Haskell : De nouvelles voies pour le parallélisme

Laurent Pierron

Haskell : De nouvelles voies pour le parallélisme

Laurent Pierron

Institut National de Recherche en Informatique et Automatique
INRIA Nancy Grand-Est, 615 rue du Jardin Botanique, 54600 Villers-les-Nancy, France

Exemple : résolution de grilles de Sudoku

- des problèmes de Sudoku sont dans un fichier texte
- une ligne du fichier représente un problème
- fonction (solve :: **String** −> **Maybe** Grid) résoud une grille

			4			8	7	
	4	7		9	2		5	
2			6				3	
9	7		5			2		3
5		8		2	4	7		6
6		4			7		8	5
	9		3		8			7
		3	2	4		1	6	
	1	2					9	

Programme séquentiel

- découpe le fichier de problèmes en lignes
- avec **map** appelle le solveur pour chaque problème (ligne)

```
main :: IO ()
main = do
    [f] <- getArgs
    grilles <- fmap lines (readFile f)

    let solutions = map solve grilles -- 1

    print (length (filter isJust solutions))
```

Parallélisation : monade Eval

- découpage de la liste des grilles en deux listes égales à une grille près
- évaluation de chaque liste en parallèle en **forçant** l'évaluation
- attente du résultat de chaque évaluation
- concaténation des résultats
- remplacement de la ligne **1** du programme séquentiel par

```
    (as,bs) = splitAt (length grilles `div` 2) grilles

    solutions = runEval $ do
                  as' <- rpar (force (map solve as))
                  bs' <- rpar (force (map solve bs))
                  rseq as'
                  rseq bs'
                  return (as' ++ bs')
```

- solution peu satisfaisante :
  - commandes de *bas niveau*
  - amélioration limitée à deux coeurs
  - traitement déséquilibré entre les deux listes

Parallélisation : fonction parMap

- écriture d'une fonction générale pour paralléliser la fonction *map*
  - fonction récursive
  - ligne **2** cas trivial de la liste vide
  - ligne **4** évaluation de la fonction en parallèle
  - ligne **5** appel récursif
  - ligne **6** construction de la liste résultat

```
1 parMap :: (a -> b) -> [a] -> Eval [b]
2 parMap f [] = return []
3 parMap f (a:as) = do
4     b <- rpar (f a)
5     bs <- parMap f as
6     return (b:bs)
```

- remplacement de la ligne **1** du programme séquentiel par un appel à notre fonction *parMap*

```
    solutions = runEval (parMap solve grilles)
```

- cette fois le code est élégant

Parallélisation : monade Strategies (parList)

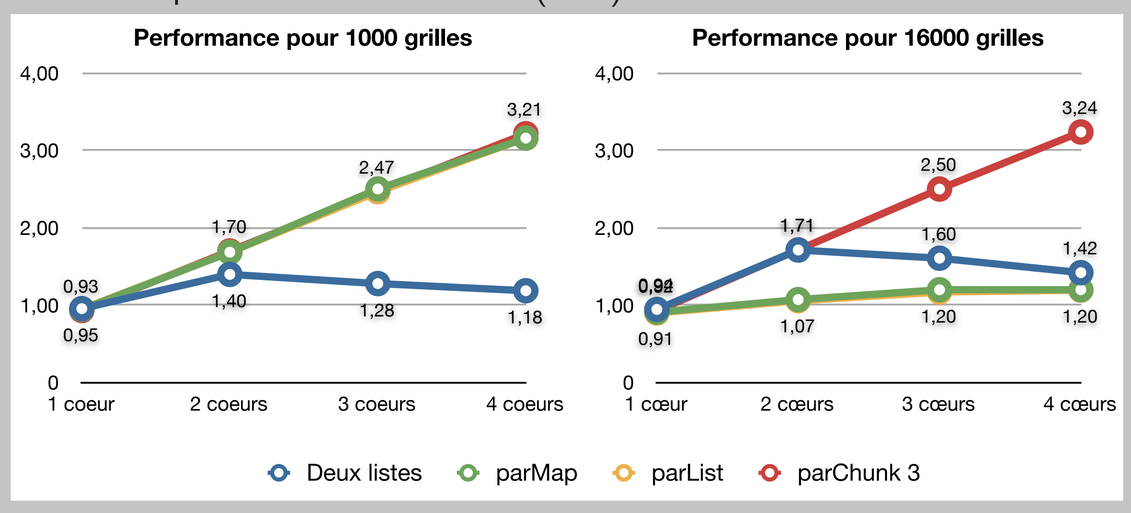
- fonction *parMap* a été généralisée dans la bibliothèque Control.Parallel.Stratégies qui définit :
  - fonctions de parcours de structures de données
  - stratégies d'évaluation combinables
- modification de la ligne **1** du programme séquentiel

```
    solutions = map solve grilles `using` parList rseq
```

- solution très élégante :
  - développement et test du code en séquentiel
  - ajout *d'annotations* pour paralléliser
  - permet de tester plusieurs stratégies de paraléllisation

Résultats sur quatre variantes de programme

- compilation : **ghc -O2 sudoku.hs -rtsopts -threaded**
  -threaded demande la parallélisation
- exécution : **sudoku sudoku17.1000.txt +RTS -N4**
  -N4 indique le nombre de coeurs (ici 4)



Concurrence : Software Transactional Memory

- un exemple de tranferts entre comptes en banque
- version Java multithreads

```
class Account {
    Int balance;
    synchronized void withdraw( int n ) {
        balance = balance - n; }
    void deposit( int n ) {
        withdraw( -n ); }
}
void transfer( Account from, Account to, Int amount ) {
    from.lock(); to.lock();
      from.withdraw( amount );
      to.deposit( amount );
    from.unlock(); to.unlock();
}
```

  - attention au **deadlock** dans **transfer**
  - délicat à mettre au point, car dépend de la charge en *threads*
- version Haskell STM

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount = do
    bal <- readTVar acc
    writeTVar acc (bal - amount)

deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)

transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically $ do
    deposit to amount
    withdraw from amount
```

  - −> **STM ()** : fonction utilisée dans mémoire transactionnelle
  - **atomically action** implique pour **action** :
    - **atomicité** : effets visibles de l'action d'un seul bloc
    - **isolation** : actions internes pas affectées par autres threads
  - pas de **deadlock** dans la version Haskell

Quelques autres modules pour le parallélisme

- **Par monad** : pour décrire les dépendances entre actions parallèles
- **Repa** : calcul haute performance sur tableaux multidimensions
- **Accelerate** : distribution de calculs principalement sur GPU
- **Async** : pour la concurrence asynchrone
- **forkIO, MVar** : pour la concurrence à un plus bas niveau que STM
- Pas de *solution magique* : bibliothèques adaptées par type de problème

Pour en savoir plus

- Haskell : `http://www.haskell.org`
- Parallel and Concurrent Programming in Haskell by Simon Marlow : `http://chimera.labs.oreilly.com/books/1230000000929`
- Beautiful concurrency by Simon Peyton-Jones : `http://research.microsoft.com/pubs/74063/beautiful.pdf`