allouent beaucoup de mémoire et que le temps passé dans les appels systèmes n'est pas pris en compte ici. Un facteur 16 est cependant une accélération significative pour une utilisation au jour le jour.

6. Conclusion et perspectives

Nous avons présenté Functory, une bibliothèque de calcul distribué pour OCaml. Les caractéristiques principales de cette bibliothèque sont son interface générique, sous forme de fonctions polymorphes d'ordre supérieur, et la possibilité de basculer aisément entre exécutions séquentielle, multi-cœurs et réseau. En particulier, Functory permet d'utiliser le même exécutable pour le patron et les ouvriers, ce qui rend le déploiement de petits programmes immédiat — les deux ne sont distingués que par une variable d'environnement. Bien entendu, Functory permet également l'utilisation de deux programmes complètement différents, ce qui est toujours le cas dans une application à grande échelle. Une autre caractéristique essentielle de Functory est son mécanisme robuste de tolérance aux pannes, qui simplifie grandement le code de l'utilisateur.

Travaux connexes. Une bibliothèque relativement proche de la nôtre est l'implémentation de MapReduce pour OCaml par Yohann Padioleau [12]. Elle est construite au dessus d'OCamlMPI [9], alors que notre bibliothèque utilise son propre protocole. OCamlMPI permet notamment un déploiement plus flexible des ouvriers. D'un autre côté, notre bibliothèque offre une interface plus générique et surtout la tolérance aux pannes. Mis à part OcamlMPI, il existe d'autres environnements sur lesquels une bibliothèque comme Functory pourrait être construite. Un exemple est Jo&Caml [11]. Il est cependant important de noter que Jo&Caml n'offre pas de tolérance aux pannes *de facto*. L'utilisateur doit l'inclure dans son code, comme cela a déjà été démontré dans certaines expériences impliquant Jo&Caml [10]. Vu que l'essentiel de la complexité de Functory se situe dans le code de tolérance aux pannes, l'apport de Jo&Caml serait relativement anecdotique.

Il existe également des bibliothèques de calcul distribué pour d'autres langages fonctionnels. L'une d'elles est le projet Disco [3], une implémentation de MapReduce dans le langage Erlang [1]. Notre bibliothèque, au contraire, n'est pas une implémentation de MapReduce. Une autre manière d'exploiter les architectures multi-cœurs consiste à faire du parallélisme de données, et cela reste pertinent dans le contexte de la programmation fonctionnelle [8]. Le parallélisme de données n'est pas du tout l'objectif de Functory.

Perspectives. La bibliothèque Functory pourrait être enrichie de nombreuses façons. L'une d'elles est la prise en compte, dans l'affectation des tâches aux ouvriers, de paramètres de ressources telles que la localité des données, la puissance de calcul, l'espace mémoire disponible, etc. On peut imaginer pour cela offrir à l'utilisateur des moyens de contrôle de l'affectation des tâches ou un processus plus automatique. Cela permettrait d'utiliser Functory dans les mêmes contextes que MapReduce. Il est important de noter qu'à l'heure actuelle, en l'absence de toute information concernant les tâches, leur ordonnancement est complètement arbitraire. Les modules Cores et Network utilisent de une simple

file pour les tâches restant à effectuer et les nouvelles tâches, produites par le patron, sont ajoutées à la fin de cette file.

Une autre amélioration consiste à faciliter le déploiement en réseau, en permettant l'ajout et le retrait dynamique de machines ouvrières. Pour l'instant notre bibliothèque utilise une liste de machines spécifiées *a priori* mais elle pourrait par exemple lire des noms de machines depuis un fichier, périodiquement.

Functory offre peu d'outils pour visualiser en temps réel l'état des calculs et des communications (mis à part quelques informations de *debugging*). Calculer et stocker une telle information est immédiat, et en partie déjà fait, mais son observation en temps réel demande plus de travail. Une possibilité serait de réutiliser une partie de l'outil Ocamlviz [5].

Enfin, une fonctionnalité appréciable du MapReduce de Google est l'accélération des fins de calcul en utilisant des ouvriers disponibles pour dupliquer les toutes dernières tâches. Vu que Functory comporte déjà tout le mécanisme de tolérance aux pannes, ajouter une telle fonctionnalité devrait être relativement simple.

Remerciements. Les auteurs remercient Alain Mebsout et Johannes Kanig pour avoir testé la bibliothèque Functory et pour leurs commentaires, ainsi que Claude Marché pour sa relecture attentive de cet article.

Références

[1] The Erlang Programming Language. `http://www.erlang.org/`.

[2] The GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`.

[3] The Disco Project, 2009. `http://discoproject.org/`.

[4] Parallel Symbolic Computation 2010 (PASCO), 2010. `http://pasco2010.imag.fr/`.

[5] Sylvain Conchon, Jean-Christophe Filliâtre, Fabrice Le Fessant, Julien Robert, and Guillaume Von Tokarski. Observation temps-réel de programmes Caml. In *Vingt-et-unièmes Journées Francophones des Langages Applicatifs*, Vieux-Port La Ciotat, France, January 2010. INRIA.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[7] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[8] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores : Nested Data Parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[9] Xavier Leroy. OCamlMPI : Interface with the MPI Message-passing Interface. `http://pauillac.inria.fr/~xleroy/software.html`.

[10] Louis Mandel and Luc Maranget. Programming in JoCaml (tool demonstration). In *17th European Symposium on Programming (ESOP 2008)*, pages 108–111, Budapest, Hungary, April 2008.

[11] Louis Mandel and Luc Maranget. The Jo&Caml Language, 2008. `http://jocaml.inria.fr/`.

[12] Yoann Padioleau. A poor's man MapReduce for OCaml, 2009. `http://www.padator.org/ocaml/mapreduce.pdf`.