Computational Astrophysics with Go

Pramod Gupta,[1]

[1]*Department of Astronomy, University of Washington, Seattle, Washington, USA;* psgupta@uw.edu

Abstract. Go is a relatively new open-source language from Google. It is a compiled language and so it is quite fast compared to interpreted languages. Moreover, it is based on a design principle of simplicity. In this paper, I discuss the suitability of Go for Computational Astrophysics based on using Go for Monte Carlo Radiative Transfer. I find that even though the language was not designed for scientific computing, its speed and simplicity make Go an excellent language for Computational Astrophysics.

1. Introduction

Computational astrophysicists have traditionally used compiled languages like C, C++ or Fortran for computation intensive research areas such as radiative transfer or N-body simulations. Since these languages are compiled, they have excellent run time performance. However, compared to interpreted languages like Python, they are relatively difficult to use due to lack of automatic memory management and run time checks.

The Go programming language is compiled so it has excellent performance. Moreover, it also has automatic memory management (garbage collection) and run time checks. Hence, one is lead to the question: Is Go a practical language for Computational Astrophysics? In this paper I propose an answer to this question based on my experience with using Go for Monte Carlo Radiative Transfer.

2. Go

Go was first released in 2009 and version 1.0 was released in 2012. Hence it is a relatively new language. The language is from Google but it is an open source language. Its creators are Robert Griesemer, Rob Pike and Ken Thompson. (Ken Thompson is a Turing award winner and the creator of UNIX.). The standard book on Go is "The Go Programming Language" (Donovan & Kernighan (2015)). One of the authors of this book is the same Kernighan who wrote the "The C Programming Language" (Kernighan & Ritchie (1988)). Hence, Go has some very distinguished computer scientists associated with it.

Go is a compiled, statically typed language with built-in concurrency. Simplicity was an important design goal for the creators of Go. Due to the scale of Google, they were also concerned with improving performance by reducing compilation times and reducing running times. Also to increase reliability, Go has features which are common in interpreted languages like Python but which do not exist in compiled languages

like C, C++ and Fortran. These are features such as automatic memory management (garbage collection) and run time checks (e.g. to detect array index out of bounds).

As noted in the previous paragraph, Go has several positive features. However, the question remains: Is Go a practical language for computational astrophysics? The only way to find out is to implement computational astrophysics code in Go. Hence, I implemented a Monte Carlo Radiative Transfer program in Go.

3. Monte Carlo Radiative Transfer

We consider Monte Carlo Radiative Transfer (MCRT) with scattering and absorption in spherical layers in an exoplanet atmosphere. An introduction to MCRT with polarization in a spherical geometry is given in Code & Whitney (1995). A parallel beam of photons is incident on the planet's atmosphere. An incoming photon enters the atmosphere with a Stokes vector $(I, Q, U, V) = (1, 0, 0, 0)$. It then travels an optical depth τ till it gets scattered or absorbed. The optical depth τ is given by $\tau = -\log(1 - \xi)$ where ξ is a random number between 0 and 1. The probability of the photon getting scattered is equal to the single scattering albedo. If the photon gets scattered, then the new Stokes vector is the product of a 4×4 matrix and the old Stokes vector. The random direction of scattering is dependent on a probability distribution based on the same matrix. A photon which reaches the surface of the planet gets absorbed or reflected back by the Lambertian surface. Hence, a photon either gets absorbed within the atmosphere, or it gets absorbed on the surface of the planet or it exits at the top of the atmosphere. If the photon exits at the top of the atmosphere then its exiting direction (θ, ϕ) is recorded. (Here θ and ϕ are the usual spherical polar coordinates.) For accurate results, a large number of incoming photons have to be simulated. Hence the program has loops with a large number of iterations.

As seen in the previous paragraph, the MCRT code uses multi-dimensional arrays, random numbers and large number of iterations for multiple loops. These are typical parts of a computational astrophysics programs. Hence, even though the present paper's conclusions are based on this program, they are more generally applicable to other computational astrophysics programs such as N-body programs.

4. Experience with Go

As noted above, simplicity was a design goal for the Go. Unlike most common languages of the last two decades, Go has no inheritance, no templates/generics, and no exceptions. The language was designed to be easy to learn for users with prior experience in the commonly used languages such as C, C++, Java, Python etc. Hence it is easy for computational astrophysicists to learn the language.

Go does automatic memory management (garbage collection) and run time checks such as array bounds checking. This increases the running time (compared to C, C++, and Fortran) but it also increases the reliability of the code. Since Go is a compiled language, the run time performance is very good compared to interpreted languages. Go is very strict about types. Even for numerical types, there is no promotion of `int64` to `float64`. For example if `x` is a `float64` variable and `y` is a `int64` variable then `x=x+y` will not compile. One must use `x=x+float64(y)`. This strict typing prevents various errors. Since each line is a statement, one does not get the kind of odd error

messages which one can get in C and C++ by missing a semicolon. Since all variables are initialized to a default value of the type (e.g. default value for `int64` is `0`), one does not get the run time errors due to uninitialized variables. Another feature of Go is that a function can return multiple values. This makes it possible to have a clear separation between input parameters and output parameters. For example, in below code, `x` and `y` are the input parameters and `a` and `b` are the output parameters:

```
func some_function( x float64, y float64) (a float64, b float64){
    //do some calculations
    return a, b
}
```

Another function can call `some_function()` like below:

```
a, b = some_function( x, y )
```

Multi-dimensional arrays are essential to much of computational astrophysics. Passing multidimensional arrays to functions or subroutines is a common step in most programs. For passing such arrays to functions, C++ requires the sizes for all dimensions (except the first) to be known at compile time. Similarly, Go requires the sizes of all the dimensions to be known at compile time. However, Go has slices which are like dynamic arrays. Hence, one can use a slice of slices (i.e. dynamic array of dynamic arrays) to create a 2 dimensional array of `float64` (similar to `double` type in C and C++ and `double precision` type in Fortran). The code looks like below:

```
func make2DsliceFloat64( XSize int, YSize int) ([][] float64){
    var i int
    //make 2D slice (like array of arrays)
    // Allocate the top-level slice.
    a := make([][]float64, XSize)
    // Allocate the next-level slices.
    for i=0; i <XSize; i++ {
        a[i] = make([]float64, YSize)
    }
    return a
}
```

Here `make2DsliceFloat64()` creates and returns a slice of slices. In the calling program, one would call `make2DsliceFloat64()` as below to make `3 x 3` matrix for the `float64` type:

```
var matrix1 [][]float64
matrix1 = make2DsliceFloat64(3, 3)
```

Since Go has garbage collection, the user does not need to remember to free the memory used by `matrix1`. The above code is similar to the Python numpy statement `a = numpy.zeros((3,3))`. Note that since Go does not have templates/generics, one would need to write another function `make2DsliceInt64()` for the `int64` type. We can pass `matrix1` to a function `trace()` like below

```
trace(matrix1, 3, 3)
```

Also if we pass `matrix1` to a function then the function can modify the elements of `matrix1`. Built-in variables of type `string`, `int64`, `float64` etc. and `struct` variables are passed by value so (just like in C) we must use pointers if we want to modify the variable in the function.

The Go compiler has fast compile times and it gives helpful and readable error messages. The executable produced by the Go compiler is statically linked which means that you can compile Go code on your machine, transfer the executable to another machine and it will run fine as long as both machines have the same operating system. The other machine does not need to have Go installed and it does not need any libraries other than those which are part of the operating system. This is very convenient if you develop your code on your desktop or laptop and then run the executable on a remote machine (e.g. a supercomputer). Go also has built-in features to run code concurrently.

Go has a standard code formatting tool gofmt. After formatting code with gofmt, everyone's code looks the same. This makes it easier to read code written by others. The language has excellent documentation and a large number of built-in libraries. The built-in packages for complex numbers and random numbers are especially useful for scientific computing. The external Gonum project has additional scientific computing packages. However, compared to Python, the number and scope of scientific computing packages is quite limited.

5. Conclusions

Go can be learned quickly by computational astrophysicists since they are already know C, C++ or Fortran. Automatic memory management, run time checks, and strict typing, reduce the code development time. Since it is a compiled language, the run time performance is very good. Due to Go's newness and since it is not targeted at scientific computing, there are a limited number of scientific computing libraries. Hence, Go may not be a feasible choice for projects which depend on specialized libraries. However, for projects which are not dependent on such libraries (such as Monte Carlo Radiative Transfer and N-body simulations), Go is an excellent language for computational astrophysics.

References

Code, A. D., & Whitney, B. A. 1995, Astrophysical Journal, 441, 400

Donovan, A. A., & Kernighan, B. W. 2015, The Go Programming Language (Academic Press)

Kernighan, B. W., & Ritchie, D. M. 1988, The C Programming Language (Prentice Hall), 2nd ed.