ng: What next-generation languages can teach us about HENP frameworks in the manycore era

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) frameworks were written before multicore systems became widely deployed. A 'single-thread' execution model naturally emerged from that environment, however, this no longer fits into the processing model on the dawn of the manycore era. Although previous work focused on minimizing the changes to be applied to the LHC frameworks (because of the data taking phase) while still trying to reap the benefits of the parallel-enhanced CPU architectures, this paper explores what new languages could bring to the design of the next-generation frameworks.

Parallel programming is still in an intensive phase of R&D and no silver bullet exists despite the 30+ years of literature on the subject. Yet, several parallel programming styles have emerged: actors, message passing, communicating sequential processes, task-based programming, data flow programming, . . . to name a few.

We present the work of the prototyping of a next-generation framework in new and expressive languages (python and Go) to investigate how code clarity and robustness are affected and what are the downsides of using languages younger than FORTRAN/C/C++.

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, the strategy devised and implemented in AthenaMP [6] where the fork system call and the *Copy-On-Write (COW)* mechanism were leveraged in order to save memory footprint and use multiple cores will probably not scale up to manycore systems as *COW*'s efficiency is bounded as well as the amount of *RAM* available on a given machine. Indeed, the amount of physical memory associated to a core will not scale with the increasing number of cores. A 'one-event/one-core/one-process' strategy, even if GNU/Linux has this codepath optimized, will bring the machine on its knees when thousands of cores will be available, especially if each of these processes perform a non-negligible amount of (possibly chaotic) I/O.

Therefore, it seems more efficient to have at least the event-level [1] data parallel processing being performed in the same address space. In a `C++` world, this means multithreading and raises all the issues already noted during the development of `AthenaMT` [6]:

- it is hard to get a multithreaded application right,
- hard to keep it right,
- hard to keep it efficient and optimized across releases.

Even if the next version of `C++` [8] will improve the situation with `lambdas`, `std::future` and `std::thread`, at least on the standardization and portability fronts, this will be achieved at the cost of complicating further the language. At this point, it would seem reasonable to ask if using a new language more capable at leveraging multithreaded environments would be a more reasonable alternative.

This paper explores such a path. We first recall the basic architecture of the Gaudi [3] framework to identify the main components which would need modifications in a multithreaded environment. Then, after motivating why we chose `Go`, we will introduce some of its most relevant features with regard to concurrency and how these have been translated into a new `Go`-based framework, `ng-go-gaudi`. Finally, after having presented scalability results, we will draw some conclusions and propose ideas on future work and possible improvements to `ng-go-gaudi`.

2. Athena/Gaudi refresher

Gaudi [3] is an object-oriented `C++`-based software framework built around the *Component Object Model (COM)* [7]. *Data* objects (event data, detector data or statistical data) are recorded into and retrieved from a component: the *data store*. *Algorithm* objects are the components which manipulate this data or create new and more refined data quantities by interacting with the *data store*. The creation of these algorithms and proper state transitions are ensured and orchestrated by a central service, the `ApplicationManager`, while the in-order scheduling of the algorithms is managed by the `EventLoopManager`, as schematized in figure 1.

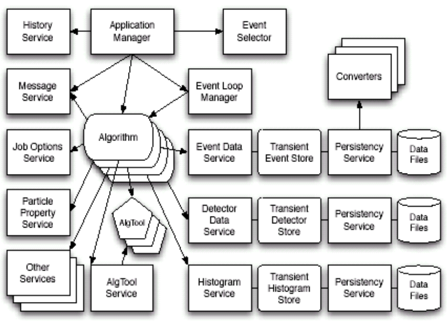

Figure 1. Overview of the Gaudi framework. At the center are the multiple algorithms, interacting with many core services (`Event Data Service`, `JobOptionsSvc`,...) and being scheduled by the `EventLoopManager` which is itself steered by the `ApplicationManager`.

```
class IAlgorithm : public IInterface {
  public:
    virtual StatusCode initialize() = 0;
    virtual StatusCode execute() = 0;
    virtual StatusCode finalize() = 0;
};
```

Figure 2. `C++ Algorithm` interface.

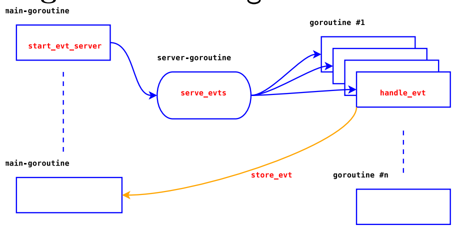

Figure 3. Overview of the parallelized event processor.

As can be seen in figure 1, the workhorse component is the `Algorithm` one whose (simplified) interface is reported in figure 2. The `execute()` method is called for each event and usually

[1] *i.e.:* as opposed to *e.g.* a conditions-level data

involves retrieving data from the event store as well as recording new more refined data in that event store. Previous work [6] focused on maintaining that interface, while modifying the framework behind the scene to leverage the `fork()` and *COW* mechanisms to transparently parallelize the GAUDI application at the event level.

Keeping these key elements in mind, we now investigate what a GAUDI-like framework would probably look like if it were written in a new more parallel- or concurrent-friendly language.

3. New languages

Since `HENP` and `C++` met to produce (among other projects) GAUDI and `ROOT` [9], the language landscape greatly changed. Many new languages appeared or became *"mainstream"* and, while closely following the 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 [2]. 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.

Other languages such as HASKELL [10] have been considered for this study. Indeed, functional languages are a great substrate for automated code parallelization [11] thanks to their *"no side effects"* [3] property, sidestepping race conditions. However, functional programming languages are probably not yet fitting into the average physicist software toolbox, with problems on their own (such as space leaks stemming from the laziness of HASKELL) lacking proper tools to debug, and were thus discarded from this study. `Vala` [12] was considered because of its support for interfaces, which match very well the GAUDI COM architecture, as well as for its ability to asynchronously start tasks and co-routines. However, the lack of documentation and the fact that "only" GNOME is using this language, disqualified it for this study.

We were hence left with `Go`.

3.1. 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.

`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. [4]

Moreover, `Go` support for interfaces which resembles the *duck-typing* motto of `python` fits nicely into the GAUDI framework. Finally and more importantly, `Go` has language support for concurrency, modelled after the *Communicating Sequential Processes (CSP)* [13] 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.

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

[3] this is true for the so-called *pure* functional languages.

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

In `Go`, the typesafe mechanism to exchange data between `goroutines`, is called *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 2010, `Go` is lacking a few features which would probably make the current implementation of `ng-go-gaudi` 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 [5] and the lack of operators overloading.

4. `ng-go-gaudi` implementation

`ng-go-gaudi` is a `Go` implementation of a minimal framework modeled after Gaudi.

The current implementation can be found in a `Mercurial` repository [14] and holds:

- an application manager, an event processor, and a data store service,
- base classes for algorithms with support for messaging and configuration *via* properties,
- a simple `JSON` output stream and a simple `Go` bytestream (`gob`) output stream,
- and few simple test algorithms (`adder`, `counter`, ...).

4.1. Parallelizing the event loop

Leveraging the *embarassingly parallel* nature of the typical `HENP` application, the event processor was parallelized, following the ideas of `AthenaMP` and `AthenaMT`. The overall architecture of this parallelization is schematized in figure 3 and the code to achieve it is reproduced in figure 4. Lines 23 to 33 setup a server `goroutine` which will take a buffered [6] input queue of events to process and (eventually) asynchronously call an event handling function to process the events. These processed events will then be removed from the input queue and will appear on the output one. Following a typical concurrent `Go` pattern, a third *channel*, the `quit` one, is also created to notify clients when the event source is done, allowing to cleanly terminate the event processing. be cleanly terminated.

As shown in figure 3, each event is processed by a dedicated `goroutine`, so the event processing is concurrent. This means each `goroutine` needs its own data store and thus each algorithm needs to know which data store it should interact with. To fulfill that requirement, the Gaudi algorithm interface had to be extended to encode the data provenance and make the event context explicit, as shown in figures 6 and 7.

4.2. Parallel `I/O`

`JSON` and `gob` output streams have been implemented to study the feasability of a parallel `I/O` persistency system. In each case, the data is transfered from the data store to the concrete output stream via *channel*s, that data is then owned by *the* `goroutine` commiting it to disk.

The `JSON` service implementation of the `NewOutputStream` method of figure 5 follows the typical `Go` pattern already described for the parallel event loop where three channels are created (input, errors and quit) and a `goroutine` which polls on each of these channels. The `JSON` output handle is then handed these three channels to pump data in, as shown in figure 8.

4.3. Job configuration and results

As mentioned previously, current (2010) `Go` does not support dynamic code loading [7]. This can be worked around by leveraging the fast compilation of `Go` code. Indeed, `ng-go-gaudi` job

[5] also called templates in `C++`.

[6] The input queue is buffered to limit the number of in-flight events.This can of course be configured at the command line level.

[7] This limitation should be lifted in a future `Go` version.

```go
func (self *evtproc)
mp_NextEvent(evtmax int) kernel.Error {
  handle := func(evt *evtstate,
                 out_queue chan <- *evtstate) {
    evt.sc = self.ExecuteEvent(evt)
    out_queue <- evt
  }

  serve_evts := func(
    in_evt_queue <- chan *evtstate,
    out_evt_queue chan <- *evtstate,
    quit <- chan bool) {
    for {
      select {
        case ievt := <-in_evt_queue:
          go handle(ievt, out_evt_queue)
        case <-quit:
          return
      }
    }
  }

  start_evt_server := func(nworkers int)
    (in_evt_queue,
     out_evt_queue chan *evtstate,
     quit chan bool) {
    in_evt_queue = make(chan *evtstate, nworkers)
    out_evt_queue = make(chan *evtstate)
    quit = make(chan bool)
    go serve_evts(in_evt_queue, out_evt_queue,
                  quit)
    return in_evt_queue, out_evt_queue, quit
  }

  in_evt_queue, out_evt_queue, quit \
    := start_evt_server(self.nworkers)
  for i:=0; i<evtmax; i++ {
    in_evt_queue <- new_evtstate(i)
  }
  // ...
  return kernel.StatusCode(0)
}
```

Figure 4. Go code realizing the parallelization of the event loop.

```go
package kernel

/// handle to a concurrent output stream
type IOutputStream interface {
  /// write (and possibly commit)
  /// data to the stream
  Write(data interface{}) Error
  /// closes and flushes the output stream
  Close() Error
}

/// interface to a concurrent output
/// stream server
type IOutputStreamSvc interface {
  /// returns a new output stream
  NewOutputStream(stream_name string)
    IOutputStream
}
```

Figure 5. Go interfaces for Gaudi-I/O.

```go
package kernel

type IAlgorithm interface {
  Initialize() Error
  Execute(ctx IEvtCtx) Error
  Finalize() Error
}
```

Figure 6. Extended IAlgorithm interface.

```go
package testalg

import "gaudi/kernel"

type myalg struct {
  kernel.Algorithm
}

func (self *myalg)
Execute(ctx kernel.IEvtCtx) kernel.Error {
  store := self.EvtStore(ctx)
  store.Put("foo", 42)
  return kernel.StatusCode(0)
}
```

Figure 7. Example of accessing a particular data store in client code.

```go
package outstream
import "json"
/// output stream using JSON as a format
type json_outstream_handle struct {
  svc kernel.IService
  w *os.File
  enc *json.Encoder
  data chan interface{}
  errs chan os.Error
  quit chan bool
}

func (self *json_outstream_handle)
Write(data interface{}) kernel.Error {
  self.data <- data
  select {
    case err := <-self.errs:
      return kernel.StatusCodeWithErr(1, err)
    default:
      return kernel.StatusCode(0)
  }
  return kernel.StatusCode(0)
}
```

Figure 8. JSON concrete backend.

configuration is performed by running a `python` script which will generate `Go` code compiled down to an executable holding the concrete list of all components to be used at runtime [8].

An excerpt of such a job configuration, scheduling 1000 algorithms (incrementing integers and displaying them) multiplexed on 5000 `goroutines`, can be seen in figure 9 which, by varying the number of cores being used at runtime leads to the performance plot in figure 10. The scalability problem which can be observed has been attributed after inspection of the `goroutines` profiles to mutex bottlenecks mainly at the messaging layer: each component can print messages on screen

[8] In C++ Gaudi, this is done via `dlopen` and a plugin manager.

at various verbosity levels, but as the standard output is shared between all these components, a contention appears on this resource.

```
app.props.EvtMax = 10000
app.props.OutputLevel = 1

app.svcs += Svc("gaudi/kernel/evtproc:evtproc",
                "evt-proc",
                OutputLevel=Lvl.INFO,
                NbrWorkers=5000)

app.svcs += Svc("gaudi/kernel/datastore:datastoresvc",
                "evt-store")
app.svcs += Svc("gaudi/kernel/datastore:datastoresvc",
                "det-store")

for i in xrange(500):
    app.algs += Alg("gaudi/tests/pkg2:alg_adder",
                    "addr--%04i" % i,
                    SimpleCounter="my_counter")
    app.algs += Alg("gaudi/tests/pkg2:alg_dumper",
                    "dump--%04i" % i,
                    SimpleCounter="my_counter",
                    ExpectedValue=i+1)
```

Figure 9. `python` code used to configure an `ng-go-gaudi` job.

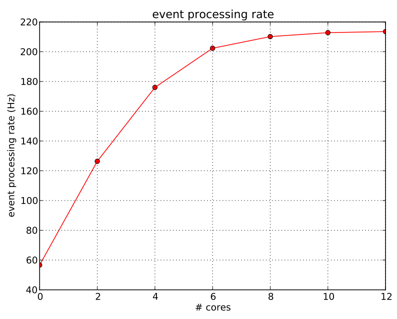

Figure 10. Event processing rate of an `ng-go-gaudi` application when varying the number of used cores.

5. Conclusions

We presented a prototype of a Gaudi-like framework written in `Go` to investigate what next generation frameworks could look like when leveraging new languages better tailored at exploiting concurrency and parallelism. A concurrent event loop manager and concurrent output streams were implemented for this study. Even if some performance problems were uncovered - which could easily be addressed by redesigning the `gaudi/msgstream` to reduce contention on `stdout` or by reducing the garbage collector pressure through a tighter integration with the event loop model of Gaudi - the implementation of concurrent patterns and the ability to compose them was greatly eased by `Go` primitives and builtin support. We propose to further continue the prototyping of `ng-go-gaudi` and address its shortcomings. Future work will also investigate the feasibility to develop sub-event concurrency *e.g.* by instrumenting the `datastore` accesses to infer a dataflow to allow the parallel execution of algorithms, and explore ways to improve the memory locality of our big software framework applications - an obvious way would be to break the single application into a flock of smaller more specialized ones.

References

[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] Mato P 1998 Gaudi-architecture design document Tech. Rep. LHCb-98-064 Geneva
[4] The `python` programming language, `http://python.org`
[5] The `Go` programming language, `http://golang.org/`
[6] Binet S, et al. "Harnessing multicores: strategies and implementations in Atlas", CHEP, 2009
[7] COM, `http://en.wikipedia.org/wiki/Component_Object_Model`
[8] The `C++` programming language, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf`
[9] The `ROOT` framework, `http://root.cern.ch`
[10] The `Haskell` Programming Language, `http://www.haskell.org`
[11] Harris T, Singh S, "Feedback Directed Implicit Parallelism", ICFP, 2007
[12] `Vala`, `http://live.gnome.org/Vala`
[13] CSP, `http://en.wikipedia.org/wiki/Communicating_sequential_processes`
[14] `ng-go-gaudi mercurial` repository, `http://bitbucket.org/binet/ng-go-gaudi`