Can Go address the multicore issues of today and the manycore problems of tomorrow?

Sébastien Binet

Laboratoire de l'Accélérateur Linéaire, Université Paris-Sud XI, 91898, Orsay, FR

E-mail: binet@cern.ch

Abstract. Current High Energy and Nuclear Physics (HENP) libraries and frameworks were written before multicore systems became widely deployed and used. From this environment, a 'single-thread' processing model naturally emerged but the implicit assumptions it encouraged are greatly impairing our abilities to scale in a multicore/manycore world. While parallel programming - still in an intensive phase of *R&D* despite the 30+ years of literature on the subject - is an obvious topic to consider, other issues (build scalability, code clarity, code deployment and ease of coding) are worth investigating when preparing for the manycore era. Moreover, if one wants to use another language than C++, a language better prepared and tailored for expressing concurrency, one also needs to ensure a good and easy reuse of already field-proven libraries. We present the work resulting from such investigations applied to the Go programming language. We first introduce the concurrent programming facilities Go is providing and how its module system addresses the build scalability and dependency hell issues. We then describe the process of leveraging the many (wo)man-years put into scientific FORTRAN/C/C++ libraries and making them available to the Go ecosystem. The ROOT data analysis framework, the C-BLAS library and the Herwig-6 MonteCarlo generator will be taken as examples. Finally, performances of the tools involved in a small analysis written in Go and using ROOT I/O library will be presented.

1. Introduction

The *"Free Lunch"* is over: Moore's law [1] can not be as easily leveraged as in the past, computer scientists and software writers have now to be familiar with Amdahl's law [2]. Indeed, computers are no longer getting faster: instead, they are growing more and more CPUs, each of which is no faster than the previous generation.

This increase in the number of cores evidently calls for more parallelism in HENP software. Fortunately, typical HENP applications (event reconstruction, event selection,...) are usually *embarrassingly parallel*, at least at the coarse-grained level: one "just" needs to call in parallel the portion of code which massages the events retrieved from the detector (the event loop) while still executing sequentially all the code processing each event.

However, this *'one event per core'* strategy may not scale up to hundreds or thousands of cores, thus requiring library implementers and code writers at large to deal with sub-event parallelism. Parallel programming in C++ has been done and is doable, but even if many libraries such as Threading Building Blocks [6] or OpenMP [7] exist to help the burden, such an endeavour is still quite tedious and error prone. Note that C++11 with lambda functions, std::thread, std::future and std::promise will likely improve the situation but at the price

of sprinkling templates everywhere in the code, slowing down further the edit-compile-run cycle, and complicating further an already not so simple language. The recent gain in popularity of `python` [4] in `HENP` is probably a consequence of these two `C++` defects, even if `python` is probably not the best language to tackle parallel high performance computing. At this point, it would seem reasonable to ask if using a new language more capable at leveraging multithreaded environments would be a better alternative.

This paper explores such a path. We first introduce some of `Go`most relevant features with regard to concurrency and how its module system addresses the build scalability and dependency hell issues. We then describe the process of leveraging the many (wo)man-years put into scientific FORTRAN/`C`/`C++` libraries and making them available to the `Go` ecosystem. The `ROOT` [8] data analysis framework, the `C-BLAS` [9] library and the `Herwig-6` [10] MonteCarlo generator will be taken as examples. Finally, performances of a small analysis written in `Go` and using FORTRAN and `C++` libraries will be discussed.

2. New languages

Since `HENP` and `C++` met to produce (among other projects) GAUDI [11] and `ROOT` [8], the language landscape greatly changed. Many new languages appeared or became *"mainstream"* and, while closely following the language trend was not achieved, some adaptations were performed. For example, most of the GAUDI configuration and steering code is nowadays written in `python` and most, if not all, `C++` components (from `ROOT` and GAUDI) are also available from `python`. But `python` (or more precisely `CPython`) has well-known scalability issues in a multithreaded environment because of its *Global Interpreter Lock (GIL)* which serializes access to `python` objects [1]. Moreover, even if this issue can be worked around by writing `C` extension modules, having an event loop in an interpreted language is not the best bet CPU-speed wise.

We investigated `Go` as a possible alternative to `C++`.

3. Elements of `Go`

`Go` [5] is a new open source language from Google, first released in November 2009. It is a compiled language with a garbage collector and builtin support for reflection, first-class functions, closures and object-oriented programming. The obligatory *"hello world"* program can be found in figure 1.

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, world")
}
```

Figure 1. The obligatory *"hello world"* program, in `Go`.

`Go` is lauded to bring the best of both dynamic and static worlds:

- the feel of a dynamic language, thanks to its limited verbosity, its type inference system and its fast compile-edit-run cycle,
- the safety of a static type system,
- the speed of a machine compiled language. [2]

[1] Other `python` implementations (`JPython`, `IronPython`,...) do not present this limitation.

[2] the aim of the `Go` authors is to eventually bring the performances of a `Go` binary within 10% of `C`.

Moreover, Go support for interfaces which resembles the *duck-typing* motto of python fits nicely into HENP frameworks like GAUDI. Finally and more importantly, Go has language support for concurrency, following the *Communicating Sequential Processes (CSP)* [12] model: prefixing a method or function call with the keyword go will spawn off a goroutine: the function will be executed concurrently to other codepaths. goroutines are multiplexed onto multiple OS threads so blocked goroutines because of a non-finished I/O operation will not halt the execution of the others. Furthermore, goroutines are lightweight thanks to their variable stack size, starting small and growing as needed.

In Go, the typesafe mechanism to exchange data between goroutines, is a *channel*. Sending or receiving data on a channel is atomic and can thus be used as a synchronization mechanism. It should be noted that as of 2011, Go is lacking a few features which would probably make the implementation of typical HENP frameworks a bit easier, such as dynamic libraries and dynamic code loading. Another set of missing features more important for efficient scientific code is the lack of generics [3] and, for mathematical code clarity, the lack of operator overloading.

Go programs are constructed by linking together *packages*, whose properties allow efficient management of dependencies. Each package may in turn use facilities provided by other packages. From these inter-package dependencies, forming an acyclic dependency graph, a correct compilation order can be inferred. Within a package, all global variables, functions and types defined in that package are visible. From the outside, only the exported ones are available. In Go, the rule about visibility of information is simple: if a name (of a top-level type, function, method, constant or variable, or of a structure field or method) is capitalized, users of the package may see it. Otherwise, the name and hence the thing being named is visible only inside the package in which it is declared. This is more than a convention: the rule is enforced by the compiler. Each compiled package file imports transitive dependency informations. If A.go depends on B.go which itself depends on C.go, compiling the package A will result in first compiling C.go, B.go and then A.go. And to recompile A.go, the compiler will only have to read the object file B.o but not C.o. At scale, this can result in huge speedups: the improvements become exponential.

Third-party Go libraries and programs are also easily distributed *via* goinstall, an executable providing automatic package installation with dependencies' tracking. goinstall is the *de facto* standard to compile and distribute Go code [4]. An author wanting to distribute Go code just needs to publish it on some repository (bitbucket, launchpad, github, ...) and clients will be able to check out, compile and install it in one go, without special permissions, like in the shell session of figure 2.

```
shell> goinstall -v bitbucket.org/binet/go-croot/pkg/croot
goinstall: big: skipping standard library
goinstall: fmt: skipping standard library
[...]
goinstall: bitbucket.org/binet/go-ctypes/pkg/ctypes: gomake -f- install
[...]
goinstall: bitbucket.org/binet/go-croot/pkg/croot: gomake -f- install
```

Figure 2. Installing libraries or applications with goinstall. The croot Go package is being goinstalled. Note how goinstall automatically installed the ctypes package, a dependency of croot.

[3] also called templates in C++.

[4] goinstall is being refactored into a family of go binaries providing compilation and installation facilities in view of a frozen and long term supported Go-1 version.

Then, clients can import these packages like any other, with the correct import path, as shown in figure 3.

```
package mypackage

import "bitbucket.org/binet/go-croot/pkg/croot"
// croot can now be used like any package:
// file := croot.OpenFile(...)
```

Figure 3. Importing and using third-party packages, using the correct import path.

Finally, a categorized list of all `goinstall`-able packages and commands is kept on the `godashboard` [13], providing the last piece for an effective `CPAN`-like facility.

4. Wrapping foreign language libraries

To paraphrase Tim Mattson, *"successful new languages build on existing languages and where possible, support legacy software. C++ grew out of C. Java grew out of C++."* `Go` is no exception and provides a standard way to access `C` libraries: `cgo`.

4.1. Wrapping `C` libraries

`cgo` allows to access and use entities defined in a `C` library with limited effort, as shown in figure 4.

```
package myclib
// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func foo(s string) {
  c_str := C.CString(s)  // create a C string from a Go one
  C.fputs(c_str, C.stdout)
  C.free(unsafe.Pointer(c_str))
}
```

Figure 4. Wrapping `C` `stdio` with `cgo`.

Wrapping bigger libraries, such as `C-BLAS`, is no different albeit quite repetitive and mechanical a task, which should probably be automatized, using `HENP` tools like `genreflex` or `rootcint`, or using a `SWIG` [14] interface.

4.2. Wrapping Fortran *libraries*

Wrapping Fortran 2003 code providing an `ISO C` interface is similar to the simple `C` library case. However, Fortran 77 libraries are more common in `HENP` and are dealt with the usual way:

- write a set of `C` wrappers to access the needed functionalities from the Fortran library,
- write `cgo` code to wrap the `C` wrappers.

```
package cblas

/*
 #include <complex.h>
 #cgo LDFLAGS: -lcblas
 #include "cblas.h"
*/
import "C"
import "unsafe"

/*
 C signature:
 float  cblas_sdsdot(const int N, const float alpha,
                     const float *X, const int incX,
                     const float *Y, const int incY);
 */
func Sdsdot(alpha float32, x, y []float32) float32 {
        if len(x) != len(y) {
                panic("slices' size differ")
        }

        c_N := C.int(len(x))
        c_alpha := C.float(alpha)

        c_X := (*C.float)(unsafe.Pointer(&x[0]))
        c_incX := C.int(1)

        c_Y := (*C.float)(unsafe.Pointer(&y[0]))
        c_incY := C.int(1)

        return float32(
                C.cblas_sdsdot(c_N, c_alpha,
                c_X, c_incX,
                c_Y, c_incY))
}
```

Figure 5. Wrapping `C-BLAS` with `cgo`.

Again, a tool automatizing writing this kind of repetitive and mechanical code would greatly ease the wrapping task. There is no technical showstopper to implement such a tool and evidence with `f2py`, a tool to automatically create `python` bindings for FORTRAN 77 code, shows it is indeed possible to generate the needed boilerplate code.

4.3. Wrapping `C++` libraries

Wrapping `C++` is a bit more involved a task. The current solution is to use `SWIG` and a `SWIG` interface file to generate `cgo` code. The `Go` backend of `SWIG` is sophisticated enough to handle and map `C++` constructs not existing in `Go`, back to idiomatic `Go` code. For example, overloaded functions are dispatched from a common `Go` variadic function to each `cgo`-wrapped function; while multiple inheritance is handled by defining multiple `Go` interfaces; and virtual methods are implemented by methods on interfaces. And thanks to `SWIG` machinery, it is even possible to implement a `C++` abstract class with a `Go` struct implementing the according interface.

```
/* c-hepevt/hepevt.h */

#ifdef __cplusplus
extern ''C'' {
#endif

/** event number
 */
int
hepevt_event_number();

#ifdef __cplusplus
}
#endif

/* c-hepevt/hepevt.cxx */
extern ''C'' {
  extern struct {
    char data[hepevt_bytes_allocation];
  } hepevt_;
}
#define hepevt hepevt_

static int
byte_num_to_int(unsigned int b)
{
  // consistency checks
  // ...
  if (s_sizeof_int == sizeof(short int)) {
    short int *si = (short int*)&hepevt.data[b];
    return (int)(*si);
  }
  // error handling
  // ...
  return 0;
}

/** event number
 */
int
hepevt_event_number()
{
  return byte_num_to_int(0);
}
```

Figure 6. Excerpts of the `C` wrapper around `HEPEVT`.

However, `SWIG` does not use a proper `C++` compiler to parse the `C++` header files and thus can not handle all of `C++03` constructs. Attempts at wrapping `ROOT`'s `TObject` class for this paper failed, even after trying numerous tricks like presenting a preprocessed `TObject.h` file to `SWIG`

or injecting `#ifdef SWIG` preprocessor directives [5].

A more versatile tool, built on top of `gccxml` or `clang`, would resolve this issue. In the meantime, `Go` can still leverage `ROOT` libraries, resorting to the same recipe than in the FORTRAN77 case: write a `C-API` on top of the `C++` one and write `cgo` code to wrap the former.

5. Results

To test the performances and applicability of using `Go` with *"legacy"* `C++` code, the transliteration of the canonical `ROOT TTree` reading example from the `ROOT` tutorial has been performed. The code is provided on a `Mercurial` repository [15]. The same exercize has been done using the `C-API croot` calling the usual `ROOT C++` code to infer the overhead induced by this additional layer, and disentangle the contribution from the pure `cgo` overhead. Results are reported in figure 7.

```
shell> time ttree-ex-1
 29.04s user  1.03s system 86% cpu 34.83 total (C++)
 29.12s user  1.09s system 85% cpu 35.48 total (CRoot)
 44.83s user  1.24s system 87% cpu 54.36 total (go-croot)

shell> uname -a
Linux farnsworth 3.0-ARCH #1 SMP PREEMPT
x86_64 Intel(R) Core(TM)2 Duo
CPU T9400 @ 2.53GHz GenuineIntel GNU/Linux
```

Figure 7. Timings obtained for the `ROOT TTree` canonical example.

These timings show less than a factor two overhead with regard to the `C++` version. One should note that these timings were obtained with the 64bits version of the `gc` compiler (`weekly.2011-09-01 9619`). We should note that, while performances of the `gc` compiler are expected to improve, using the `gccgo` compiler would be a better match as far as pure number crunching and calling `C` functions are concerned. Indeed, `gccgo` can on one hand leverage the full set of optimizations that `GCC` can apply and on the other hand has smaller overhead calling `C` functions [6]. Nonetheless, we kept using `gc` as this is the compiler implementing all of the `Go` features.

6. Conclusions

We presented the work on investigating `Go` as a viable `C++` replacement for the multi-core era. `Go` has many builtin capabilities to ease the burden of exposing and leveraging concurrency in `HENP` applications as previously shown in [16]. The short development cycle together with the code distribution platform and tools provide `Go` with a short *'time-to-market'* length.

Wrapping foreign language libraries, and especially `C` libraries, is made easy thanks to `cgo`, a tool shipped with the `Go` distribution. Wrapping `C++` is possible - albeit cumbersome - with `SWIG` but is subject to `SWIG`'s limitations and does not seem to be applicable to large libraries such as `ROOT`. An automatized solution built on top of `gccxml` or `clang` would greatly improve the ability of `Go` to reuse *'legacy'* libraries.

[5] While this could theoretically be a solution for `SWIG`'s preprocessing troubles with some macros in `ROOT` code although not a particularly aesthetical one, this would not address `SWIG`'s limitations in handling `<` and `>` operators.

[6] `gc` has a different calling convention than `gcc`, so `cgo` needs to perform additional work when calling `C` functions. `gccgo`, for obvious reasons, has the same calling convention than `gcc`.

References

[1] Moore E 1965 Cramming more components onto integrated circuits *Electronics Magazine* vol 38 no 8 pp 114-117

[2] Amdahl G 1967 Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities *AFIPS Conference Proceedings* pp 483-485

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

[4] The `python` programming language `http://python.org`

[5] The `Go` programming language `http://golang.org/`

[6] Intel Threading Building Blocks `http://threadingbuildingblocks.org`

[7] `OpenMP` (Open Multi-Processing) `http://openmp.org`

[8] The `ROOT` framework, `http://root.cern.ch`

[9] Lawson C L, Hanson R J, Kincaid D, and Krogh F T 1979 Basic Linear Algebra Subprograms for FORTRAN usage *ACM Trans. Math. Soft* pp 308-323 `http://www.netlib.org`

[10] Corcella G, Knowles I G, Marchesini G, Moretti S, Odagiri K, Richardson P, Seymour M H and Webber B R 2001 HERWIG 6.5 *JHEP 0101* (*Preprint* `hep-ph/0011363`)

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

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

[13] The `Go` dashboard `http://godashboard.appspot.com/project`

[14] `SWIG` `http://swig.org`

[15] `https://bitbucket.org/binet/go-io-benchmarks`

[16] Binet S 2010 ng: what next-generation languages can teach us about HENP frameworks in the manycore era *J. Phys.: Conf. Ser.* **331** 042002