Implémentation de réseaux de Kahn

Christophe Cordero, Li-yao Xia

26 mai 2013

1 Composition du projet

Nous présentons maintenant les implémentations dans l'ordre où elles ont été écrites, qui est aussi, selon nous, dans l'ordre de technicité croissante.

2 Implémentation séquentielle

On représente un processus par le type suivant :

```
type 'a process = unit -> 'a result
and 'a result =
  | Proc of 'a process
  | Res of 'a
```

Un processus est une fonction qui effectue un pas *atomique* de calcul et :

1. Si le calcul n'est pas terminé, il renvoie la suite du calcul sous forme d'un autre processus `Proc`,
2. Sinon, il renvoie le résultat `Res`.

L'exécution `run` d'un tel processus est simple : on fait avancer le calcul jusqu'à obtenir un résultat.

Cette représentation permet alors de simuler un processus `p` obtenu avec la fonction `doco`, qui représente léxecution parallèle d'une liste de processus.

On définit un pas de calcul atomique pour `p` comme l'éxecution d'un pas de calcul pour chacun des processus contenus dans la liste en argument de `doco`.

Les canaux communication sont des files.

```
type 'a in_port = 'a Queue.t
type 'a out_port = 'a Queue.t
```

On remarquera qu'on représente le blocage d'un processus `get qi` en retournant le même processus quand la file `qi` est vide.

Cette implémentation ne pose pas de problème majeur et produit un code très court.

3 En utilisant des processus et des tubes

On représente un processus par le type le plus simple :

```
type 'a process = unit -> 'a
```

Le parallélisme est implémenté avec l'utilisation de `Unix.fork`.

Les canaux de communication sont construits à l'aide des fonctions `Unix.pipe`, `Unix.in_channel_of_descr`, `Unix.out_channel_of_descr`.

```
type 'a in_port = in_channel
type 'a out_port = out_channel
```

Les types `in_channel` et `out_channel` permettent d'utiliser le module `Marshal` directement afin de transférer des valeurs de type arbitraire.

Cette implémentation ne fonctionne que si le nombre de processus est borné par une constante (nombre de processus que la machine peut gérer).

4 Parallélisation sur le réseau en utilisant des *sockets*

L'architecture proposée est la suivante :

- Le programme principal souhaite effectuer un calcul `p:'a process`. Pour cela il crée un serveur,
- Des programmes clients peuvent se connecter au serveur pour participer au calcul, (au moins un tel client est nécessaire au progrès du calcul)
- Les programmes clients peuvent être déconnectés à tout moment et le serveur principal doit pouvoir s'adapter à ces aléas.
- Les programmes clients n'ont pas de moyen de paralléliser des tâches. Le parallélisme est entièrement organisé par le serveur.

Pour cela on utilise le type suivant, inspiré de l'implémentation séquentielle.

```
type 'a process = int -> 'a result
and 'a result =
```

```
| Proc of 'a process
| Doco of unit process list * 'a process
| Res of 'a * int
| U
```

Laissons le constructeur U de côté pour l'instant.

Un processus est une fonction qui prend en argument `n : int`, et effectue au maximum `n` pas de calcul avant de retourner :

1. Si le calcul n'a pas terminé, la suite du calcul. Il y a deux cas de figure : ou bien le processus a tout simplement consommé tous les pas de calcul disponibles et il renvoie un `Proc`, ou bien il doit lancer des processus en parallèle, et il s'arrête avant et renvoie la liste des processus parallèles, ainsi que le calcul une fois que tous ces processus auront terminé. `Doco (l,p)` traduit le processus `bind (doco l) (fun () -> p)`.
2. Si le calcul est terminé, le résultat `Res` (avec le nombre de pas non consommés),

Par souci de typage, on impose que la transmission de tâches sous la forme de processus ne se fasse que pour des valeurs de type `unit process`.

Afin d'économiser la bande passante, le constructeur `U` sert d'alias au résultat `Res ((),_)` dont la première composante est constante et la deuxième superflue. (mais utile pour les calculs intermédiaires)

L'objectif visé par cette représentation est de permettre aux clients d'interrompre fréquemment le calcul.

Les processus constituent des tâches.

Les tâches sont exécutées en ordre FIFO. L'exécution peut être partielle, auquel cas la suite du calcul est replacée en fin de liste.

Les avantages recherchés sont les suivants :

- Le serveur peut sauvegarder les calcul effectués, pour avoir à en refaire le moins possible en cas de perte d'un client.
- Même avec un seul client connecté, aucun processus ne doit être suspendu indéfiniment. Si des processus peuvent terminer, notre implémentation garantit leur terminaison lors de l'exécution du programme, quels que soient le nombre de clients connectés et le nombre de tâches à accomplir.

Cela constitue implémentation particulièrement souple du modèle de calcul proposé. (au prix d'une performance réduite cependant)