Relativistic Ray Tracing in Julia

Ryan McKinnon

December 4, 2015

Abstract

In this work, we implement a relativistic ray tracer in Julia based on the `STARLESS` Python package. Our relativistic ray tracer is parallelized for shared memory architecture using multiple processes and Julia's SharedArray capability. After reviewing the relevant physics, we demonstrate the ability of our ray tracer to render relativistic scenes in which light near a black hole is distorted by its gravity. We also analyze the performance of our Julia code using the built-in profiler and discuss several optimizations that improved the runtime by just under an order of magnitude. Finally, we discuss possibilities for future optimizations and ways in which this work could be extended to distributed memory systems using MPI or a distributed array package.

1 Introduction

Ray tracing is widely used in scientific computing to render three-dimensional datasets in two-dimensional images. It is particularly useful in computational astrophysics, where researchers are often interested in visualizing the results of complex physical phenomena across large dynamic ranges [1, 7, 11, 9]. However, it is also known from the astrophysical theory of general relativity that light rays are bent in a gravitational field. That is, an extreme spacetime can distort what we consider to be normal ray propagation and require the development of physics-aware ray tracers.

While many astrophysical processes are non-relativistic and thus not affected by general relativistic corrections, there are a number of settings in which general relativity plays an important role. Examples include compact objects and black holes, which contain matter sufficiently dense to affect the local spacetime. Early efforts in adopting ray tracing for relativistic settings [4, 3, 5] have given way to mature software used in cutting-edge astrophysics research [8, 2]. Strategies for performing these calculations in parallel are also heavily studied and tailored to the wide variety of supercomputing resources available today, and the field of astrophysics is likely to become even more computational in the future.

In this paper, we port to Julia the `STARLESS`[1] Python package, a lightweight piece of software capable of ray tracing certain relativistic scenes. While not

[1] `http://rantonels.github.io/starless/`

as feature-rich and powerful as some existing ray tracers in the literature, this work demonstrates the suitability of Julia for relativistic ray tracing and the ease with which such work can be parallelized and optimized.

2 Implementation

Relativistic ray tracing is best understood by first considering non-relativistic ray tracing. In astrophysics, the one-dimensional radiative transfer equation is given by

$$I_\nu(s_1) = I_\nu(s_0)\exp(-\tau_\nu(s_0, s_1)) + \int_{s_0}^{s_1} j_\nu(s')\exp(-\tau_\nu(s', s_1))\,ds' \tag{1}$$

where $I_\nu(s)$ denotes the intensity of radiation at position s and in frequency ν, $j_\nu(s)$ is the corresponding emissivity,

$$\tau_\nu(s_0, s_1) = \int_{s_0}^{s_1} \alpha_\nu(s')\,ds' \tag{2}$$

is the optical depth between positions s_0 and s_1, and $\alpha_\nu(s)$ is the opacity at position s in frequency ν (see Equation 1 in [6] and references therein). In ray tracing a three-dimensional volume, the emissivity and opacity are determined by physically-motivated transfer functions applied to the dataset (e.g. one can use density or temperature to estimate the amount of emitted radiation). Since rays are one-dimensional, the radiative transfer equation above governs the evolution of intensity along the propagation direction. Furthermore, when generating RGB output images, we can choose three different transfer functions (corresponding to three different physical wavelengths or frequencies) and independently process data in each color channel.

By discretizing the radiative transfer equation, we can shoot rays over a gridded dataset from back to front – that is, from a background with zero intensity towards the imagined camera. Again paralleling [6], we can define

$$\theta_k = \exp(-\tau_\nu(s_{k-1}, s_k)) \tag{3}$$

and

$$b_k = \int_{s_{k-1}}^{s_k} j_\nu(s')\exp(-\tau_\nu(s', s_k))\,ds', \tag{4}$$

obtaining the recursive relation

$$I_\nu(s_k) = I_\nu(s_{k-1})\theta_k + b_k. \tag{5}$$

If s_0 and s_n denote the first and last grid positions, respectively, then

$$I_\nu(s_n) = \sum_{k=0}^{n} b_k \prod_{j=k+1}^{n} \theta_j. \tag{6}$$

This discretized equation can also be solved front to back, a method which turns out to be more computationally efficient. These equations form the basis of non-relativistic ray tracing in astrophysics, with RGB output obtained through the selection of three different frequencies. On a multicore computer, different regions or pixels of the output image can be assigned to individual processes, with ray tracing done in parallel. This introduces some initial communication overhead but ultimately yields a highly parallelizable method.

General relativity introduces some corrections to the paths that light rays follow, but for certain cases we can make a number of simplifications. The well-known Schwarzschild metric describes spacetime near an uncharged, nonrotating black hole and is given, in units where the speed of light and Schwarzschild radius have unit magnitude, by

$$ds^2 = \left(1 - \frac{1}{r}\right) dt^2 - \left(1 - \frac{1}{r}\right)^{-1} dr^2 - r^2(d\theta^2 + \sin^2\theta d\phi^2), \tag{7}$$

where now s denotes proper time, t denotes time, and (r, θ, ϕ) are the usual spherical coordinates (see [10] or other reference texts on general relativity). Defining $u = 1/r$ and focusing on the $\theta = \pi/2$ plane, for massless particles like photons one can ultimately derive the relation

$$\frac{d^2u}{d\phi^2} = -u + \frac{3u^3}{2}. \tag{8}$$

This equation can be in turn recast as a Newton-like problem, in which trajectories are integrated under the influence of a specified force field. It is this final approach that the Python `STARLESS` package follows and which we incorporate into our Julia code.

Thus, while a fully general relativistic ray tracer that integrates photon paths from first principles may be computationally very expensive, for certain geometries that are widely studied in astrophysics we can simplify the calculations considerably to make them tractable. In the case of the Schwarzschild black hole, we are able to perform calculations similar to those of Newtonian physics but with additional corrections. The underlying concepts of ray tracing – tracking light rays from multiple levels of depth and calculating opacities – remain the same.

In Julia, we ray trace the output for multiple pixels in parallel and use the SharedArray syntax to easily index a common array from multiple processes running on shared memory. The core of the parallelization is done using the code segment

```
colour_shared = SharedArray(Float64, (numPixels, 3))
@sync begin
    for (i, wpid) in enumerate(workers())
        @async begin
            remotecall_wait(wpid, raytrace_func, i,
                schedules[i], colour_shared)
```

```
        end
    end
end
```

where `colour_shared` holds the RGB image output and is accessible from every process and `raytrace_func` is the function that performs the actual ray tracing and takes as one of its arguments `schedules[i]`, an array of chunks of pixels assigned to the i-th worker. The `CHUNKSIZE` parameter controls how many pixels each worker updates in parallel during the integration process. Too many pixels can result in large memory allocations and repeated garbage collection, while too few pixels fails to take full advantage of the benefits of array operations.

We note that the Julia parallelization syntax is considerably simpler than its Python counterpart. For example, the `multiprocessing` module in Python requires the use of a `multiprocessing.Array` data type when launching worker processes, but this custom array type is not directly compatible with the `NumPy` package for array operations. This results in some array conversions that must be performed manually. On the other hand, Julia's `SharedArray` contains all parallel-friendly features under-the-hood and can be directly used in Julia functions in place of a normal array.

While the syntax may not be as simple if one were to move this Julia package to a distributed memory architecture, the ease of shared memory parallel programming in Julia certainly encourages exploration.

We also use the `ConfParser` package for reading configuration files at runtime, allowing us to separate the core ray tracing features from parameters affecting the visual output. The Julia code needed to run this is available at `http://www.github.com/RyanMcK/jlstarless`.

3 Results and Performance

To demonstrate the capabilities of a relativistic ray tracer, in Figure 1 we show a sample image consisting of a black hole, surrounding accretion disk, and background stars. As explained in the previous section, this output was generated in parallel by assigning each process a portion of the image through which to ray trace. Without a relativistic ray tracer, the ring-shaped features surrounding the black hole and distorting starlight would not render in a physically-accurate way.

After porting `STARLESS` to Julia to produce version 1 of our code, we profiled the main `raytrace_func` function using Julia's `Profile` module. We then engaged in a cycle of tweaking and profiling to make incremental improvements to the Julia code. Because ray tracing involves a significant amount of geometric calculations (e.g. vector additions, cross products, norms, etc.), profiling revealed that norm computations – in particular, computing the row-wise norm of an $N \times 3$ matrix – were the most time-consuming portion of our code. We wrote version 2 of our code using the `sumabs2` function instead of a combination of `mapslices` and `norm`, speeding up the code by roughly a factor of 5.8.

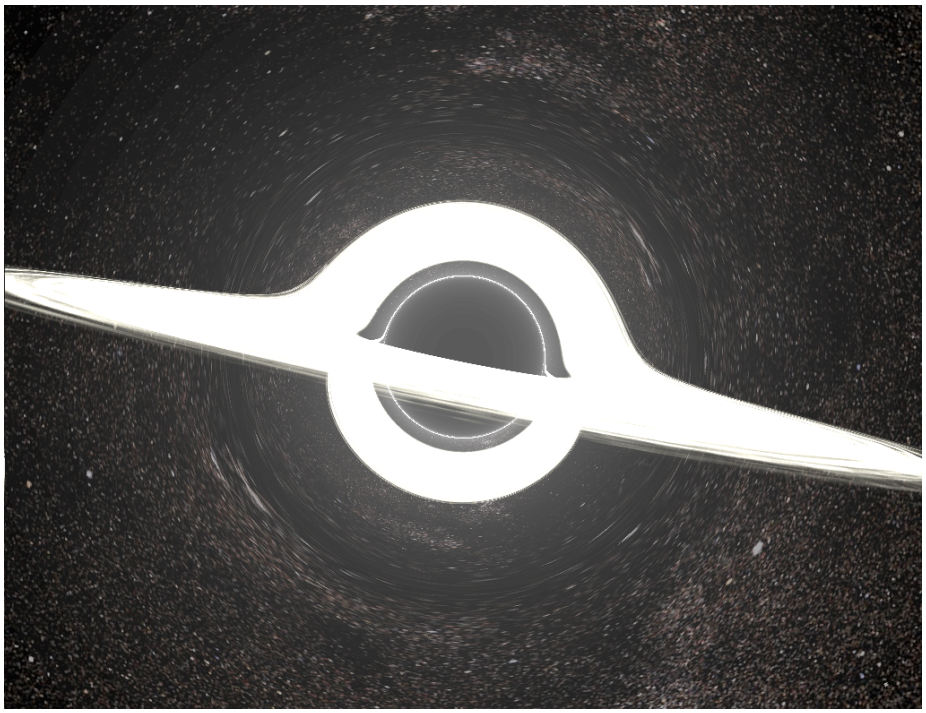

Figure 1: The default scene rendered by the Julia relativistic ray tracer, including a central black hole and associated accretion disk. The black hole appears to bend light from stars in a nearby ring. The contrast of this image has been slightly enhanced to improve the appearance of faint stars in the background.

Another expensive part of the ray tracer was the Runge-Kutta integrator, which originally had been wrapped in a function. This resulted in the creation of several temporary arrays, which in turn led to excessive execution time. The integration portion of our code was sped up by moving the Runge-Kutta calculations inline and removing all function calls. We were then able to eliminate many temporary arrays by performing in-place operations and avoiding implicit copies in long expressions. This resulted in version 3 of our code, with a speedup factor of 1.2 compared to version 2.

The final version of our code improved runtime by reducing the `CHUNKSIZE` parameter, which we noted above controls the number of pixels simultaneously updated by each process and so determines the size of many arrays created during Runge-Kutta integration. Using the `--track-allocation=user` command-line option in Julia, we were able to track memory usage line by line and spot the worst offending lines. By reducing `CHUNKSIZE` by a factor of 64, we were able to rein in large memory allocations and improve runtime by a further factor of 1.3 over version 3. While we settled on a final `CHUNKSIZE` value by trial and error, in the future it may be worthwhile to investigate schemes that adaptively select the appropriate `CHUNKSIZE` at runtime given the characteristics and memory speeds of the particular machine used for ray tracing.

In total, this reduced the runtime of our original Julia code by a factor of around 8.6, just under an order of magnitude. To visualize these performance results, in Figure 2 we plot the execution time for each version of this Julia ray tracer as a function of number of cores. The data include time spent allocating pixel chunks to each process and the actual ray tracing but exclude the small amount of time used to load initial configuration files. These runs were conducted on Harvard University's Odyssey supercomputer, which has nodes with 64 cores sharing memory. All versions of the code display a scaling fairly close to the ideal $1/N$ scaling, suggesting that these methods would continue to work on supercomputers with even larger shared memory nodes.

Despite the improvements that we have made, there are a number of avenues to pursue for even more performance gains. While the code we have ported to Julia relies on shared memory machines having many cores per node, not all supercomputers fit this model. However, there are other Julia frameworks that could take advantage of a distributed memory architecture and allow the ray tracer to use hundreds or thousands of cores. These include the `DistributedArrays`[2] package for indexing arrays on distributed memory as easily as with Julia's SharedArray syntax and its alternative, Message Passing Interface wrappers like `MPI.jl`[3] for more traditional, communication-intensive parallel programming. It is also possible that there are even faster ways of computing row-wise norms than the `sumabs2` function, perhaps by using some external package designed for high-performance computing. Furthermore, there are likely optimizations that could be made to the numerical integrator used: currently, a fixed number of integration steps are used for each pixel, while there

[2] `https://github.com/JuliaParallel/DistributedArrays.jl`
[3] `https://github.com/JuliaParallel/MPI.jl`

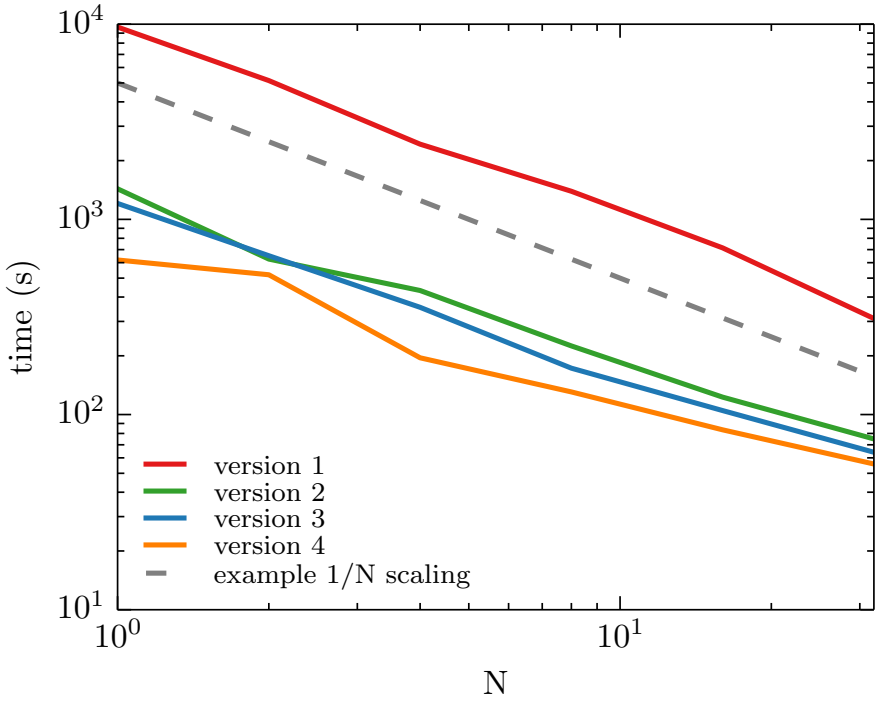

Figure 2: Timing comparison for the four versions of Julia code written, shown as colored lines. The gray dashed line shows the slope of an ideal $1/N$ scaling. The final code, version 4, yields a scaling only slightly worse than the ideal case.

may be some cases in which a pixel's RGB output value converges quickly and only a few integration steps are actually needed.

4 Conclusion

We have demonstrated Julia to be a capable language for astrophysics-inspired ray tracing of relativistic scenes, based on the `STARLESS` Python package. Shared memory programming in Julia, using the SharedArray syntax, is even easier than in Python, and we were able to use Julia's native profiling features to improve code performance by almost an order of magnitude. We display close to ideal scaling out to 32 cores on shared memory and expect these results to hold for even larger machines, provided the number of pixels being rendered is sufficiently high.

This project has also revealed the ease with which Python codes can be ported to Julia. In particular, Julia's built-in support for arrays and the ability to use SharedArrays seamlessly in array functions and operations makes the resulting code easier to write and parallelize. As more third-party libraries are developed in Julia, particularly ones for scientific computing, we expect the appeal of Julia to grow.

References

[1] T. Abel and B. D. Wandelt. Adaptive ray tracing for radiative transfer around point sources. *MNRAS*, 330:L53–L56, March 2002.

[2] C.-k. Chan, D. Psaltis, and F. Özel. GRay: A Massively Parallel GPU-based Code for Ray Tracing in Relativistic Spacetimes. *ApJ*, 777:13, November 2013.

[3] C. T. Cunningham. The effects of redshifts and focusing on the spectrum of an accretion disk around a Kerr black hole. *ApJ*, 202:788–802, December 1975.

[4] C. T. Cunningham and J. M. Bardeen. The Optical Appearance of a Star Orbiting an Extreme Kerr Black Hole. *ApJ*, 183:237–264, July 1973.

[5] J.-P. Luminet. Image of a spherical black hole with thin accretion disk. *A&A*, 75:228–235, May 1979.

[6] T. Peters. The physics of volume rendering. *European Journal of Physics*, 35(6):065028, November 2014.

[7] H. Trac and R. Cen. Radiative Transfer Simulations of Cosmic Reionization. I. Methodology and Initial Results. *ApJ*, 671:1–13, December 2007.

[8] F. H. Vincent, T. Paumard, E. Gourgoulhon, and G. Perrin. GYOTO: a new general relativistic ray-tracing code. *Classical and Quantum Gravity*, 28(22):225011, November 2011.

[9] M. Vogelsberger, S. Genel, D. Sijacki, P. Torrey, V. Springel, and L. Hernquist. A model for cosmological simulations of galaxy formation physics. *MNRAS*, 436:3031–3067, December 2013.

[10] R. M. Wald. *General relativity*. 1984.

[11] J. H. Wise and T. Abel. ENZO+MORAY: radiation hydrodynamics adaptive mesh refinement simulations with adaptive ray tracing. *MNRAS*, 414:3458–3491, July 2011.