A simulated telescope's view of the sky in the presence of a Schwarzschild blackhole

Steven Dorsher

May 5, 2016

1 Introduction

With the LIGO detection, many people are interested in the question of how blackholes make the sky appear to the external observer. Understanding the question of how they will lens the presence of stars behind them could help make blackholes without accretion disks or partners indirectly observable. The problem of a Schwarzschild blackhole in a vaccuum with light passing by it from a static field of stars is not a particularly challenging problem and appears to have been solved numerically in the 1970's or 1980's, though I coudn't find a specific source. Many people since then have solved more difficult problems for rotating, perturbed, or asymptotically cosmological spacetimes.

In this project, I numerically integrate the eight coupled ordinary differential equations governing light rays (null geodesics) backward from a telescope plane near an ordinary (no spin or charge) in a vaccuum. I iterate over many pixels in the telescope plane, and determine their origin on the "sky" at some finite distance away. From that, I reconstruct a png file of what the sky looks like in the presence of a Schwarzschild black hole, at the location of the Earth, and a telescope, $700M$ away. Henceforth, the mass of the blackhole will simply be referred to as M. I will use units of $c = G = 1$.

2 The geodesic equations

2.1 Overview

In general relativity, the differential equations governing the path matter or light takes in a vacuum are called the geodesic equations. They are second order rank four tensor equations, which are separable, resulting in eight coupled ordinary differential equations. The geodesic equation states that a free particle, which feels no forces other than the curvature of spacetime that causes gravity, follows a path that is the shortest path between two points.

2.2 Computation

The geodesic equation is given by

$$\frac{d^2x^\mu}{d\lambda^2} = -\Gamma^\mu_{\alpha\beta}\frac{dx^\alpha}{d\lambda}\frac{dx^\beta}{d\lambda} \tag{1}$$

Here x^μ is the μth component of a spacetime four-vector, $\Gamma^\mu_{\alpha\beta}$ are the Christoffel connections, and λ is an affine parameter that plays the role of proper time for a null geodesic, since a null geodesic has no rest frame. There is an implicit summation convention over repeated indices, where upper and lower indices are connected by the metric $g_{\mu\nu}$ and the inverse metric $g^{\mu\nu}$. [1]

The metric determines the distance between two points in a curved spacetime. In a flat spacetime like the one near Earth (far away from relativistic sources), the Minkowski metric is

$$ds^2 = -dt^2 + dx^2 + dy^2 + dz^2 = -dt^2 + dr^2 + r^2(d\theta^2 + \sin^2\theta d\phi^2) \tag{2}$$

In matrix form this can be written:

$$\eta_{\mu\nu} = \eta^{\mu\nu} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

In the Schwarzschild metric,

$$g_{\mu\nu} = \begin{pmatrix} -(1-\frac{R}{r}) & 0 & 0 & 0 \\ 0 & (1-\frac{R}{r})^{-1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2\theta \end{pmatrix} \tag{4}$$

where $R = 2M$ is the Schwarszchild radius of the blackhole, beyond which no light can escape. [1] Everything referenced in this section can be looked up in a book [1], but I have rederived it in Mathematica as well. The goal was to directly obtain C++ code from Mathematica, but the scope of the project proved too large and instead I directly parallelized the python prototype.

The inverse metric is given by

$$g^{\mu\nu} = \begin{pmatrix} -(1-\frac{R}{r})^{-1} & 0 & 0 & 0 \\ 0 & (1-\frac{R}{r}) & 0 & 0 \\ 0 & 0 & r^{-2} & 0 \\ 0 & 0 & 0 & r^{-2}\csc^2\theta \end{pmatrix} \tag{5}$$

Note that both the metric and the inverse metric are diagonal and static (independent of time) in these coordinates, symmetries that will lead to a reduction in the number of non-zero Christoffel connections.

The Christoffel connections are computed according to the following equation:

$$\Gamma^{\mu}_{\alpha\beta} = \frac{1}{2} g^{\mu\gamma}(\partial_\beta g_{\gamma\alpha} + \partial_\alpha g_{\gamma\beta} - \partial_\gamma g_{\alpha\beta}) \tag{6}$$

Where $\partial_\gamma = \frac{\partial}{\partial x^\gamma}$ for four-coordinates x. [1] These result in 12 independent Christoffel symbols for the Schwarzschild metric, dependant upon functions of the coordinates, except time.

Setting the four-velocity $u^\mu = \frac{dx^\mu}{d\lambda}$. it is possible to split the four components of the geodesic equation (Equation 1) into eight coupled ordinary differential equations using these Christoffel symbols. The results are given in the next section.

2.3 Result

The eight coupled geodesic equations are

$$\frac{dt}{d\lambda} = u_t, \tag{7}$$

$$\frac{dr}{d\lambda} = u_r, \tag{8}$$

$$\frac{d\theta}{d\lambda} = u_\theta, \tag{9}$$

$$\frac{d\phi}{d\lambda} = u_\phi, \tag{10}$$

$$\frac{du_t}{d\lambda} = -r^{-1}F^{-1}u_t u_r, \tag{11}$$

$$\frac{du_r}{d\lambda} = -\frac{1}{2}Rr^{-2}Fu_t^2 + \frac{1}{2}Rr^{-2}F^{-1}u_r^2 + rFu_\theta^2 + rF\sin^2\theta u_\phi^2, \tag{12}$$

$$\frac{du_\theta}{d\lambda} = -2r^{-2}u_r u_\theta + \frac{1}{2}\sin(2\theta)u_\phi^2, \tag{13}$$

$$\frac{du_\phi}{d\lambda} = -2r^{-1}u_r u_\phi - 2\cot\theta u_\theta u\phi \tag{14}$$

where $F = 1 - \frac{R}{r}$.

3 The adaptive fourth order Runga-Kutta method

Because the coordinates become singular at the event horizon– that is, it takes an infinite amount of time for a particle, as seen by an outside observer, to cross the event horizon– integrations that pass close to the event horizon or plunge into it will need an adaptive step size. It will also be necessary to implement some threshold in adaptive step size, close to the horizon, below which the light is considered to be close enough to the black hole that it will probably enter the event horizon if evolved for eternity. The adaptive step size method chosen was the fourth order adaptive Cash-Karp Runge-Kutta method, as explained in reference [2].

3.1 Method

The general approach of the Cash-Karp algorithm is to take both a fifth order and an embedded fourth order Runge-Kutta step, then use the difference as the error, Δ_1. Since the error should scale as the step size, h_1, to the fifth, the new step can be found by comparison to some desired error, Δ_0, as follows:

$$h_0 = h_1 \left| \frac{\Delta_0}{\epsilon \Delta_1} \right|^0 .2 \tag{15}$$

If h_0 is smaller than h_1, the algorithm retries the step to obtain better precision. Otherwise, h_0 is the new step size for the next step forward in "time" (in our case, affine parameter). Here ϵ is the tolerance of the algorithm.

There are some complicating details. The step size should not grow or shrink by too large a factor at each retrial or for each new λ step. The exponent for changing h is subtly different in the presence of a Δ_0 that is proportional to h, so we choose the more stringent requirement of the two exponents to be safe. We also add a saftey factor to be sure that we do not overshoot. [2]

The specific algorithm for the Cash-Karp Runge-Kutta is given by

$$\begin{aligned}
k_1 &= hf(x_n, y_n) && (16)\\
k_2 &= hf(x_n + a_2 h, y_n + b_{21} * k_1) && (17)\\
&... \\
k_6 &= hf(x_n + a_6 h, y_n + b_{61} k_1 + ... + b_{65} k_5) && (18)\\
y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 && (19)\\
y^*_{n+1} &= y^*_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 && (20)\\
\Delta_1 &= y_{n+1} - y^*_{n+1} && (21)
\end{aligned}$$

The scale, Δ_0, was set using the following dynamically changing function that takes into account both the absolute size of the field and the derivative of the field, as well as some small number to prevent it from becoming zero. The small number, $\beta = 10^-30$.

$$\Delta_0 = |y| + \left| h \frac{dy}{d\lambda} \right| + \beta \tag{22}$$

where y is the 8-vector of the 8 coupled evolving fields. The maximum resulting step size was used.

3.2 Testing using a simple harmonic oscillator ODE

I have tested this code using a simple harmonic oscillator ordinary differential equation. I ran it for 1000 time steps, with a period of 5 time units and an initial time step of 0.01 time units. The error grew roughly linearly with time, due to truncation error, and in Figures reffig:sho1,fig:sho2,fig:sho3, the tolerance was decreased by a factor of two and a factor of fourand correspondingly, the error at equal times is decreased by a factor of two and a factor of four.

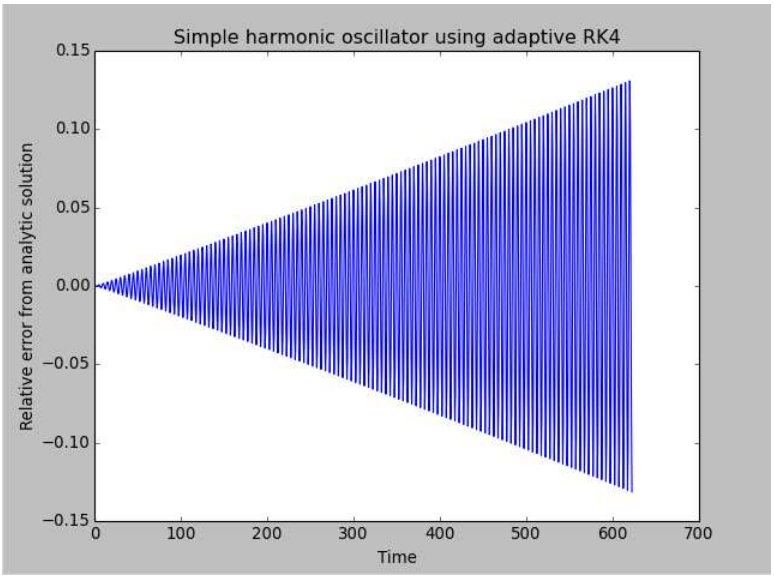

Figure 1: Relative error for a simple harmonic oscillator with period of 5 over 1000 time steps with initial time step 0.01 and tolerance of 10^-4.

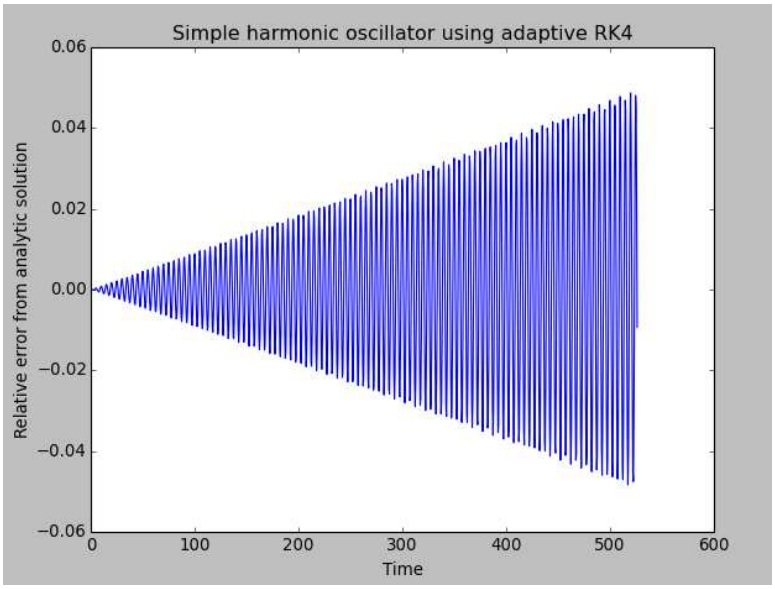

Figure 2: Relative error for a simple harmonic oscillator with period of 5 over 1000 time steps with initial time step 0.01 and tolerance of 5×10^-5. Note that it has half the tolerance of Figure 3.2, and at equal times, it has half the error.

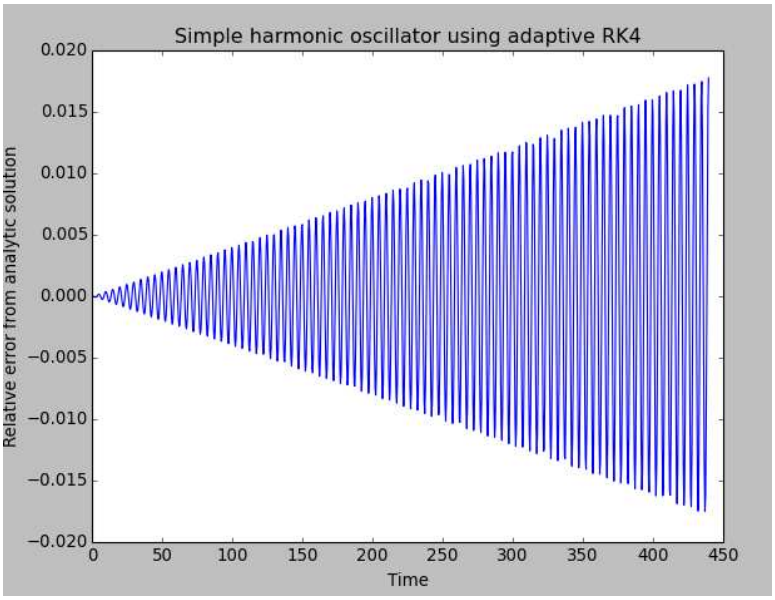

Figure 3: Relative error for a simple harmonic oscillator with period of 5 over 1000 time steps with initial time step 0.01 and tolerance of $2.5 \times 10^{-}5$. Note that it has a quarter the tolerance of Figure 3.2, and at equal times, it has a quarter the error.

4 Testing the geodesic equations with orbits with elipticity

To test the geodesic equations I implemented in python with something with an easily identifiably correct solution, I integrated geodesics of massive particles on "eliptical" orbits around the blackhole. Eliptical orbits in a Schwarzchild spacetime are not closed orbits, but rather repeat with different periods in phi and radius, precessing around the black hole.

4.1 Initial data

The initial data was found in Reference **??**. Energy, E, and angular momentum, L, are given by

$$E^2 = \frac{(p-2-2e)(p-2+2e)}{p(p-3-e^2)} \tag{23}$$

$$L^2 = \frac{p^2 R^2}{4(p-3-e^2)} \tag{24}$$

where p is the semilatus rectum (which governs the distance of the orbit away from the blackhole) and e is the eccentricity. I chose to start my orbit at apastron. The distance of apastron can be calculated from

$$r_{ap} = \frac{pR}{2(1-e)} \tag{25}$$

Since I started in the $\theta = \pi/2$ plane, $d\theta/d\lambda = 0$ for all affine parameters λ. Since the metric is static, I could choose to start at any arbitrary time, so I chose $t = 0$. Similarly, the evolution is independant of starting affine parameter, so I began at $\lambda = 0$. I chose $p = 10$, $R = 2$, $e = 0.2$.

4.2 Results

Figure 4.2 demonstrates the periodic nature of the radial coordinate, the increasing and modulated nature of the phi coordinate due to the changing velocities throughout the orbit, and the fact that the two periods are not the same. Note that the period of the radial oscillations is between two pi and four pi.

5 Null geodesic initial data

A null geodesic is the path that light travels. It has no rest frame, and no proper time. For null geodesics, I integrate them backward in time from when they arrive at the telescope plane to where they are emitted on the sky. Therefore, their initial data is really the state in which they must end, physically. The integration initial data needs to be chosen such that the rays are "emitted" (running time backward) perpendicular to and inward from the telescope plane.

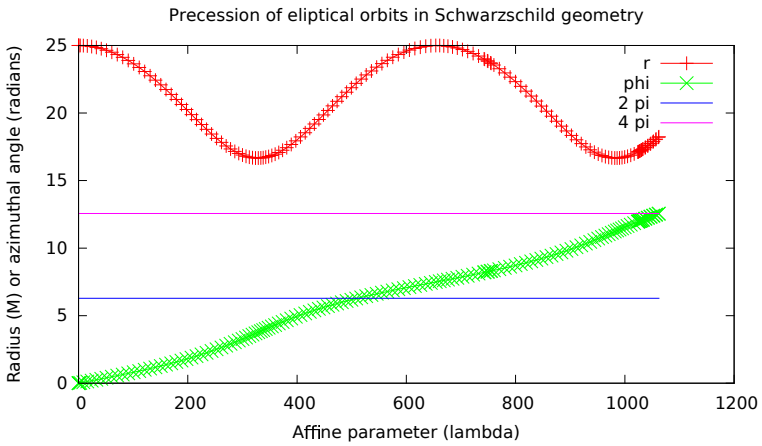

Figure 4: Eliptical, precessing orbit.

The center of the telescope plane is at a radius of 700 along the x-axis. Thus, spacetime initial data can be given quite simply in rectangular coordinates. $(t, x, y, z) = (0, x_{tele}, y_{pixel}, z_{pixel})$. Likewise, $(u_x, u_y, u_z) = (1, 0, 0)$, but the time component of the four-velocity must be chosen consistently with the null geodesic condition:

$$g_{\mu\nu} u^\mu u^\nu = 0 \tag{26}$$

The end result is:

$$u_t = F^{-1} \sqrt{F^{-1} * u_r^2 + r^2 u_\theta^2 + r^2 \sin^2\theta u_\phi^2} \tag{27}$$

where $F = 1 - \frac{R}{r}$ again.

Converting rectangular coordinates to spherical coordinates, we have

$$\begin{aligned} t &= t & (28)\\ r &= \sqrt{x^2 + y^2 + z^2} & (29)\\ \theta &= \cos^{-1}\frac{z}{r} & (30)\\ \phi &= \tan^{-1}\frac{y}{x} & (31) \end{aligned}$$

Converting rectangular velocities to spherical velocities requires chains of partial derivatives. For example:

$$u_r = \frac{\partial r}{\partial x} u_x + \frac{\partial r}{\partial y} u_y + \frac{\partial r}{\partial z} u_z \tag{32}$$

Similarly for u_θ and u_ϕ. These simplify substantially given the specific expression for the rectangular velocities. The resulting spherical velocities, specific to

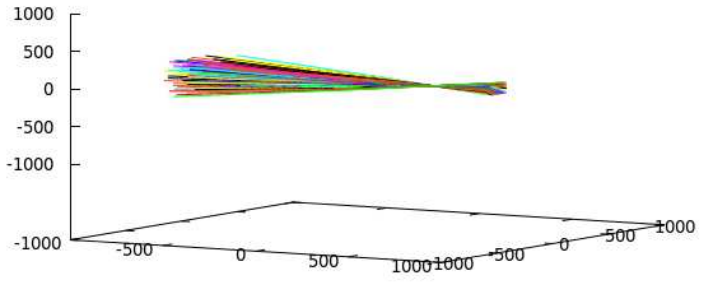

Figure 5: 100 Null geodesics viewed from the side. Some near the middle of the telescope plane (right hand side) terminate inside the black hole and thus result in an image of nothing. The rest produce images of the sky at the point where they terminate. This image is flattened for two reasons. One is that the initial image plane was half as large in z as in y. The other reason is that the z scale is smaller than the y scale on this plot.

these initial conditions, are:

$$u_r = \frac{x}{r} \tag{33}$$

$$u_\theta = \frac{xz}{r^{3/2}\sqrt{1-\frac{z^2}{r^2}}} \tag{34}$$

$$u_\phi = -\frac{y}{x^2+y^2} \tag{35}$$

To determine the end conditions of the computation, I will have to stop the loop when the light has reached the edge of the "sky" at $r = 1000$ or approaches the blackhole horizon at $r = 2$. In practice, the computation stops at 2 plus 10^{-10} if the step in lambda is less than $10^{-}14$.

5.1 The resulting orbits

Figure 5.1 shows the resulting orbits for a wide image plane of 200 by 100 with 100 evenly sampled geodesics. No remarkable effects can be seen in this figure, though I know from printed output that some terminate inside the blackhole.

6 Generating a telescope image

There are several steps in creating an image map. After integrating backward to obtain theta and phi on the sky, the theta and phi coordinates must be

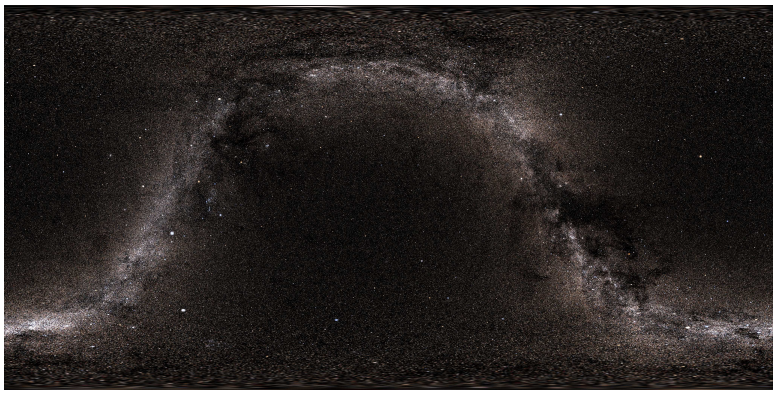

Figure 6: Tycho survey sky map from NASA. Full sky, theta along vertical axis, phi along horzontal axis.

converted to pixels on the sky image map. Png images are stored in flat row flat pixelformat for the purposes of this project so that they may more easily be diveded into chunks and distributed to processes. In flat row flat pixel format, each pixel is represented by three uint8's in an array. After one pixel comes the next in that row. After that row comes the next row, all in one long array with no sub-arrays. The index of the start pixel for the telescope array (the output array) is determined by:

```
telestart=(xpix+ypix*pixelwidth)*3
```

The start pixel for the location on the sky to be read is determined by:

```
xout=int(phi*skypixelwidth/2./pi)
yout=int(theta*skypixelheight/pi)
skystart=(xout+yout*skypixelwidth)*3
```

The three pixels following telestart are set to the three pixels following skystart if the geodesic ended in the sky, and are set to red if the geodesic ended in the blackhole. The skymap used for the external sky is a fully sky NASA image of the Milky Way from the Tycho survey shown in Figure 6. The horizontal axis is phi and the vertical axis is theta. It has a resolution of 4096 by 2048.

7 Profiling

7.1 Serial code profile

Code is given in Section 14. The code was profiled using cProfile with the cumtime command to obtain the "tottime", or total time that each function ran. The results are shown below.

```
64100465 function calls (64098561 primitive calls) in 456.492 seconds
```

```
Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.005    0.005  456.495  456.495 geodesic12.py:1(<module>)
        1    0.049    0.049  455.993  455.993 geodesic12.py:301(main)
     2500   14.274    0.006  437.151    0.175 geodesic12.py:189(integrateNullGeo
   457700  233.164    0.001  406.786    0.001 geodesic12.py:75(adaptiveRK4)
  3495212   84.325    0.000  119.433    0.000 geodesic12.py:161(geodesic)
 11551770   46.510    0.000   46.510    0.000 {numpy.core.multiarray.array}
  1518756    4.131    0.000   27.738    0.000 fromnumeric.py:1621(sum)
  1518756    2.732    0.000   21.889    0.000 _methods.py:23(_sum)
  4556268    5.910    0.000   21.229    0.000 function_base.py:786(copy)
  1518769   19.156    0.000   19.156    0.000 {method 'reduce' of 'numpy.ufunc'
        1    3.721    3.721   18.789   18.789 png.py:1974(read_flat)
 27846638   12.796    0.000   12.796    0.000 {range}
     2049    0.024    0.000   11.263    0.005 png.py:1688(iterstraight)
     2048    0.012    0.000   10.919    0.005 png.py:1472(undo_filter)
     2048   10.630    0.005   10.905    0.005 png.py:2406(undo_filter_up)
```

These results demonstrate that the adaptive Runge-Kutta method requires most of the time, but it is not possible to parallelize over iterations of the adaptive Runge-Kutta method because subsequent steps depend on previous steps. The closest, independent, surrounding option is to parallelize over pixels. However, we expect load balancing issues because some integrations will run much longer than others, especially those that approach the blackhole (due to the coordinate singularity). In fact, these problems are seen in Sections **??**.

8 Timing estimate

All of my timing was done comparing the parallel program with one process to the parallel program with more processes. I realize there is some overhead of paralellization, but it should be much less than the time required for integration in a densely enough sampled map.

The run time for a single process should be, at most, equal to the number of pixels (wh) times the Runge-Kutta step time (t) times the maximum number of Runge-Kutta steps per pixel (N).

$$T = whtN \tag{36}$$

where w is width and h is height in pixels. For a single process, for $w = 32$ and $h = 32$, I find $t = 0.000472310777906$ s, $N = 1022$, and $T = 97.651763916$. The upper bound I obtain from these numbers is 494 s, which is a factor of 5 larger than the measured runtime. The reason for the difference is primarily load imbalancing– most pixels don't take nearly so many steps (N) to run as the longest pixel, which plunges into the blackhole.

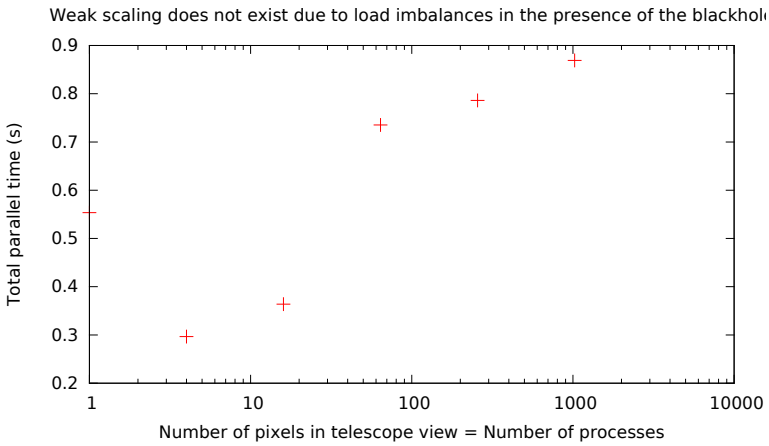

Figure 7: Weak scaling is not present– the load increases with quadrupoling of both processors and load.

9 Weak scaling

Weak scaling states that if you double your work load and double your number of processes, you should get the same time. I tested this with doubling in both the width and height dimension and quadrupoling my processes. My data is shown below.

```
#w h n N trk4 T
1 1 1 948 0.000573131857039 0.553431987762
16 16 256 975 0.000633819052514 0.786194086075
2 2 4 168 0.00106210084189 0.296684980392
32 32 1024 1022 0.000523686408997 0.869103193283
4 4 16 193 0.0014385967057 0.363858938217
8 8 64 940 0.000662297898151 0.735347032547
```

Figure 9 shows this data as a function of number of pixels or number of processes. For zero pixels, the time is high because the single pixel is located at the center of the telescope field of view and plunges into the blackhole. At low numbers of pixels, the field of view is poorly sampled and no geodesics cross the horizon. After 64 pixels, the field is well enough sampled to have geodesics crossing the horizon, and again the execution time rises, because of the load imbalancing.

We can futher investigate this load imbalancing issue. I have plotted the maximum number of adaptive Runge-Kutta iterations (N) as a function of processes in Figure 9. It demonstrates that this same sudden change occurs, at the same number of processes, in the maximum number of integration steps. This is strong evidence for load imbalancing as the source of the problem.

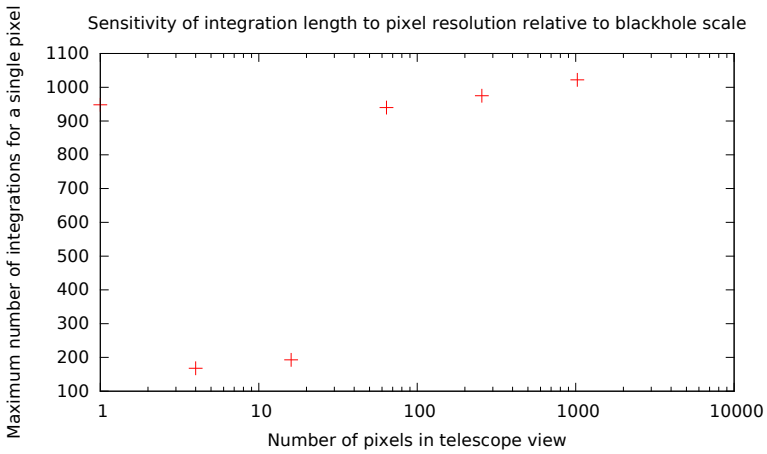

Figure 8: The maximum integration length, N, as a function of number of pixels in telescope view, or number of processes. This demonstrates the sensitivity to load imbalancing in weak scaling.

10 Amdahl's law

Amdahl's law states that speedup is given by

$$S(n) = \frac{T_1}{T_n} = \frac{1}{1 - p + \frac{p}{n}} \tag{37}$$

where p is the percent parallelized and n is the number of processes. Strong scaling is how the speedup varies with a fixed problem size. To determine this, I used a 32 by 32 pixel telescope map on Blue Waters. My timing data is shown below.

```
#w h n N trk4 T
32 32 1024 1022 0.000820902677683 1.10371899605
32 32 1 1022 0.000472310777906 97.651763916
32 32 16 1022 0.000555939086781 13.3469061852
32 32 256 1022 0.000575474568538 2.14842200279
32 32 4 1022 0.000547813355208 36.3236739635
32 32 64 1022 0.00068316065485 3.74180793762
```

Figure 10 demonstrates that I have sub-Amdahl's law scaling at all number of processes, but that I do in fact achieve scaling of the same power law form, just with a different exponent. I performed a fit to determine the percent parallel. Within error on order 10^{-5}, the percent parallel was exactly 1. My timing data finds a percent parallel on order 10^{-4} different from 1, and that is the number I have used in this plot. That is unsurprising, given that I have defined my total time to exclude input and output, which leaves little else except the parallel region. Probably the reason for the sub-Amdahl's law scaling is load balancing

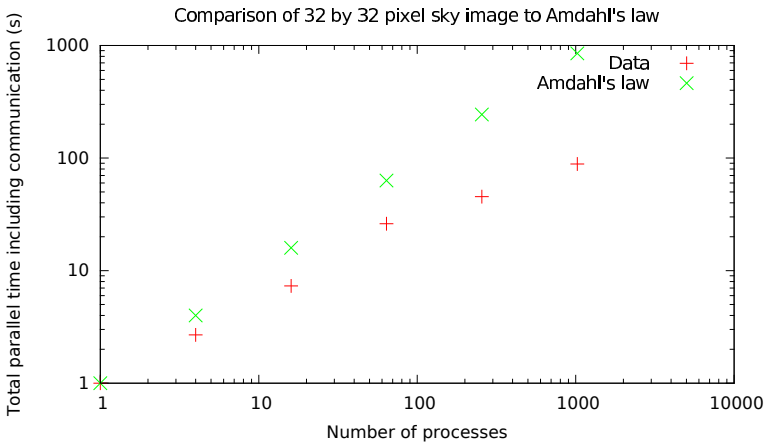

Figure 9: Sub Amdahl's law scaling is obtained due to load balancing problems.

problems, as demonstrated in Sections **??**. The reason the sudden difference at a certain process number occurs in weak scaling but not strong scaling is because in weak scaling the process number is tied to the pixel density, which is tied to whether or not there is a geodesic that enters the blackhole horizon, a binary event.

11 Image

I was unable to obtain a pixel map using one of the supercomputers, so Figure 11, generated using four processes on my laptop, is my final result. Super Mike does not have mpi4py. Blue Waters does not have the python png package. With further time, it is possible a solution would have existed, but I did not have it.

12 Summary

In summary, I have implemented a parallel program in python, using mpi4py, to integrate light rays backward in time from a telescope field of view to their source position on the sky, forming an image on a telescope plane. It is parallelized by pixel. In the image, the absense of light inside the horizon, the overdensity of stars near the blackhole horizon, and the warping of the stellar field at several times the radius of the horizon are all visible.

My timing estimates are imprecise due to load imbalancing, but my upper bound is not violated. My program does not have good weak scaling, with a sudden change where the pixel resolution goes from containing pixels that have trajectories that take them into the blackhole and those that do not. In strong scaling, the form of Amdahl's law is echoed, but it appears to be roughly

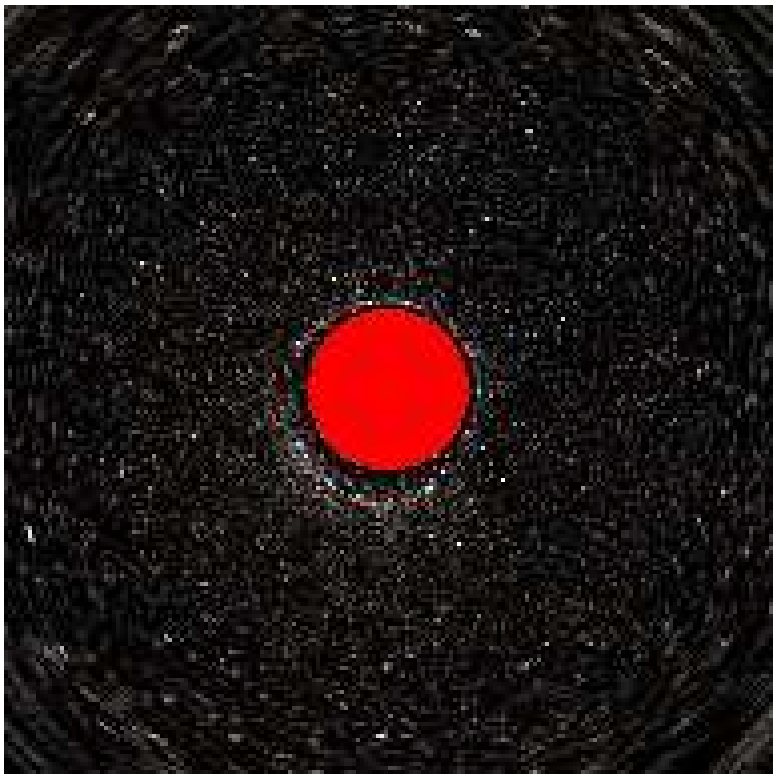

Figure 10: A 192 by 192 pixel image in a 50 by 50 M image plane generated using 4 processes. The overdensity of stars close to the black hole horizon is visible. So is the warping of the image at several times the horizon radius. From previous images I've seen, I know that it should return to a normal star field far outside this ring.

the square root in magnitude. All of these issues are due to load imbalancing, which must be addressed if this project will be continued. If this project will be continued, it is likely that it will be rewritten in C++ as well.

13 Bibliography

References

[1] Sean M. Carroll *An Introduction to General Relativity Spacetime and Geometry* Addison Wesley, San Francisco, 2004.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery *Numerical Recipes in Fortran: The Art of Scientific Computing* Cambridge University Press, New York, 2nd edition, 1992.

[3] Anath Grama, Anshul Gupta, George Karypis, Vipin Kumar *Introduction to Parallel Computing* Pearson Education Limited, Essex, 2nd edition, 2003.

[4] Adam Pound, Eric Poisson *Osculating orbits in Schwarzchild spacetime, with an application to extreme mass-ratio inspirals* Physical Review D 77, 0044013, 2008.

14 Appendix: Serial code

```
import numpy as np
from math import pi,sin,cos,sqrt,atan,acos
from matplotlib import pyplot,image
import PIL, png
from scipy import misc

a=np.array([0, .2, .3, .6, 1., 7./8.])
b=np.array([[0.,0.,0.,0.,0.],[.2,0.,0.,0.,0.],[3./40.,9./40.,0.,0.,0.],[3./10.,
c = np.array([37./378., 0., 250./621., 125./594., 0., 512./1771.])
cstar = np.array([2825./27648., 0., 18575./48384., 13525./55296., 277./14336., 0
dc = np.array([277./64512.,0.,-6925./370944.,6925./202752.,277./14336.,-277./708
lena=len(a)



# t is lamb, affine parameter
# y is u, four velocity, or x, four position

def initialize(pixelcoord,Rplane,pixelheight,pixelwidth,skypixelwidth,skypixelhe
    #set origin of pixel plane at x axis
    t = 0.
```

```
    x = Rplane
    y = (pixelcoord[0]-pixelwidth/2.)*imagewidth/float(pixelwidth)
    z = (pixelcoord[1]-pixelheight/2.)*imageheight/float(pixelheight)
    r= sqrt(x*x+y*y+z*z)
    phi = atan(y/x)
    theta = acos(z/r)

    #initial u perpendicular to plane.
    #magnitude of u is arbitrary— affine parameter makes it rescalable
    #(ut)^2-(uy)^2-(ux)^2-(uz)^2=0 so ut = +-ux
    #for x decreasing as t increases, ut = -ux (inward)
    uy = 0.
    uz = 0.
    ux = 1.


    invr = 1./r
    invrsq = invr*invr
    #rhosq = x*x + y*y
    #facrrho= rhosq+r*r
    #for this specific case, where ux = 1:
    ur = -x/r
    #utheta = -x*z*z/sqrt(rhosq)/r/facrrho
    #uphi = y/rhosq
    utheta = x*z*invr*invrsq*sqrt(1.-z*z*invrsq)
    uphi = -y/(x*x+y*y)
    rmRs = r-Rs
    st = sin(theta)
    ut = sqrt((ur*ur*r/rmRs+r*r*utheta*utheta+r*r*st*st*uphi*uphi)/rmRs*r)
    initcoords =np.array([t, r, theta, phi, ut, ur, utheta, uphi])
#     coords = initcoords
#     rmRs2 = coords[1]-Rs
#     testnull = -rmRs2/coords[1]*coords[4]*coords[4]+coords[5]*coords[5]*coords[
    #testnull = -rmRs/r*ut*ut+ur*ur*r/rmRs+r*r*(utheta*utheta+st*st*uphi*uphi)
    #print(testnull)
    #print(testnull/max(np.absolute(initcoords[4:8])))
    return initcoords

def initializeElliptical(eccentricity,semilatusr,Rs):
    r2 = semilatusr*0.5*Rs/(1.-eccentricity)
    print(eccentricity, semilatusr, Rs,r2)
    theta = pi/2.
    phi = 0.
    t = 0.
    utheta = 0.
    temp = 1./(semilatusr - 3. - eccentricity*eccentricity)
```

```
    angularL = 0.5*semilatusr*Rs*sqrt(temp)
    energy=sqrt((semilatusr-2.-2.*eccentricity)*(semilatusr-2.+2.*eccentricity)/
    uphi = angularL/r2/r2
    ur = 0.
    ut = energy/(1.-Rs/r2)
    return np.array([t,r2,theta,phi,ut,ur,utheta,uphi])

def adaptiveRK4(t,y,h,func,maxfunc,arg,yscale,epsilon):
#    fadapt = open("adaptout.txt", "a")
    leny=len(y)
    safetyfac = 0.9
    pgrow =-0.20
    pshrink =-0.25
    errcon = 1.89e-4 #see NR in Fortran
    hnew=h/2.
    #hlast = h
    while True:
        # j and i are reversed from Numerical Recipes book (page 711)
        #loop over y indices
        k=np.zeros((leny,lena))
        tprimearg=t
        yprime = np.copy(y)
        yprimestar = np.copy(y)
        for j in range(0,len(a)): #for all terms summed in method
            tprimearg = t+a[j]*h
            yprimearg = np.copy(y)
            for n in range(0,leny): #over all variables in y vector
                for i in range(0,j): #over all indices of k
                    #update for next term of k in calculation
                    yprimearg[n]+=b[j,i]*k[n,i]
            k[:,j]=h*func(tprimearg,yprimearg,arg)
        yprime = y+np.sum(np.multiply(c,k),axis=1)
        yerr =  np.sum(np.multiply(dc,k),axis=1)
        yprimestar =np.copy(y)+ np.sum(np.multiply(cstar,k),axis=1)
        #delta0 = np.absolute(np.multiply(epsilon,yscale))
        #yerr = yprime - yprimestar
        errmax = maxfunc(yerr,yscale,yprime)
        errmax/=epsilon
        if (errmax>1):
            hnew = safetyfac*h*pow(errmax,pshrink)
            if(hnew<0.1*h):
                hnew=.1*h
            h=hnew
#           outlist=np.array([t,yprime[0],yprime[1],yprime[2],yprime[3],yprime[
#           for item in outlist:
#               fadapt.write("%s\t" % item)
```

```
#             fadapt.write("\n")
        else:
            if(errmax>errcon):
                hnew = safetyfac*h*pow(errmax,pgrow)
            else:
                hnew = 5.*h
#             outlist=np.array([t,yprime[0],yprime[1],yprime[2],yprime[3],yprime[
#             for item in outlist:
#                 fadapt.write("%s\t" % item)
#             fadapt.write("\n")
#             fadapt.close()
            return t+h,yprimestar,hnew
            #false break for testing
            #break
            #problem has something to do with break conditions
            #print("breaking")
    #tprime = t+h
    #print(h)
    tprime = t+h
#     fadapt.close()
    return tprime,yprimestar,hnew
#     return tprime,yprimestar,h

def linearMaxFunc(yerr,yscale,yprime):
    errmax = max(np.absolute(yerr/yscale))
    return errmax

def sphericalMaxFunc(yerr,yscale,x):
    rscale=yscale[1]/x[1]
    invst = 1./sin(x[2])
    errmax = max(abs(x[0]/yscale[0]),abs(x[1]/yscale[1]),abs(x[2]/yscale[2]*rsca
    return errmax



def ignoretMaxFunc(yerr,yscale,x):
    errmax = max(np.absolute(yerr[1:8]/yscale[1:8]))
    return errmax

def rk4(t,y,h,func,arg):
    k1=h*func(t,y,arg) #no t on right hand side of these equations
    k2 = h*func(t+0.5*h,y+0.5*k1,arg)
    k3 = h*func(t+0.5*h,y+0.5*k2,arg)
    k4 = h*func(t+h,y+k3,arg)
    t=t+h
    return t,np.copy(y)+k1/6.+k2/3.+k3/3.+k4/6.
```

```
def geodesic(lamb,x,Rs):
    #returns a vector of the four acceleration
    #declare some constants
    rmRs=x[1]-Rs
    ct=cos(x[2])
    st=sin(x[2])
    invrmRs = 1./rmRs
    invr = 1./x[1]
    temp1 = 0.5*Rs*invr
    x7sq = x[7]*x[7]
    x5invr=x[5]*invr
    #calculate dut
    dut = -Rs*x[4]*x[5]*invr*invrmRs
    #calculate dur
    cut = -temp1*rmRs*invr*invr
    cur = temp1*invrmRs
    cutheta = rmRs
    cuphi = rmRs*st*st
    #print("x=",x)
    dur =cut*x[4]*x[4]+cur*x[5]*x[5]+cutheta*x[6]*x[6]+cuphi*x7sq
    #calculate dutheta
    dutheta = -2.*x[6]*x5invr+ct*st*x7sq
    #calculate duphi
    duphi =-2.*(x5invr+ct/st*x[6])*x[7]
    rhs=np.array([x[4],x[5],x[6],x[7],dut, dur, dutheta, duphi])
    #print(cut,cur,cutheta,cuphi)
    return np.array([x[4],x[5],x[6],x[7],dut, dur, dutheta, duphi])

def integrateNullGeodesic(xpix, ypix, pixelheight,pixelwidth, skypixelheight,sky
    pixelcoord=np.array([xpix,ypix])
    coords = initialize(pixelcoord,Rplane,pixelheight,pixelwidth,skypixelwidth,s
    r=coords[1]
    lamb=0.
    color = 1
    n=0
    h=hinit
    phi=coords[3]
    while(r<=Router):
        yscale =np.absolute(coords)+np.absolute(h*geodesic(lamb,coords,Rs))+tiny
        lamb,coords,h=adaptiveRK4(lamb,coords,h,geodesic,linearMaxFunc,Rs,yscale
        r=coords[1]
        phi=coords[3]
        if (r<Rfac*Rs) and (h<heps):
            color = 0
            break
```

```
            n+=1
            if((n%10000)==0): print(n,r,coords[2],phi,h)
        if(coords[2]<0.):
            temp = (-coords[2])%(pi)
            coords[2]=pi-temp
        else:
            coords[2]%=pi

        if(coords[3]<0.):
            temp=(-coords[3])%(2.*pi)
            coords[3]=2.*pi-temp
        else:
            coords[3]%=(2.*pi)
        rmRs2 = coords[1]-Rs
        #testnull = -rmRs2/coords[1]*coords[4]*coords[4]+coords[5]*coords[5]*coords[
        #if(abs(testnull)>1.e-7): print(xpix,ypix,"Null test failed")
        telestart = (xpix+ypix*pixelwidth)*3
        xout = int(coords[3]*skypixelwidth/2./pi)
        yout = int(coords[2]*skypixelheight/pi)
        skystart = (xout+yout *skypixelwidth)*3
        return skystart,telestart,color

#parabola test
def parabola(t,y,ab):
    ab[0] = a
    ab[1] =b
    ynew=np.array([a*t+b,a*t+b])
    return ynew

def sinusoid(t,y,params):
    amp = params[0]
    omega = params[1]
    phase = params[2]
    ynew = amp*omega*cos(omega*t+phase)
    ynewarray = np.array([ynew,ynew])
    return ynewarray

def sho(t,y,omega): #y[0] is u, y[1] is x
    shorhs = np.array([-omega*omega*y[1], y[0]])
    return shorhs
#a=3.
#b=1.
#ab = np.array([a, b])
def test():
    amp = 1.0
    omega = 2.*pi/5.
```

```
    phase =0.
    tiny = 1.e-30
    params = np.array([amp, omega, phase])
    t=np.zeros(1000)
    y=np.zeros((len(t),2))
    h=1.e-2
    yn=np.zeros(2)
    yn[1]=1.0
    tn=0.0
    yscale =np.absolute(yn)+np.absolute(np.multiply(h,sho(tn,yn,omega)))+tiny
    epsilon=1.e-4
    for n in range(0,len(t)):
        yscale =np.absolute(yn)+np.absolute(h*sho(tn,yn,omega))+tiny
        #print(n,yscale)
        t[n]=tn
        y[n,:]=yn
        #tn,yn =rk4(tn,yn,h,sho,omega)
        #print(tn,yn,h," hello")
        tn,yn,h=adaptiveRK4(tn,yn,h,sho,linearMaxFunc,omega,yscale,epsilon)
        #print(tn,yn,h)
    pyplot.figure()
    pyplot.xlabel("Time")
#     tn,yn,h=adaptiveRK4(tn,yn,h,sho,omega,yscale,epsilon)

#pyplot.ylabel("Relative error from analytic solution")
    #pyplot.title("RK4 solution to dx/dt = omega*cos(omega*t)")
    #pyplot.ylim([-1.,1.])
#     pyplot.xlim([0.,10.])
#     pyplot.plot(t,y[:,1])
#     pyplot.plot(t,np.cos(omega*t))

    pyplot.plot(t, y[:,1]-np.cos(omega*t))

#pyplot.plot(t,(y[:,1]-0.5*a*t*t-b*t)/(0.5*a*t*t+b*t))
    pyplot.show()

    #tested by comparison to Mathematica using "testrule"
    #x=np.array([1.27,1.86,.3,.2])
    #Rs=.3
    #print gammat(x,Rs)
    #print gammar(x,Rs)
    #print gammatheta(x,Rs)
    #print gammaphi(x,Rs)
    return 0

#sky is 4096 by 2048
```

```
#skypixelheight = 2048
#skypixelwidth = 4096

def main():
    #image plane's center is at Rplane<Router (radius of outer shell)
    #h=1.e-3
    hinit=1.e-1
    #h=1.e-4
    Router = 1000.
    Rplane = 700.
    Rs = 2.
    pixelwidth = 51
    pixelheight = 51
    every = 1
    deltalamb = 1.e-1
    #epsilon = 1.e-6
    #yscale = [500.,500.,pi,2.*pi,-1.,1.,1.,1.]
    imagewidth = 50;
    imageheight = 50;
    tiny = 1.e-30
    epsilon=1.e-8
    eccentricity = 0.2
    Rfac = 1.+1.e-10
    heps = 1.e-14
    semilatusr = 10.0    #affine = np.zeros(20000)
    fsky = open("skymap.png","r")
    reader = png.Reader(fsky)
    skypixelwidth, skypixelheight, skypixels, metadata=reader.read_flat()
    telepixels = np.zeros((pixelwidth*pixelheight*3),dtype=np.uint8)

    for ypix in range(1,pixelheight,every):
        for xpix in range(1,pixelwidth,every):
            skystart,telestart,color=integrateNullGeodesic(xpix, ypix, pixelheigl
            if(color==1):
                #skytemp = skypixels[xout,yout]
                #for pix in range(3):
                    #telepixels[telestart+pix]= skytemp[pix]
                telepixels[telestart:telestart+3]=skypixels[skystart:skystart+3]
            else:
                telepixels[telestart]=255 #leave other two indices zero

    ftele = open("teleview.png", "w")
    telewrite=png.Writer(width=pixelwidth,height=pixelheight,greyscale=False,alp
    telewrite.write_array(ftele,telepixels)
    ftele.close()
    fsky.close()
```

```
    #scipy.imsave?
#     pyplot.figure()
#     pyplot.plot(affine,xout[:,0])
#     pyplot.show()
#
#     pyplot.figure()
#     pyplot.plot(affine,xout[:,1])
#     pyplot.show()

#     pyplot.figure()
#     pyplot.plot(affine,xout[:,2])
#     pyplot.show()

#     pyplot.figure()
#     pyplot.plot(affine,xout[:,3])
#     pyplot.show()

#     pyplot.figure()
#     pyplot.plot(affine,hs)
#     pyplot.show()

main()
#test()
```

15 Appendix: Parallel code

15.1 Python code

```
import numpy as np
from math import pi,sin,cos,sqrt,atan,acos
#from scipy import misc
from sys import argv
from mpi4py import MPI
from time import time, ctime

a=np.array([0, .2, .3, .6, 1., 7./8.])
b=np.array([[0.,0.,0.,0.,0.],[.2,0.,0.,0.,0.],[3./40.,9./40.,0.,0.,0.],[3./10.,
c = np.array([37./378., 0., 250./621., 125./594., 0., 512./1771.])
cstar = np.array([2825./27648., 0., 18575./48384., 13525./55296., 277./14336., 0
dc = np.array([277./64512.,0.,-6925./370944.,6925./202752.,277./14336.,-277./708
lena=len(a)



# t is lamb, affine parameter
# y is u, four velocity, or x, four position
```

```
def initialize(pixelcoord,Rplane,pixelheight,pixelwidth,skypixelwidth,skypixelhe
    #set origin of pixel plane at x axis
    t = 0.
    x = Rplane
    y = (pixelcoord[0]-(pixelwidth-1)/2.)*imagewidth/float(pixelwidth)
    z = (pixelcoord[1]-(pixelheight-1)/2.)*imageheight/float(pixelheight)
    r= sqrt(x*x+y*y+z*z)
    phi = atan(y/x)
    theta = acos(z/r)

    #initial u perpendicular to plane.
    #magnitude of u is arbitrary— affine parameter makes it rescalable
    #(ut)^2-(uy)^2-(ux)^2-(uz)^2=0 so ut = +—ux
    #for x decreasing as t increases, ut = -ux (inward)
    uy = 0.
    uz = 0.
    ux = 1.


    invr = 1./r
    invrsq = invr*invr
    ur = -x/r
    utheta = x*z*invr*invrsq*sqrt(1.-z*z*invrsq)
    uphi = -y/(x*x+y*y)
    rmRs = r-Rs
    st = sin(theta)
    ut = sqrt((ur*ur*r/rmRs+r*r*utheta*utheta+r*r*st*st*uphi*uphi)/rmRs*r)
    initcoords =np.array([t, r, theta, phi, ut, ur, utheta, uphi])
    return initcoords

def initializeElliptical(eccentricity,semilatusr,Rs):
    r2 = semilatusr*0.5*Rs/(1.-eccentricity)
    print(eccentricity, semilatusr, Rs,r2)
    theta = pi/2.
    phi = 0.
    t = 0.
    utheta = 0.
    temp = 1./(semilatusr - 3. - eccentricity*eccentricity)
    angularL = 0.5*semilatusr*Rs*sqrt(temp)
    energy=sqrt((semilatusr-2.-2.*eccentricity)*(semilatusr-2.+2.*eccentricity)/
    uphi = angularL/r2/r2
    ur = 0.
    ut = energy/(1.-Rs/r2)
    return np.array([t,r2,theta,phi,ut,ur,utheta,uphi])
```

```
def adaptiveRK4(t,y,h,func,maxfunc,arg,yscale,epsilon):
    leny=len(y)
    safetyfac = 0.9
    pgrow =-0.20
    pshrink =-0.25
    errcon = 1.89e-4 #see NR in Fortran
    hnew=h/2.
    nsteps=0
    #hlast = h
    while True:
        # j and i are reversed from Numerical Recipes book (page 711)
        #loop over y indices
        k=np.zeros((leny,lena))
        tprimearg=t
        yprime = np.copy(y)
        yprimestar = np.copy(y)
        for j in range(0,len(a)): #for all terms summed in method
            tprimearg = t+a[j]*h
            yprimearg = np.copy(y)
            for n in range(0,leny): #over all variables in y vector
                for i in range(0,j): #over all indices of k
                    #update for next term of k in calculation
                    yprimearg[n]+=b[j,i]*k[n,i]
            k[:,j]=h*func(tprimearg,yprimearg,arg)
        yprime = y+np.sum(np.multiply(c,k),axis=1)
        yerr =  np.sum(np.multiply(dc,k),axis=1)
        yprimestar =np.copy(y)+ np.sum(np.multiply(cstar,k),axis=1)
        errmax = maxfunc(yerr,yscale,yprime)
        errmax/=epsilon
        if (errmax>1):
            hnew = safetyfac*h*pow(errmax,pshrink)
            if(hnew<0.1*h):
                hnew=.1*h
            h=hnew
            nsteps+=1
        else:
            if(errmax>errcon):
                hnew = safetyfac*h*pow(errmax,pgrow)
            else:
                hnew = 5.*h
            nsteps+=1
            return nsteps,t+h,yprimestar,hnew
    tprime = t+h
    return tprime,yprimestar,hnew
```

```
def linearMaxFunc(yerr,yscale,yprime):
    errmax = max(np.absolute(yerr/yscale))
    return errmax



def geodesic(lamb,x,Rs):
    #returns a vector of the four acceleration
    #declare some constants
    rmRs=x[1]-Rs
    ct=cos(x[2])
    st=sin(x[2])
    invrmRs = 1./rmRs
    invr = 1./x[1]
    temp1 = 0.5*Rs*invr
    x7sq = x[7]*x[7]
    x5invr=x[5]*invr
    #calculate dut
    dut = -Rs*x[4]*x[5]*invr*invrmRs
    #calculate dur
    cut = -temp1*rmRs*invr*invr
    cur = temp1*invrmRs
    cutheta = rmRs
    cuphi = rmRs*st*st
    #print("x=",x)
    dur =cut*x[4]*x[4]+cur*x[5]*x[5]+cutheta*x[6]*x[6]+cuphi*x7sq
    #calculate dutheta
    dutheta = -2.*x[6]*x5invr+ct*st*x7sq
    #calculate duphi
    duphi =-2.*(x5invr+ct/st*x[6])*x[7]
    rhs=np.array([x[4],x[5],x[6],x[7],dut, dur, dutheta, duphi])
    #print(cut,cur,cutheta,cuphi)
    return np.array([x[4],x[5],x[6],x[7],dut, dur, dutheta, duphi])

def integrateNullGeodesic(xpix, ypix, pixelheight,pixelwidth, skypixelheight,sky
    pixelcoord=np.array([xpix,ypix])
    coords = initialize(pixelcoord,Rplane,pixelheight,pixelwidth,skypixelwidth,s
    r=coords[1]
    lamb=0.
    color = 1
    n=0
    h=hinit
    phi=coords[3]
    totnsteps=0
    while(r<=Router):
        yscale =np.absolute(coords)+np.absolute(h*geodesic(lamb,coords,Rs))+tiny
```

```
            nsteps,lamb,coords,h=adaptiveRK4(lamb,coords,h,geodesic,linearMaxFunc,Rs
            r=coords[1]
            phi=coords[3]
            totnsteps+=nsteps
            if (r<Rfac*Rs) and (h<heps):
                color = 0
                break
            n+=1
            if((n%10000)==0): print(n,r,coords[2],phi,h)
        if(coords[2]<0.):
            temp = (-coords[2])%(pi)
            coords[2]=pi-temp
        else:
            coords[2]%=pi

        if(coords[3]<0.):
            temp=(-coords[3])%(2.*pi)
            coords[3]=2.*pi-temp
        else:
            coords[3]%=(2.*pi)
        rmRs2 = coords[1]-Rs
        telestart = (xpix+ypix*pixelwidth)*3
        xout = int(coords[3]*skypixelwidth/2./pi)
        yout = int(coords[2]*skypixelheight/pi)
        skystart = (xout+yout *skypixelwidth)*3
        return totnsteps,skystart,telestart,color


def main():

    comm = MPI.COMM_WORLD
    id= comm.Get_rank()
    wsize= comm.Get_size()
    tstart = MPI.Wtime()
    #fsky = open("skymap.png","r")
    #reader = Reader(fsky)
    #skypixelwidth, skypixelheight, skypixels, metadata=reader.read_flat()
    skypixelwidth = 4096
    skypixelheight = 2048
    skypixels = np.zeros((skypixelwidth*skypixelheight*3),dtype=np.uint8)
    pixelwidth = int(argv[1])
    pixelheight = int(argv[2])
    tskymapstart = MPI.Wtime()
    telepixels = np.zeros((pixelwidth*pixelheight*3),dtype=np.uint8)
    colorpixels = np.zeros((pixelwidth*pixelheight),dtype=np.uint8)
    skystartall = np.zeros((pixelwidth*pixelheight),dtype=np.uint32)
```

```
telestartall = np.zeros((pixelwidth*pixelheight),dtype=np.uint32)
colorall = np.zeros((pixelwidth*pixelheight),dtype=np.uint8)
totnstepsall=np.zeros((wsize),dtype=np.uint32)
tskymapend = MPI.Wtime()
tskymap = tskymapend-tskymapstart

tmin = 1.e6
tpercparmin=1.e6
hinit=1.e-1
#h=1.e-4
Router = 1000.
Rplane = 700.
Rs = 2.
every = 1
deltalamb = 1.e-1
imagewidth = 50.;
imageheight = 50.;
tiny = 1.e-30
epsilon=1.e-8
eccentricity = 0.2
Rfac = 1.+1.e-10
heps = 1.e-14
semilatusr = 10.0


tstartpp=MPI.Wtime() #percent parallelized
numperprocess = pixelheight*pixelwidth/wsize
skystart=np.zeros((numperprocess),dtype=np.int32)
telestart=np.zeros((numperprocess),dtype=np.int32)
color = np.zeros((numperprocess),dtype=np.int8)
totnsteps=np.zeros((numperprocess),dtype=np.int32)
trk4all=np.zeros((numperprocess),dtype=np.float)
ttelestop = MPI.Wtime()
ttele = ttelestop-tstartpp
trk4=float("inf")
for index in range(numperprocess):
    ypix = int((id*numperprocess+index)/pixelwidth)
    xpix = (id*numperprocess+index)%pixelwidth
    tstartrk4=MPI.Wtime()
    totnsteps[index],skystart[index],telestart[index],color[index]=integratel
    tendrk4=MPI.Wtime()
    trk4=min(trk4,(tendrk4-tstartrk4)/float(totnsteps[index]))
totnstepsmax=max(totnsteps)
tstoppp = MPI.Wtime()
tpercpar=tstoppp-tstartpp
```

```
    comm.Barrier()
    if id==0:
        totnstepsmaxall=0
    else:
        totnstepsmaxall=None
    comm.Barrier()

    totnstepsmaxall=comm.reduce(totnstepsmax,op=MPI.MAX,root=0)
    tskymapall = comm.reduce(tskymap, op=MPI.MAX, root=0)
    tteleall = comm.reduce(ttele,op=MPI.MAX,root=0)
    comm.Gatherv(skystart,skystartall,root=0)
    comm.Gatherv(telestart, telestartall, root=0)
    comm.Gatherv(color,colorall, root=0)
    trk4min=comm.reduce(trk4,op=MPI.MIN,root=0)
    comm.Barrier()
    tend = MPI.Wtime()
    tall = tend-tstart
    if id==0:
        tindexstart = MPI.Wtime()
        for index in range(pixelheight*pixelwidth):

            if(colorall[index]==1):
                telepixels[telestartall[index]:telestartall[index]+3]=skypixels[
            else:
                telepixels[telestartall[index]]=255 #leave other two indices zer
        tindexend = MPI.Wtime()
        tindex = tindexend-tindexstart
    #if id==0:
        #twritestart = MPI.Wtime()
        #ftele = open('teleview_{pw}_{ph}_{ws}.png'.format(pw=pixelwidth,ph=pixe
        #telewrite=Writer(width=pixelwidth,height=pixelheight,greyscale=False,al
        #telewrite.write_array(ftele,telepixels)
        #ftele.close()
        #twriteend=MPI.Wtime()
        #twrite = twriteend-twritestart
    #fsky.close()
    comm.Barrier()
    tmax = comm.reduce(tall,MPI.MAX,root=0)
    tpercparmin = comm.reduce(tpercpar/tall,op=MPI.MIN,root=0)
    comm.Barrier()
    if (id==0):
#       print("Telescope dimensions in M", 2.*imagewidth, 2.*imageheight)
#       print("Telescope resolution", pixelwidth, pixelheight)
#       print("Skymap resolution", skypixelwidth, skypixelheight)
#       print("Schwarzschild radius in M", 2.*Rs)
#       print("Outer radius in M", 2.*Router)
```

```
#         print("Telescope radius in M", 2.*Rplane)
#         print("Number of processes = ",wsize)
#         print("Maximum number of integration steps taken is",totnstepsmaxall)
#         print("The time for a single step of the RK4 is",trk4min)
#         print("Total runtime = ",tmax)
#         print("Fraction parallel = ", tpercparmin)
          print pixelwidth,pixelheight,wsize,totnstepsmaxall,trk4min,tmax,tpercparr

    MPI.Finalize()
main()
```

15.2 Strong scaling batch file

```
#!/bin/bash
### set the number of nodes
### set the number of PEs per node
#PBS -l nodes=64:ppn=16:xe
### set the wallclock time
#PBS -l walltime=1:00:00
### set the job name
#PBS -N SDstrong
### set the job stdout and stderr
#PBS -e $PBS_JOBID.err
#PBS -o $PBS_JOBID.out
### set email notification
#PBS -m bea
#PBS -M sdorsh1@lsu.edu
### In case of multiple allocations, select which one to charge
#PBS -A babq
### Set umask so users in my group can read job stdout and stderr files
#PBS -W umask=0027
###export MPICH_NEMESIS_ASYNC_PROGRESS=1
###export MPICH_MAX_THREAD_SAFETY=multiple
module load bwpy
module load bwpy-mpi
cd $PBS_O_WORKDIR
aprun -n 1 python geodesic19.py 32 32  > Strongscaling/out32_32_1.$PBS_JOBID
aprun -n 4 python geodesic19.py 32 32  > Strongscaling/out32_32_4.$PBS_JOBID
aprun -n 16 python geodesic19.py 32 32  > Strongscaling/out32_32_16.$PBS_JOBID
aprun -n 64 python geodesic19.py 32 32  > Strongscaling/out32_32_64.$PBS_JOBID
aprun -n 256 python geodesic19.py 32 32  > Strongscaling/out32_32_256.$PBS_JOBID
aprun -n 1024 python geodesic19.py 32 32  > Strongscaling/out32_32_1024.$PBS_JOB
###aprun -n 128 ./mmvmpi -N 16 16384 > out16384_128.$PBS_JOBID
###aprun -n 128 ./mmvmpi -N 16 102400 > out102400_128.$PBS_JOBID
```

15.3 Weak scaling batch file

```
#!/bin/bash
### set the number of nodes
### set the number of PEs per node
#PBS -l nodes=64:ppn=16:xe
### set the wallclock time
#PBS -l walltime=1:00:00
### set the job name
#PBS -N SDweak
### set the job stdout and stderr
#PBS -e $PBS_JOBID.err
#PBS -o $PBS_JOBID.out
### set email notification
#PBS -m bea
#PBS -M sdorsh1@lsu.edu
### In case of multiple allocations, select which one to charge
#PBS -A babq
### Set umask so users in my group can read job stdout and stderr files
#PBS -W umask=0027
###export MPICH_NEMESIS_ASYNC_PROGRESS=1
###export MPICH_MAX_THREAD_SAFETY=multiple
module load bwpy
module load bwpy-mpi
cd $PBS_O_WORKDIR
aprun -n 1 python geodesic19.py 1 1  > Weakscaling/out1_1_1.$PBS_JOBID
aprun -n 4 python geodesic19.py 2 2  > Weakscaling/out2_2_4.$PBS_JOBID
aprun -n 16 python geodesic19.py 4 4  > Weakscaling/out4_4_16.$PBS_JOBID
aprun -n 64 python geodesic19.py 8 8  > Weakscaling/out8_8_64.$PBS_JOBID
aprun -n 256 python geodesic19.py 16 16  > Weakscaling/out16_16_256.$PBS_JOBID
aprun -n 1024 python geodesic19.py 32 32  > Weakscaling/out32_32_1024.$PBS_JOBID
###aprun -n 128 ./mmvmpi -N 16 16384 > out16384_128.$PBS_JOBID
###aprun -n 128 ./mmvmpi -N 16 102400 > out102400_128.$PBS_JOBID
```