Introduction à `Go` & Retours d'expérience

Sébastien Binet

CNRS/IN2P3/LPC

Juin 2016

Il n'y a plus de *"Free Lunch"* possible : la loi de Moore [1] n'est plus aussi aisée à mettre en œuvre que par le passé. Les développeurs de logiciels, ainsi que les scientifiques, doivent maintenant se familiariser avec la loi d'Amdahl [2]. En effet, la fréquence d'horloge des processeurs n'augmente plus avec chaque nouvelle génération : les processeurs gagnent de plus en plus de cœurs, mais sans changer la fréquence par rapport à la génération précedente.

L'augmentation du nombre de cœurs appelle à une utilisation plus répandue du parallélisme dans les logiciels développés par les communautés `HEP` et `Astro`. Traditionnellement, le parallélisme a été exploité au niveau du traitement des événements dans les logiciels de reconstruction et de simulation des expériences de physique des particules : il suffit de lancer N *jobs* sur les fermes de calcul, la Grille ou un *cloud*. Plusieurs événements peuvent être traités en parallèle, même si les algorithmes de reconstruction sont toujours appliqués de manière séquentielle à chacun de ces événements.

Cependant, chaque nœud de calcul hébergeant chacun de ces programmes, met à disposition des ressources finies (empreinte mémoire, *CPU*, descripteurs de fichiers, *sockets*, *E/S*, etc...), ressources dont le passage à l'échelle est plus problématique lorsque l'empreinte mémoire des *jobs* de reconstruction des expériences du LHC atteint 2 à 4 *Go* de `RAM`. Cette exploitation du parallélisme atteint ses limites avec les ordres de grandeurs du LHC.

Ainsi, il semble plus efficace d'effectuer le traitement parallèle des événements dans le même espace mémoire. Pour un programme écrit en `C++`, cela se traduit par l'utilisation de plusieurs *threads*. Malheureusement, la programmation parallèle multi-tâche (*multithreading*) est notoirement connue pour ses difficultés de mise en œuvre :

- il est très ardu d'écrire correctement une application *multithreadée*,
- il est également difficile de la garder correcte en fonction du temps,
- et aussi ardu de la garder efficace et optimisée au cours des développements successifs.

De plus, même si `C++11` [3] et `C++14` [4] apportent enfin la standardisation tant attendue des APIs de programmation parallèle (`std::thread`, `std::mutex`, `std::future`), cela se fait au prix d'une complexification encore plus poussée de ce langage, sans pourtant exposer une API de haut niveau. Il semble donc judicieux de se demander s'il n'existerait pas un nouveau langage de programmation mieux adapté à l'exploitation des architectures parallèles...

Pourquoi pas `Go` [5] ?

Anatomie de `Go`

`Go` est un langage de programmation, libre (licence `BSD-3`), initialement développé par Google et annoncé au monde en novembre 2009. C'est un langage compilé, avec gestion automatique de la mémoire *via* un ramasse-miettes (*garbage collector*), le support de l'introspection de type, des fonctions anonymes, des *closures* [1] et de la programmation orientée objet. La syntaxe de `Go` est une réminiscence de celle du `C`. Un bref aperçu d'un premier programme en `Go` est donné dans la figure 1.

```
1  // Command hello provides a greeting in french.
2  package main
3
4  import (
5      "fmt"
6  )
7
8  func main() {
9      fmt.Println("Bonjour la Lettre IN2P3.")
10 }
```

FIGURE 1: Un premier programme en `Go`.

`Go` est apprécié pour sa capacité à apporter au développeur, ainsi qu'à l'utilisateur final, le meilleur des mondes "dynamique" et "statique" :

- l'aisance de programmation des langages dynamiques grâce à sa verbosité limitée, son système d'inférence de type et un cycle de développement `edit-compile-run` extrêmement rapide,
- la robustesse d'un système de typage statique,
- la vitesse d'un exécutable compilé en langage machine.

De plus, le support de `Go` pour les interfaces ressemble fortement au *duck-typing* de `Python` [6] et se prête très bien à l'écriture de vastes cadriciels (*frameworks*) tels que ceux utilisés et développés pour les expériences au LHC.

Enfin, `Go` expose un support accru et intégré au langage, de la programmation concurrente *via* le modèle *CSP* (*Communicating Sequential Processes* [7]). En effet, il suffit de préfixer l'appel à une méthode avec le mot-clé `go` pour que son exécution s'effectue de manière concurrente aux autres fonctions, et ce, dans un *thread* léger appelé `goroutine`. Les `goroutines` sont multiplexées sur plusieurs

1. Les *closures* sont des fonctions qui capturent des références à des variables libres de l'environnement.

threads natifs : une `goroutine` bloquée, par exemple, par une opération d'`E/S` (disque, réseau, etc...), n'empêchera pas l'exécution des autres `goroutines` du programme. De plus, les `goroutines` sont très peu gourmandes en ressources grâce à leur pile de petite taille et ajustée à l'exécution. Ceci permet d'en lancer un très grand nombre et ce sur des machines très ordinaires : il n'est pas rare de voir des programmes comportant des milliers de `goroutines` sur des laptops aux capacités modestes, prouesse impossible avec des *threads* natifs.

Dans le langage `Go`, le mécanisme permettant d'échanger des données entre `goroutines` et ce sans remettre en cause la sûreté du système de typage (*type safety*), est appelé un `channel`. Envoyer et recevoir des données par un `channel` sont des opérations atomiques et permettent donc de synchroniser l'exécution des `goroutines`. Un exemple de communication entre `goroutines` est donné dans la figure 2 : la `goroutine` principale, `main`, lance deux `goroutines`, `gen` et `square`. `gen` génère la suite des nombres entiers de 0 à $+\infty$ et remplit le `channel in`. `square` extrait les nombres entiers de ce `channel` et remplit le `channel out` avec le carré de ces nombres. La `goroutine main` extrait les données du `channel out` et les affiche à l'écran.

```
1  package main
2
3  func main() {
4      in := make(chan int)
5      out := make(chan int)
6      go gen(in)
7      go square(out, in)
8      for i := range out {
9          println(i)
10     }
11 }
12
13 // gen generates numbers [0...)
14 func gen(ch chan int) {
15     i := -1
16     for {
17         i++
18         ch <- i
19     }
20 }
21
22 // square returns the square of each number.
23 func square(out, in chan int) {
24     for i := range in {
25         out <- i*i
26     }
27 }
```

FIGURE 2: Deux `goroutines`, `gen` et `square` se communiquent des données.

`Go` permet également d'appeler aisément du `C` *via* `cgo` [8]. Il suffit d'importer

le pseudo `package "C"` et d'indiquer les fichiers d'en-tête et bibliothèques pour utiliser les fonctions et types exposés par cette bibliothèque `C`. Un exemple est donné dans la figure 3.

```go
package main

// #cgo LDFLAGS: -lm
// #include <math.h>
import "C"

func main() {
    println("C.sqrt(4)=", C.sqrt(4))
}
```

FIGURE 3: Programme `Go` utilisant la fonction `sqrt` de la bibliothèque `C libm`.

Depuis la publication de la version `1.0` de `Go` en 2012, le langage est considéré comme stable et complètement rétrocompatible : chaque nouvelle version de `Go` (une tous les six mois en moyenne) compilera correctement un programme valide de 2012. Ce contrat de stabilité est également appliqué à la bibliothèque standard livrée avec le compilateur.

De par son origine et son ADN, `Go` permet de développer rapidement des programmes d'envergure. Son modèle de compilation d'éxecutables statiques permet également de les déployer aisément sur de grandes infrastructures : `Go` est rapidement devenu la *lingua franca* du développement *cloud*. Est-il adapté à l'environnement HEP et astro ?

Retours d'expérience

Analyse & simulation

`Go` a fait ses débuts dans HEP *via* la réimplémentation du *framework* de contrôle hors-ligne d'ATLAS et LHCb : GAUDI [9]. Cette réimplémentation, appelée simplement `fwk`, est regroupée sous l'ombrelle de l'organisation `go-hep` [10]. Le but principal de `go-hep/fwk` [11] était de démontrer la viabilité et l'adéquation de `Go` pour les *control frameworks* LHC, mais également de montrer l'aisance avec laquelle la programmation parallèle peut être mise en œuvre avec `Go`.

Un autre axe de travail était de montrer qu'un cadriciel concurrent comme `fwk` était non seulement adapté aux grosses expériences LHC avec leur infrastructure sous-jacente, mais était également adapté aux applications de plus petite taille telles qu'une analyse individuelle ou bien une bibliothèque de simulation. En effet, une des critiques récurrentes des physiciens vis-à-vis de GAUDI est sa lourdeur d'implémentation ainsi que la difficulté de l'installer sur une machine de bureau, ce qui en fait une plateforme de développement d'analyses peu séduisante, malgré sa robustesse et sa capacité à traiter les volumes de données du LHC.

Cet axe de travail a été concrétisé sous la forme de `fads` [12]. `fads` est la réimplémentation de DELPHES [13], un simulateur de détecteur de physique des

particules. DELPHES est un ensemble de composants `C++` basés sur ROOT dont le *design* ne se prête pas aisément à une implémentation *multithreadée*. Les détails de la comparaison entre les deux applications sont disponibles dans [14] : le passage à l'échelle de `fads` est nettement meilleur que DELPHES, tant en empreinte mémoire qu'en fréquence de traitement d'événements. Ces performances sont reportées dans la figure 4.

Au cours du développement de `fads`, plusieurs bibliothèques d'interfaçage avec l'existant (ROOT, `HepMC`, `HEPEVT`) ont dû être développées, ainsi que des bibliothèques d'analyse (histogrammes, *plot*s). Ce travail a permis de montrer que `Go` est un langage viable et compétitif dans le cadre du calcul et de l'analyse de données, et ce, même dans le microcosme HEP.

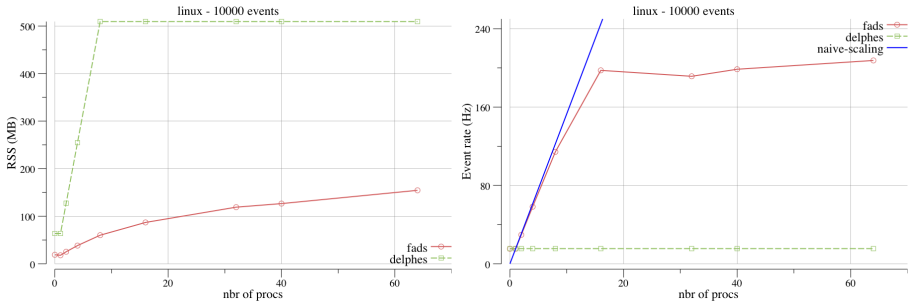

FIGURE 4: Empreinte mémoire (à gauche) et fréquence de traitement des événements (à droite). Résultats obtenus sur un serveur `Linux` de 40 cœurs (20 physiques). À noter que, passés 8 processus concurrents, le serveur n'avait plus assez de ressources (RAM) pour DELPHES.

Contrôle commande & monitoring

Un axe de recherche plus récent est l'investigation de `Go` et sa pertinence dans le monde du contrôle commande. Dans le cadre de l'expérience LSST, un banc de test pour la caractérisation des moteurs pour le changeur de filtres de la caméra devait être réalisé. Ces moteurs peuvent être commandés *via* plusieurs interfaces et protocoles (CANBUS, MODBUS, ...) : c'est le protocole MODBUS qui a été finalement retenu et mis en œuvre.

Malgré la jeunesse de `Go`, une bibliothèque prenant en charge le protocole MODBUS était déjà disponible sous licence libre, distribuée *via* `github` [15] et écrite par la communauté. Comme pour tous les `packages Go`, une simple commande a suffi pour installer cette bibliothèque :

```
$> go get github.com/goburrow/modbus
```

L'application permettant d'envoyer des commandes aux moteurs, de *monitorer* leurs positions, températures et autres grandeurs est en fait un serveur web écrit en `Go`. En effet, l'offre des GUIs en `Go` est encore limitée : même s'il existe des *bindings* vers la plupart des bibliothèques graphiques portables (Qt, GTK, ...), leur installation n'est pas aussi aisée qu'un `package` écrit totalement en `Go`. Il existe bien le début d'une bibliothèque en `Go`, mais elle est

encore en chantier [16]. La solution retenue a donc été l'écriture d'un exécutable servant une page web HTML5 utilisant `Polymer` [17] pour réaliser l'interface graphique. L'utilisateur envoie des commandes depuis l'interface, commandes qui sont ensuite relayées au serveur *via* des WebSockets. Le serveur se charge de la communication avec les moteurs et renvoie résultats des commandes, histogrammes, flux vidéo et grandeurs *monitorées* à l'utilisateur au moyen d'un autre WebSocket. L'authentification des utilisateurs, ainsi que la syndicalisation des flux et connexions, sont gérées côté serveur. Cette architecture orientée web permet *in fine* une grande transparence réseau, et ce, même pour des clients Windows. La figure 5 constitue un aperçu de l'interface graphique.

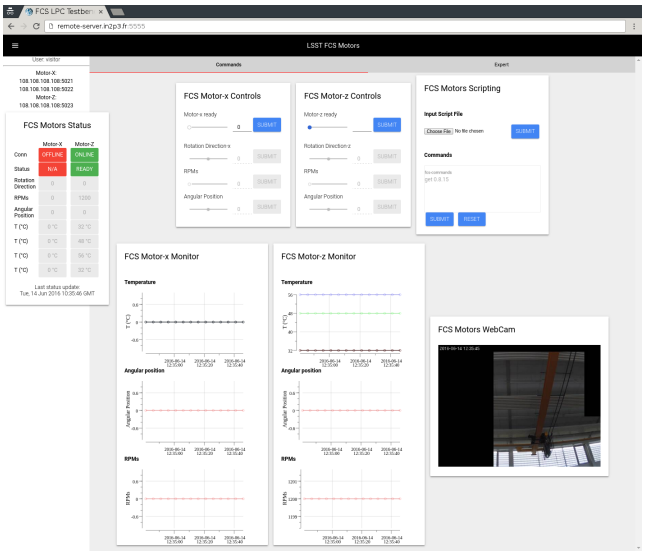

FIGURE 5: Interface graphique pour le banc de test du LPC. L'interface est en HTML5/`Polymer`, le serveur en `Go`.

Conclusions

Cet article a présenté le langage `Go`. Son approche pragmatique et sa volonté de rester simple (mais pas simpliste), couplées à son modèle de programmation concurrente, font de `Go` un langage résolument adapté à l'environnement plus hétérogène des machines multicœurs d'aujourd'hui et de demain.

Malgré son relatif jeune âge, `Go` comporte déjà la plupart des bibliothèques nécessaires à la programmation d'applications de calcul et d'analyse. Les outils intégrés à la chaîne de développement de `Go` permettent, de plus, de rapidement optimiser un code donné (CPU, mémoire, concurrence, I/O, ...). Enfin, l'empreinte mémoire réduite de `Go` par rapport à `Java` et ses facilités en programmation concurrente en font une alternative crédible dans le domaine de l'acquisition et *monitoring* de données, et le contrôle commande.

`Go` est d'ores et déjà le langage du *cloud*. Peut-être aura-t-il une vie dans HEP et en Astro ? Une chose est sûre : il a tous les atouts pour y arriver et supplanter `C++`, `Python` et `Java`.

Références

[1] Moore E, "Cramming more components onto integrated circuits", Electronics Magazine, 1965

[2] Amdahl G, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967

[3] The `C++11` programming language, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf`

[4] The `C++14` programming language, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf`

[5] `Go`, `https://golang.org`

[6] The `Python` programming language, `http://python.org`

[7] CSP, `http://en.wikipedia.org/wiki/Communicating_sequential_processes`

[8] `cgo`, `https://golang.org/cmd/cgo/`

[9] Mato P 1998 GAUDI-architecture design document Tech. Rep. LHCb-98-064 Geneva

[10] The `go-hep` project, `https://github.com/go-hep`

[11] The `go-hep/fwk` concurrent control framework, `https://github.com/go-hep/fwk`

[12] `fads` : a Fast Detector Simulation toolkit, `https://github.com/go-hep/fads`

[13] DELPHES 3 : a modular framework for fast simulation of a generic collider experiment, `arXiv:1307.6346`

[14] `fads` @ HEP Software Foundation workshop, `https://indico.cern.ch/event/357737/contributions/1770401/`

[15] `https://github.com/goburrow/modbus`

[16] `https://github.com/golang/exp/tree/master/shiny`

[17] `https://www.polymer-project.org/`