Vision, Modeling, and Visualization (2022)
J. Bender, M. Botsch, and D. Keim (Eds.)

Astray: A Performance-Portable Geodesic Ray Tracer

A. C. Demiralp, M. Krüger, C. Chao, T. W. Kuhlen, and T. Gerrits

RWTH Aachen University, Germany

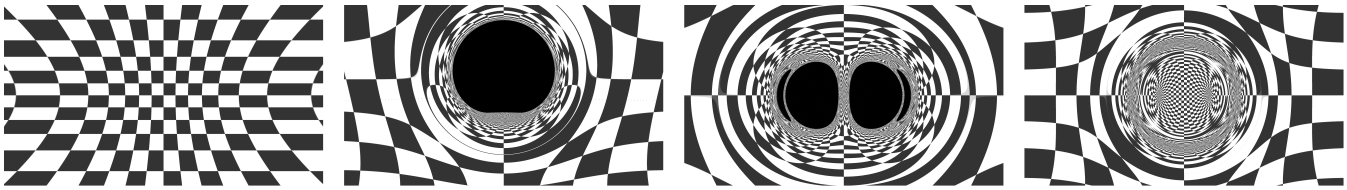

Figure 1: *Visualizations of (a) Minkowski, (b) Kerr, (c) Kastor-Traschen, (d) Morris-Thorne spacetimes created with Astray.*

Abstract

Geodesic ray tracing is the numerical method to compute the motion of matter and radiation in spacetime. It enables visualization of the geometry of spacetime and is an important tool to study the gravitational fields in the presence of astrophysical phenomena such as black holes. Although the method is largely established, solving the geodesic equation remains a computationally demanding task. In this work, we present Astray; a high-performance geodesic ray tracing library capable of running on a single or a cluster of computers equipped with compute or graphics processing units. The library is able to visualize any spacetime given its metric tensor and contains optimized implementations of a wide range of spacetimes, including commonly studied ones such as Schwarzschild and Kerr. The performance of the library is evaluated on standard consumer hardware as well as a compute cluster through strong and weak scaling benchmarks. The results indicate that the system is capable of reaching interactive frame rates with increasing use of high-performance computing resources. We further introduce a user interface capable of remote rendering on a cluster for interactive visualization of spacetimes.

CCS Concepts

• ***Applied computing*** → *Physics;* • ***Computing methodologies*** → *Ray tracing; Parallel algorithms;* • ***Human-centered computing*** → *Scientific visualization;*

1. Introduction

While classical ray tracing is concerned with creating photo-realistic renderings of geometry and volumes by computing the motion of light in space, geodesic ray tracing is the numerical method for computing the motion of matter and radiation in spacetime, according to the laws of general relativity. It is a valuable tool for studying the geometry of spacetime in the presence of astrophysical phenomena such as black holes and gravitational waves. It is also useful for validation of observational data, such as the recent images of the M87* created by the Event Horizon Telescope [AAA*21], against theoretical models. Various geodesic ray tracers have been published over the years. Yet these often focus on specific metrics, utilize parallelization to a limited extent, or both.

In this paper, we propose several contributions offering a general, parallel, and extendable architecture for geodesic ray tracing to be used by domain scientists such as theoretical physicists. We present two tools; Astray, a geodesic ray tracing library, and Astrid, an exemplary end-user application based on Astray.

Astray is developed according to the following system design principles:

- General, such that it supports geodesic ray tracing in any spacetime with a known metric tensor.
- Efficient, such that it utilizes the optimal parallelization paradigms wherever applicable.
- Scalable, such that it is capable of running on hardware ranging from standard workstations to large compute clusters.
- Extensible, such that it is capable of supporting new metrics with ease and thus allowing fast hypothesis evaluation.

Astrid is an application suite that enables interactive visualiza-

DOI: 10.2312/vmv.20221208

tion of spacetimes. It is built with Astray and consists of two components:

- Astrid server: A geodesic ray tracing server that accepts integration and observation parameters from clients, renders an image according to them, and sends an image back.
- Astrid client: A user interface to interact with and present the images received from a server.

The client and the server are packaged into a single application that can run in either of the two modes. The separation of rendering from the user interface enables the ray tracer to run on a remote machine or cluster, bridging end-user hardware to high-performance computing resources.

Emphasizing extensibility and ease-of-use, we aim to lower the barrier of entry for domain scientists to incorporate geodesic ray tracing into their workflow. Experts can focus on exploring the parameter space of existing metrics and implement their own metrics with ease. Due to the parallelization and distribution of tasks being abstracted by the library, the domain scientists are not required to have background knowledge on these topics. The focus on performance and scalability presents a first step toward interactive analysis of spacetime geometry: We aim to present not only a useful addition to the toolset of domain experts but also a way for an easier creation of explorative software such as immersive visualization environments or tools aimed toward teaching general relativity.

In the following sections, we explain how these goals are achieved, evaluate our framework on various hardware and with various metrics, measuring the performance with regard to the capability of providing interactive frame rates.

2. Background

This section is a brief summary of general relativity, focusing on the concepts necessary for geodesic ray tracing. For an in-depth review of the subject, we refer the reader to [Har03].

2.1. Spacetime

Spacetime is a 4-dimensional manifold consisting of 1 temporal and 3 spatial components. It is a model of the universe. A point in spacetime is called an event, and can be described by a coordinate consisting of a 4-vector: $[t\ x_1\ x_2\ x_3]^\mathrm{T}$ where t is the time coordinate and x_i are the components of a point located in a 3-dimensional coordinate system such as the Cartesian, cylindrical or spherical coordinates.

2.2. Line Element

In 3-dimensional Cartesian coordinates, the Pythagorean theorem describes the squared distance between two points as:

$$ds^2 = dx^2 + dy^2 + dz^2 \tag{1}$$

where ds is called the line element and $[dx\ dy\ dz]^\mathrm{T}$ is the vector of differences between the two points, in terms of each coordinate axis. The line element concept is generalized in Riemannian geometry, removing the requirement of the coordinate axes to be orthonormal:

$$ds^2 = \sum_{i,j}^{n} g_{ij}\, dx^i\, dx^j \tag{2}$$

where n is the number of dimensions and g_{ij} is a second rank, symmetric, positive definite tensor called the *metric tensor*. Notice that Equation 1 is reconstructed when the metric tensor is set to the identity I_3.

2.3. Metric

In general relativity, the metric tensor appears in the Einstein field equations:

$$R_{ij} - \frac{1}{2} R g_{ij} + \Lambda g_{ij} = \frac{8\pi G}{c^4} T_{ij} \tag{3}$$

where R_{ij} is the Ricci curvature tensor, R is the Ricci scalar, T_{ij} is the energy-momentum tensor, and Λ, π, G, c are scalar constants. Every solution to the Einstein field equations is characterized by a metric tensor g_{ij}, enabling measurement of distances in the spacetime described by it. Since spacetime is defined as a pseudo-Riemannian, specifically a Lorentzian manifold, the metric tensor is not required to be positive definite. This implies that ds^2 in Equation 2 may be positive, zero, or negative, corresponding to time-like, light-like, and space-like intervals respectively.

An important set of mathematical objects derived from the metric tensor are the Christoffel symbols of the second kind Γ^i_{jk}. These are computed symbolically from the derivative of the metric, and describe the variation of the basis vectors due to a change in coordinates. In a more formal sense, the Christoffel symbols are the coefficients of the Levi-Civita connection, the unique affine connection on the tangent bundle of the manifold described by the metric. They establish the notions of differentiation and integration on the manifold and hence are essential to the computation of geodesics.

2.4. Geodesic Equation

The motion of free falling particles in spacetime is described by the geodesic equation:

$$\sum_{i}^{n} \left(\frac{d^2 x_i}{d\lambda^2} + \sum_{j,k}^{n} \Gamma^i_{jk} \frac{dx_j}{d\lambda} \frac{dx_k}{d\lambda} \right) = 0 \tag{4}$$

where λ is an affine parameter, specifically proper time τ in the case of time-like geodesics. There are three types of geodesics corresponding to the time-like, light-like, and space-like intervals, classified based on the sign of the (initial) direction. In the context of ray tracing, time-like and light-like geodesics are of interest as they describe the motion of objects with mass and photons respectively. The geodesic equation is numerically solved by decomposing the second-order differential equation into a pair of coupled first-order equations, which are then iterated using a method such as fourth-order Runge-Kutta.

In geodesic ray tracing, Equation 4 is iteratively solved for each ray on the image plane of the observer with $n = 4$. The number of rays depends on the size of the image plane, ranging from several

thousand to millions of pixels. The number of iterations varies from a few hundred for quick tests to several millions for high-quality images. It is therefore essential for a high-performance ray tracer to solve this equation as efficiently as possible. Furthermore, as the equation depends on the Christoffel symbols of an arbitrary metric, a geodesic ray tracer striving for generality should allow the user to set the Christoffel symbols for distinct metrics in order to support a wide range of spacetimes.

3. Related Works

Several geodesic ray tracers were proposed in the last decades. An early implementation was presented in [Wei00], generalizing concepts from standard ray tracing to relativistic settings. The performance of this ray tracer was improved through the utilization of graphics processing units (GPU) in [WSE04]. Various applications based on the ray tracer were presented in [WBE*06], yet real-time geodesic ray tracing was referred to as challenging if not impossible using hardware available at the time. Questioning whether this statement still holds, we revisit the problem with a distributed approach on modern hardware.

Another geodesic ray tracing library supporting a variety of metrics called Motion4D was introduced in [MG09b]. Several performance improvements to this library were described in [KMA*12] and [MBW15], along with a user interface called GeoViS presented in [Mü14]. Motion4D and the catalogue of spacetimes [MG09a] accompanying the library contain the numerical descriptions of most known spacetimes to date, and have enabled us to implement a wide range of metrics, such as the ones seen in Figure 1. Although Motion4D provides GPU support, and GeoViS uses the Message Passing Interface (MPI) [MPI21] for threading, their distributed computing capabilities are limited.

The more recent approaches presented in [CPO13], [PYYY16], [CMOP18] and [VE21] are also capable of running on GPUs, but they provide no distributed computing support and are specialized for the Kerr spacetime. The ray tracers presented in [VPGP11], [VCAVL*22], [PRCB22], and [Whi22] are more general, yet are limited to compute processing units (CPU) on a single computer. The approach presented in [PMNJ18] offers no parallelization at all.

Based on the literature review, we conclude that while various geodesic ray tracers exist that either focus on performance or generality, there are few capable of both. Furthermore, the majority of the geodesic ray tracers to date do not provide distributed computing support, implying limited scalability. In contrast to the existing solutions, Astray simultaneously targets performance, scalability, and generality. It is a flexible system capable of scaling from a single computer to a cluster of compute or graphics processors, using identical code. It is further capable of working with any spacetime with a known metric tensor and includes optimized implementations of common spacetimes. We demonstrate the features of the system in Section 4, and assess its performance on a mobile workstation as well as a compute cluster in Section 5.

```cpp
#include <astray/api.hpp>
int main(int argc, char** argv)
{
 using scalar_type = double;
 using metric_type = ast::metrics::schwarzschild<scalar_type>;
 using geodesic_type =
  ast::geodesic<scalar_type,ast::runge_kutta_4<scalar_type>>;

 ast::ray_tracer<metric_type, geodesic_type> ray_tracer;
 ray_tracer.set_image_size      ({1920,1080});
 ray_tracer.set_lambda_step_size(0.01);
 ray_tracer.set_iterations      (1000);

 auto& transform = ray_tracer.get_observer().get_transform();
 transform.translation = {0.0, 0.0, -10.0};
 transform.look_at     ({0.0, 0.0,   0.0});

 const auto image = ray_tracer.render_frame();
 image.save("example.png")
}
```

Listing 1: *A simple example for rendering the Schwarzschild metric using Astray, which describes a static uncharged black hole.*

4. Implementation

The ray tracer class, seen in the example presented in Listing 1, is the outermost abstraction exposed to the users. It is parametrized by the type of the metric tensor and the geodesic, the latter being additionally parametrized by the method of integration. It allows adjustment of integration parameters, including the number of iterations and λ step size, and furthermore provides mutable access to the observer. It is responsible for rendering the frame, which can then be saved to a file as seen in the example, or streamed to a client.

As summarized in Section 2, geodesic ray tracing involves solving the geodesic equation in an iterative manner. The equation in turn depends on the Christoffel symbols of the metric it is being solved for. Following the theory closely, two central abstractions in Astray are the geodesic and the metric. The geodesic class encapsulates the functionality to iterate a single particle in a metric. The kernel evaluating the geodesic equation is seen in Listing 2.

Although a single iteration may be fast, the geodesic kernel is called multiple times for every iteration of each ray. For example, the kernel is evaluated 4 times for a single iteration of a fourth-order Runge-Kutta method on a single ray. It, therefore, becomes computationally expensive quickly and has to be implemented efficiently as it is potentially the main bottleneck of geodesic ray tracing.

The library is equipped with an ordinary differential equation

```cpp
[&metric] __device__ (const scalar_type t, const value_type& y)
{
  value_type dydt;
  dydt.head(4) = y.tail(4);
  auto christoffel_symbols =
    metric.christoffel_symbols(y.head(4));
  for (auto i = 0; i < 4; ++i)
    for (auto j = 0; j < 4; ++j)
      for (auto k = 0; k < 4; ++k)
        dydt .tail(4)[k] -= christoffel_symbols(i, j, k)
          * y.tail(4)[i]
          * y.tail(4)[j];
  return dydt;
}
```

Listing 2: *The kernel solving the geodesic equation. The value_type is an 8 component vector, describing the position and the direction of the ray.*

```
template <
  coordinate_system_type system,
  typename               scalar_type,
  typename               vector_type,
  typename               christoffel_symbols_type>
class metric
{
  virtual christoffel_symbols_type christoffel_symbols
    (const vector_type& position) = 0;

  virtual termination_reason check_termination
    (const vector_type& position, const vector_type& direction)
  {
    return termination_reason::none;
  }
  virtual scalar_type coordinate_system_parameter()
  {
    return scalar_type(0);
  }
};
```

Listing 3: *The metric interface to be implemented for each spacetime. Note that the last two functions provide default implementations and are hence optional.*

solver that is utilized to iterate the geodesic equation. The solver implements both fixed and adaptive step size methods including Runge-Kutta 4, Fehlberg 5, and Dormand Prince 5. Several standard error controllers for adaptive step size methods are also implemented, specifically the integral, proportional integral (PI), and proportional integral derivative (PID) controllers. The solver is further capable of decomposing n^{th}-order initial value problems into n coupled initial value problems. Although many differential equation solvers such as Boost.Odeint [AM11] and LLNL Sundials [HBG*05] have been presented in the past, to our knowledge, our solver is the only one capable of being instantiated in device code, which provides additional flexibility.

The library currently supports 18 spacetimes. Multiple black hole metrics, including Schwarzschild (static uncharged), Kerr (rotating uncharged), Reissner-Nordström (static charged) are implemented. Regarding gravitational wave metrics, Weber-Wheeler-Bonnor pulse, as well as Bessel waves, are supported. Several more exotic spacetimes, such as the Alcubierre warp drive and the Morris-Thorne wormhole are also available. For a complete list, we refer the reader to the source code.

Additionally, if the requested spacetime is not already available in Astray, the users can define it themselves by inheriting the metric class seen in Listing 3. The users are required to provide the Christoffel symbols of the metric as a function of position, optionally along with a termination condition which can be used to check for constraint violations, numeric errors, and spacetime breakdowns. Afterward, the new metric can be provided as a parameter to the ray tracer class and used for geodesic ray tracing. No further knowledge of the inner workings of Astray is required, which makes it easy to use for domain scientists.

The observer abstraction is the analog of a camera in standard ray tracing. It consists of a transform and a projection and is responsible for generating rays from this information. Ray generation is performed separately from tracing, which is common practice in standard ray tracers today. This allows the implementation of alternative projections with ease, as well as the use of non-rectangular image planes and even arbitrary functions for generating rays.

Geodesic ray tracing requires a capable linear algebra solver, as most equations involve vectors and tensors. To this extent we use Eigen [GJ*10] as it is highly optimized, implements tensors, and provides GPU support. Throughout the development of Astray, we have made several minor contributions to Eigen; correcting the construction of fixed-size tensors and enabling more matrix operations on the GPU.

Shared memory parallelization is built on Thrust [BH12], which provides a common interface for OpenMP, Intel TBB, and Nvidia Cuda. This allows the library to target CPUs and GPUs with identical code. However, it also requires splitting the library into host and device code, an abstraction originating in Cuda. The host code is responsible for driving the application from the CPU, whereas the device code is used for the kernels that are run in parallel on either the CPU or the GPU. The majority of Astray is device code, notable exceptions being the observer and the ray tracer which drive ray generation and tracing respectively.

Distributed memory capabilities are achieved with MPI [MPI21]. A data-parallel approach is used, in which the framebuffer is partitioned into equal-sized tiles and distributed to the processes. The size of the tiles is computed from the prime factors of the process count, which are assigned to the two dimensions of the image in a manner that minimizes the aspect ratio of the tiles. As the processors on a device are saturated by Thrust, applications are intended to run with one process per node. Currently, rendering a frame is implemented as a blocking operation as the finished tiles are collected to the root process using *MPI_Gather*, although there are no architectural limitations that would prevent non-blocking communication for asynchronous computation of multiple frames.

5. Experiments

This section describes the experiments conducted on the system to assess its runtime performance and scaling behavior. Emphasizing performance-portability, the experiments target standard consumer hardware as well as compute clusters. An important question to be answered by the measurements is whether and to what extent real-time geodesic ray tracing is possible on modern hardware.

The following parameters are constant for all benchmarks: The integration scheme is set to Runge-Kutta 4. The number of iterations is set to 1000. The λ is set to 0.0 with a step size of 0.01. The bounds are not set. The observer is located at $[0\ 5\ 0\ 0]^{\mathrm{T}}$ and is oriented toward the origin. The projection is set to perspective, with a field-of-view of 120 degrees, a focal length of 1, and a clipping range of [0.01, 100). For the Schwarzschild, Kerr, and Kastor-Traschen metrics, all masses are set to 1. For the Kerr metric, the angular momentum is set to 1. For the Kastor-Traschen metric, the two singularities are located at $[0\ 0\ 0\ -1]^{\mathrm{T}}$ and $[0\ 0\ 0\ 1]^{\mathrm{T}}$ respectively.

Measurements are taken for each of the four metrics: Minkowski, Schwarzschild, Kerr, Kastor-Traschen, and for each backend: serial, Cuda, OpenMP and TBB. Each measurement is repeated 10 times and the results are averaged.

5.1. Workstation

We measure the performance behavior of the system on standard consumer hardware. The output image size is varied as 240×135,

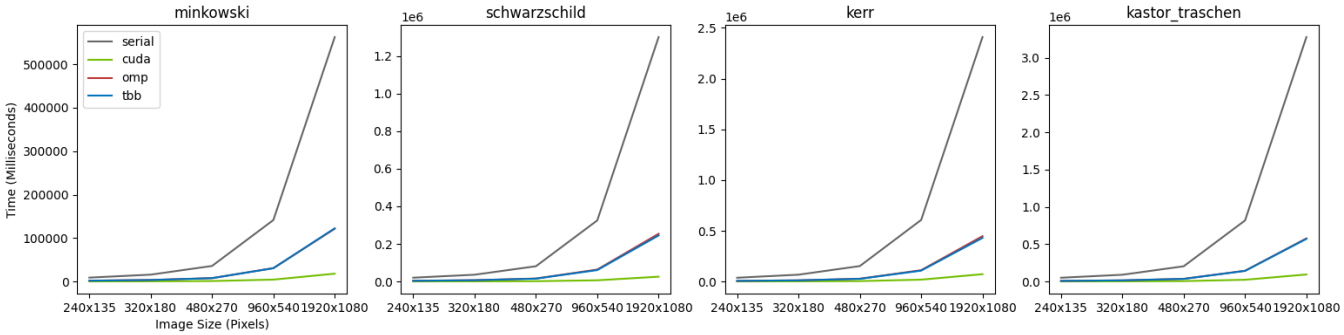

Figure 2: *Workstation benchmarks. Horizontal axes correspond to image size, vertical axes correspond to time in milliseconds needed to render a single frame. Colors correspond to the device system: Black, green, red and blue represent serial, CUDA, OpenMP and TBB respectively. Each plot corresponds to a metric. From left to right: Minkowski, Schwarzschild, Kerr, Kastor-Traschen.*

320×180, 480×270, 960×540, 1920×1080 while the processing power and all other parameters are kept constant. This allows to observe the time it takes to render a 1080p image with decent detail and to see if a configuration supporting interactive frame rates is possible on a single device.

5.2. Cluster

We measure the strong and weak scaling of the system on a compute cluster. For strong scaling, the number of processes is varied as 1, 2, 4, 8, 16 and the image size is fixed to 1024^2. Thus, it provides insight into the system's response to an increase in processing power as it is performing a constant amount of work and to find a configuration with a duration suitable for interactive visualization. For weak scaling, the number of processes is varied as 1, 2, 4, 8 and 16 simultaneously with the image size, which is varied as 512^2, 724^2, 1024^2, 1448^2, 2048^2. Showing to what extent the system provides a constant response to an equivalent increase in processing power and work.

6. Results & Discussion

The workstation benchmarks are run on an MSI GE75-8SF laptop equipped with an Intel Core i7-8750H processor, an Nvidia GeForce RTX 2080 graphics processor, and 64GB of DDR4 memory. The results are presented in Figure 2.

The following table displays the minimum and the maximum speedup relative to the serial approach across all workstation measurements:

	Speedup (Workstation)
OpenMP	4.63 - 5.66
TBB	4.67 - 5.73
Cuda	33.08 - 50.77

The CPU backends show almost identical performance, with TBB slightly outperforming (<0.16%) OpenMP in 19 out of 20 cases, the only exception being the Minkowski metric at 1080p. Cuda provides the best performance, yet still requires several minutes to generate 1080p images with 1000 iterations per pixel, implying that real-time geodesic ray tracing with sufficient detail is out of reach, at least on a single device.

Interestingly, the Minkowski metric appears to benefit less from parallelization, being limited to a speedup of 4.67 on the CPU and 37.49 on the GPU. This is counter-intuitive at first since its geodesics are predictable straight lines. Yet the system does not apply such predictions for generality and iterates regularly with all Christoffel symbols set to zero. Essentially, the Minkowski metric measures the raw integration time, omitting any overhead introduced by the computation of the Christoffel symbols.

The strong and weak scaling benchmarks were run on the Aix-CAVE partition of the RWTH Aachen compute cluster. Each node is equipped with 2 Intel Xeon Silver 4114 processors, 2 Nvidia Quadro P6000 graphics processors, and 192GB of DDR4 memory. The benchmarks are conducted using 1 to 16 processors in an interactive session. The results are presented in Figure 3. The serial measurements are omitted from the plots as they interfere with the scale.

The following table displays the minimum and the maximum speedup relative to the serial approach across all strong scaling measurements on the cluster:

	Speedup (Cluster)
OpenMP	23.97 - 31.19
TBB	24.79 - 31.21
Cuda	27.3 - 36.18

The strong scaling measurements display similar behavior for all three backends. TBB performs consistently better than OpenMP in the cluster, with the difference reaching 8.31% in the single process case of the Kerr metric. Cuda still provides the best performance, yet the distinction between the CPU and the GPU backends is less emphasized in comparison to the workstation benchmarks. This might be attributed to the relative capabilities of the hardware installed on the cluster. Aside from scaling, the results indicate that

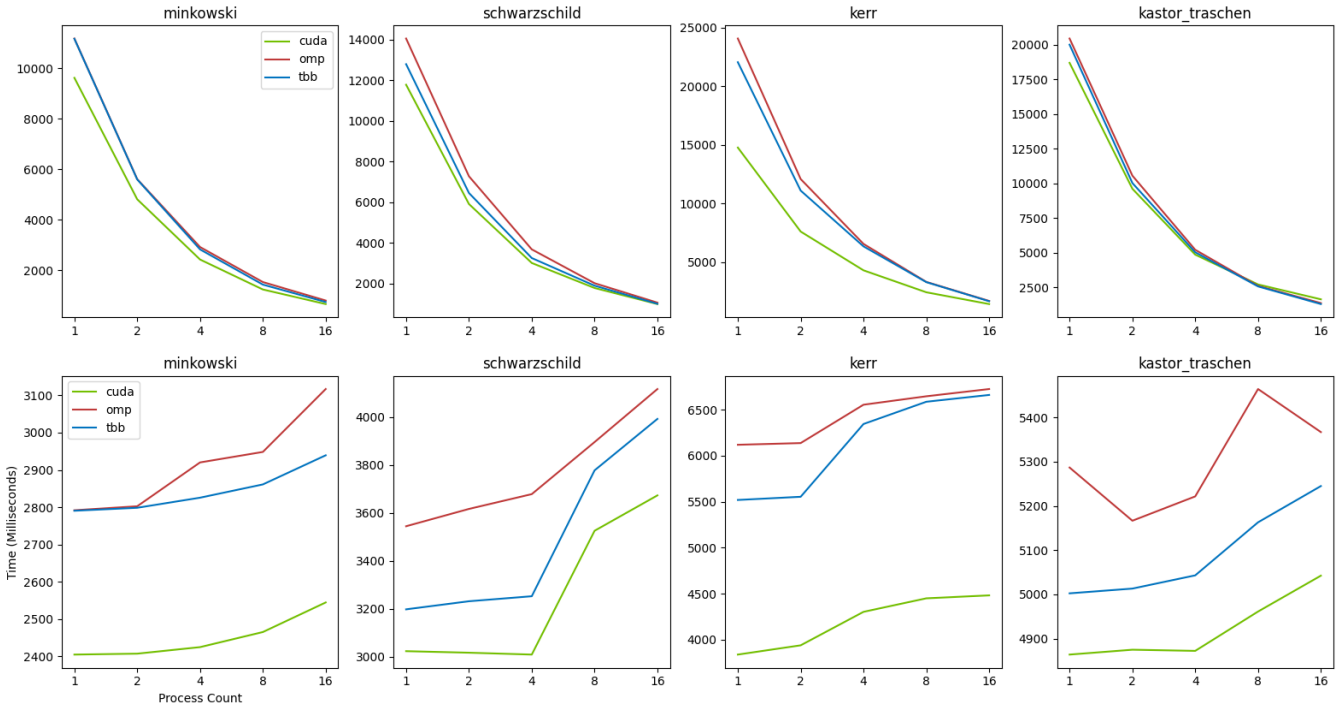

Figure 3: *Strong (top) and weak (bottom) scaling benchmarks. Horizontal axes correspond to processor count (and image size for weak scaling), vertical axes correspond to time in milliseconds needed to render a single frame. Colors correspond to the device system: Green, red, blue represent CUDA, OpenMP, TBB respectively. Each plot corresponds to a metric. From left to right: Minkowski, Schwarzschild, Kerr, Kastor-Traschen.*

near-interactive visualization is possible, as sub-second computation times are reached with 16 processes.

The weak scaling measurements show variations reaching up to 500 milliseconds. This is expected to an extent, as the benchmarks were run interactively rather than in batch mode, implying that the processors were not reserved for the application. The duration of ray tracing also appears to increase with additional processes, despite work per process being kept constant. The amount of increase is similar for the CPU and the GPU backends, indicating a common serial operation such as gathering the output image to the root process as a potential reason. Further investigation remains as future work.

7. User Interface

In addition to the library, an end-user tool was developed with the intent of making geodesic ray tracing accessible to users without programming experience: *Astrid* is a ray tracing server and user interface built upon Astray, enabling interactive visualization of a wide range of spacetimes. The interface, seen in Figure 4, allows the users to adjust the metric, the observer, and the integration parameters, and immediately render a result. The position and the orientation of the observer may also be updated using standard first-person controls with movement keys and mouse rotation. The users can further toggle looped rendering, in which the parameter changes are transmitted continuously to the server, enabling interactive exploration of metrics. Given sufficient hardware on the server-side, the users can then browse the metric in real-time.

Astrid is capable of rendering images on the local device the user interface is running on, as well as a remote device or cluster. This is achieved using an approach similar to ParaView [AGL05], by abstracting the rendering functionality into a server application that is distinct from the user interface. The user interface is a thin client responsible for forwarding the parameters from the user to the server via requests. The server listens for requests and replies to them with a rendered image to be displayed by the user interface.

The client and the server communicate via the ZeroMQ [Hin13] networking library, using exclusive pair sockets which provide one-to-one bidirectional communication. The parameters and the output images are serialized using Protobuf [Goo08]. A reason for utilizing these two libraries is their support for a wide variety of programming languages and runtime environments. Although both the client and the server are regular desktop applications, these allow the development of alternative clients compatible with the server, such as web and mobile apps. This in turn makes it possible to access geodesic ray tracing from devices were local rendering would not be feasible.

The user interface is capable of launching a server as a subprocess and connecting to it automatically for local rendering. In a distributed setting, the user interface makes the requests to the

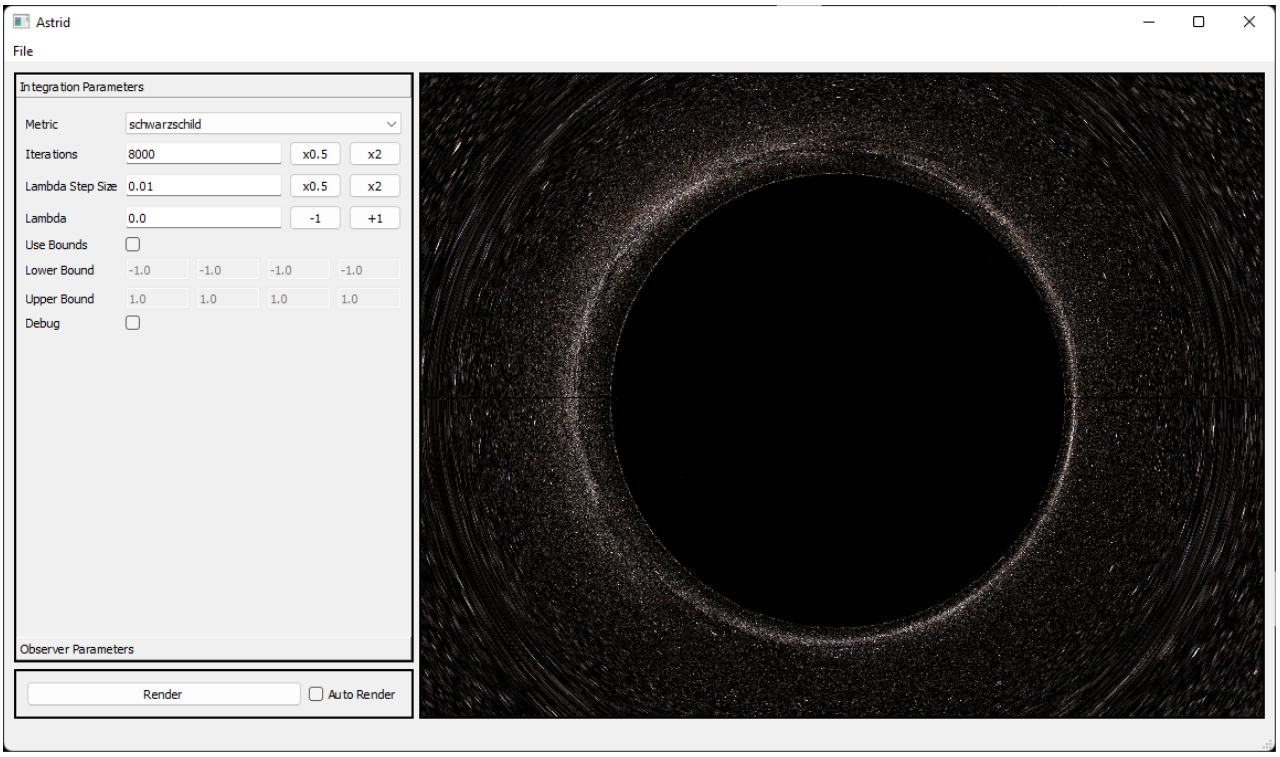

Figure 4: *An overview of Astrid's user interface. The viewport displays a Schwarzschild black hole with the Tycho skymap [NAS09] in the background.*

root node which responds with the gathered image. For more detail and a tutorial on Astrid, we refer the reader to its documentation.

8. Conclusion & Future Work

We have presented a high-performance geodesic ray tracing system capable of scaling from single computers to compute clusters. Based on the performance study conducted on the system, we have shown that it is capable of near-interactive geodesic ray tracing on a Tier-3 cluster. An accompanying user interface capable of remote rendering was also presented, with the intent of reaching users without programming experience. Astray and Astrid are open-source software distributed under the BSD 3-Clause license, accessible at https://github.com/vrgrouprwth/astray and https://github.com/vrgrouprwth/astrid respectively.

In the future, more sophisticated load balancing and space partitioning methods could be adapted from standard ray tracing to accelerate the computation. We further consider regularly sampling the Christoffel symbols into a 4D data structure in a precomputation step and probing this structure at run-time in order to improve performance. Another direction is spectral rendering, which is necessary for the correct simulation of several effects such as the Doppler red/blue shift. Aside from technical improvements, we intend to reach out to the astrophysics community to promote the library for research and education.

9. Acknowledgements

The authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the individual state governments for supporting this work/project as part of the NHR funding.

References

[AAA*21] AKIYAMA K., ALGABA J. C., ALBERDI A., ALEF W., ANANTUA R., ASADA K., AZULAY R., BACZKO A.-K., BALL D., BALOKOVIĆ M., ET AL.: First M87 Event Horizon Telescope Results. VII. Polarization of the Ring. *The Astrophysical Journal Letters 910*, 1 (2021), L12. doi:10.3847/2041-8213/abe71d. 1

[AGL05] AHRENS J., GEVECI B., LAW C.: ParaView: An End-User Tool for Large-Data Visualization. *The Visualization Handbook 717*, 8 (2005). 6

[AM11] AHNERT K., MULANSKY M.: Odeint–solving ordinary differential equations in C++. In *AIP Conference Proceedings* (2011), vol. 1389, American Institute of Physics, pp. 1586–1589. 4

[BH12] BELL N., HOBEROCK J.: Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*. 2012, pp. 359–371. doi:10.1016/B978-0-12-385963-1.00026-5. 4

[CMOP18] CHAN C.-K., MEDEIROS L., OZEL F., PSALTIS D.: GRay2: A General Purpose Geodesic Integrator for Kerr Spacetimes. *The Astrophysical Journal 867*, 1 (Oct. 2018), 59. doi:10.3847/1538-4357/aadfe5. 3

[CPO13] CHAN C.-K., PSALTIS D., OZEL F.: GRay: A Massively Parallel GPU-Based Code for Ray Tracing in Relativistic Spacetimes.

The Astrophysical Journal 777, 1 (Oct. 2013), 13. doi:10.1088/0004-637x/777/1/13. 3

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen: A C++ Linear Algebra Library, 2010. URL: https://eigen.tuxfamily.org. 4

[Goo08] GOOGLE: Protocol Buffers, 2008. URL: https://developers.google.com/protocol-buffers. 6

[Har03] HARTLE J. B.: Gravity: an introduction to einstein's general relativity, 2003. 2

[HBG*05] HINDMARSH A. C., BROWN P. N., GRANT K. E., LEE S. L., SERBAN R., SHUMAKER D. E., WOODWARD C. S.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS) 31*, 3 (2005), 363–396. 4

[Hin13] HINTJENS P.: *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013. 6

[KMA*12] KUCHELMEISTER D., MÜLLER T., AMENT M., WUNNER G., WEISKOPF D.: GPU-based four-dimensional general-relativistic ray tracing. *Computer Physics Communications 183*, 10 (2012), 2282–2290. doi:10.1016/j.cpc.2012.04.030. 3

[MBW15] MÜLLER T., BOBLEST S., WEISKOPF D.: Visualization Showcase: General-Relativistic Black Hole Visualization. In *Eurographics Symposium on Parallel Graphics and Visualization* (2015), The Eurographics Association, pp. 29–32. doi:10.2312/pgv.20151152. 3

[MG09a] MÜLLER T., GRAVE F.: Catalogue of Spacetimes, 2009. doi:10.48550/ARXIV.0904.4184. 3

[MG09b] MÜLLER T., GRAVE F.: Motion4D – A library for lightrays and timelike worldlines in the theory of relativity. *Computer Physics Communications 180*, 11 (2009), 2355–2360. doi:10.1016/j.cpc.2009.07.014. 3

[MPI21] MPI FORUM: *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021. URL: https://mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. 3, 4

[Mü14] MÜLLER T.: GeoViS - Relativistic ray tracing in four-dimensional spacetimes. *Computer Physics Communications 185*, 8 (2014), 2301–2308. doi:10.1016/j.cpc.2014.04.013. 3

[NAS09] NASA / GODDARD SPACE FLIGHT CENTER SCIENTIFIC VISUALIZATION STUDIO: The Tycho Catalog Skymap - Version 2.0, 2009. URL: https://svs.gsfc.nasa.gov/3572. 7

[PMNJ18] PIHAJOKI P., MANNERKOSKI M., NÄTTILÄ J., JOHANSSON P. H.: General Purpose Ray Tracing and Polarized Radiative Transfer in General Relativity. *The Astrophysical Journal 863*, 1 (Aug. 2018), 8. doi:10.3847/1538-4357/aacea0. 3

[PRCB22] PELLE J., REULA O., CARRASCO F., BEDERIAN C.: Skylight: a new code for general-relativistic ray-tracing and radiative transfer in arbitrary spacetimes, 2022. doi:10.48550/ARXIV.2206.06429. 3

[PYYY16] PU H.-Y., YUN K., YOUNSI Z., YOON S.-J.: Odyssey: A Public GPU-Based Code for General Relativistic Radiative Transfer in Kerr Spacetime. *The Astrophysical Journal 820*, 2 (Mar. 2016), 105. doi:10.3847/0004-637x/820/2/105. 3

[VCAVL*22] VELÁSQUEZ-CADAVID J., ARRIETA-VILLAMIZAR J., LORA F., PIMENTEL O., OSORIO-VARGAS J.: OSIRIS: a new code for ray tracing around compact objects. *The European Physical Journal C 82* (Feb. 2022). doi:10.1140/epjc/s10052-022-10054-0. 3

[VE21] VERBRAECK A., EISEMANN E.: Interactive Black-Hole Visualization. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021), 796–805. doi:10.1109/TVCG.2020.3030452. 3

[VPGP11] VINCENT F. H., PAUMARD T., GOURGOULHON E., PERRIN G.: GYOTO: a new general relativistic ray-tracing code. *Classical and Quantum Gravity 28*, 22 (Oct. 2011). doi:10.1088/0264-9381/28/22/225011. 3

[WBE*06] WEISKOPF D., BORCHERS M., ERTL T., FALK M., FECHTIG O., FRANK R., GRAVE F., KING A., KRAUS U., MÜLLER T., NOLLERT H.-P., MENDEZ I., RUDER H., SCHAFHITZEL T., SCHAR S., ZAHN C., ZATLOUKAL M.: Explanatory and illustrative visualization of special and general relativity. *IEEE Transactions on Visualization and Computer Graphics 12*, 4 (2006), 522–534. doi:10.1109/TVCG.2006.69. 3

[Wei00] WEISKOPF D.: Four-Dimensional Non-Linear Ray Tracing as a Visualization Tool for Gravitational Physics. In *Proceedings of the Conference on Visualization '00* (Washington, DC, USA, 2000), VIS '00, IEEE Computer Society Press, p. 445–448. doi:10.1109/VISUAL.2000.885728. 3

[Whi22] WHITE C. J.: Blacklight: A General-Relativistic Ray-Tracing and Analysis Tool, 2022. doi:10.48550/ARXIV.2203.15963. 3

[WSE04] WEISKOPF D., SCHAFHITZEL T., ERTL T.: GPU-Based Nonlinear Ray Tracing. *Computer Graphics Forum 23*, 3 (2004), 625–633. doi:10.1111/j.1467-8659.2004.00794.x. 3