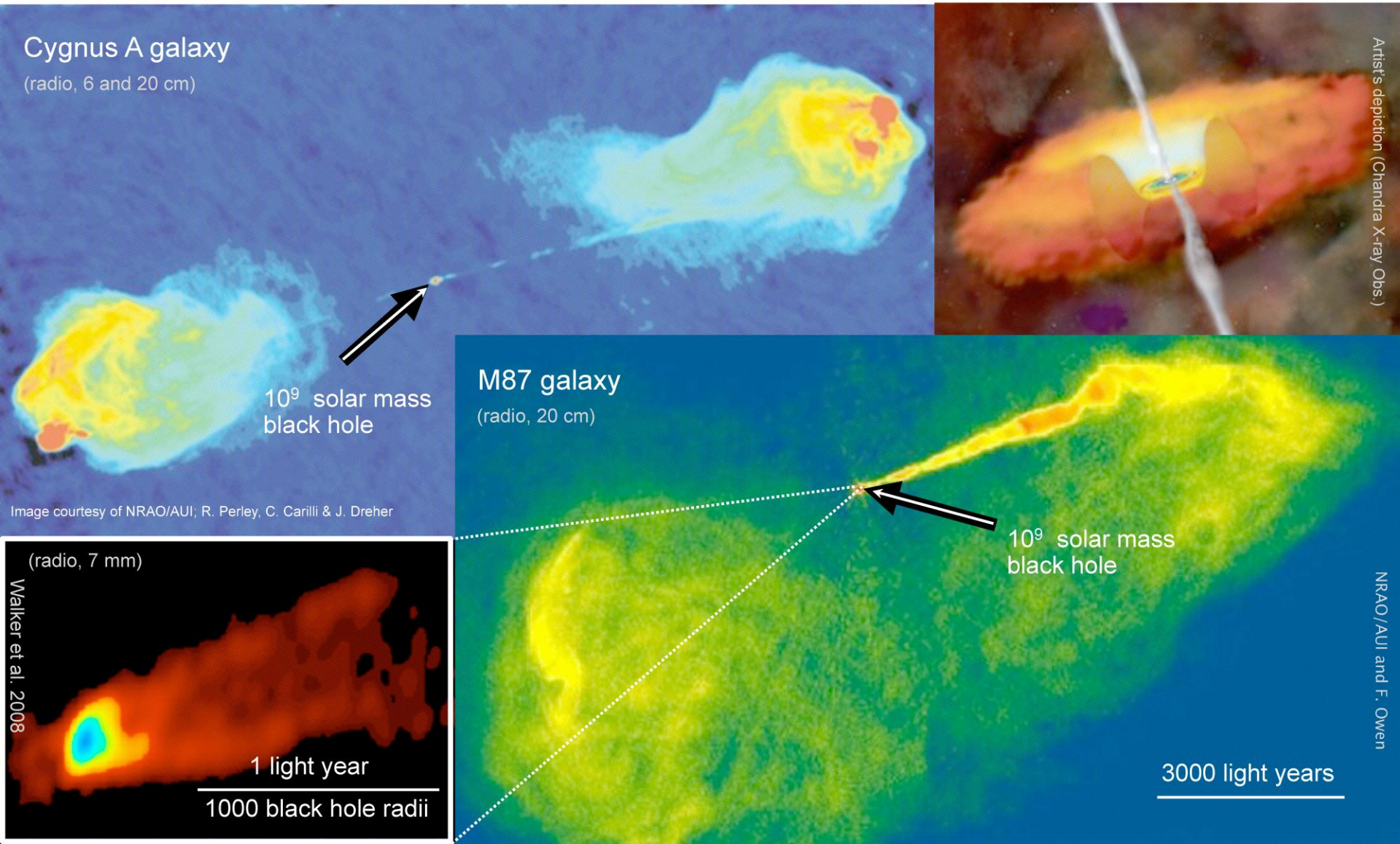
# Acceleration of general relativistic MHD simulation algorithms using GPU and OpenACC

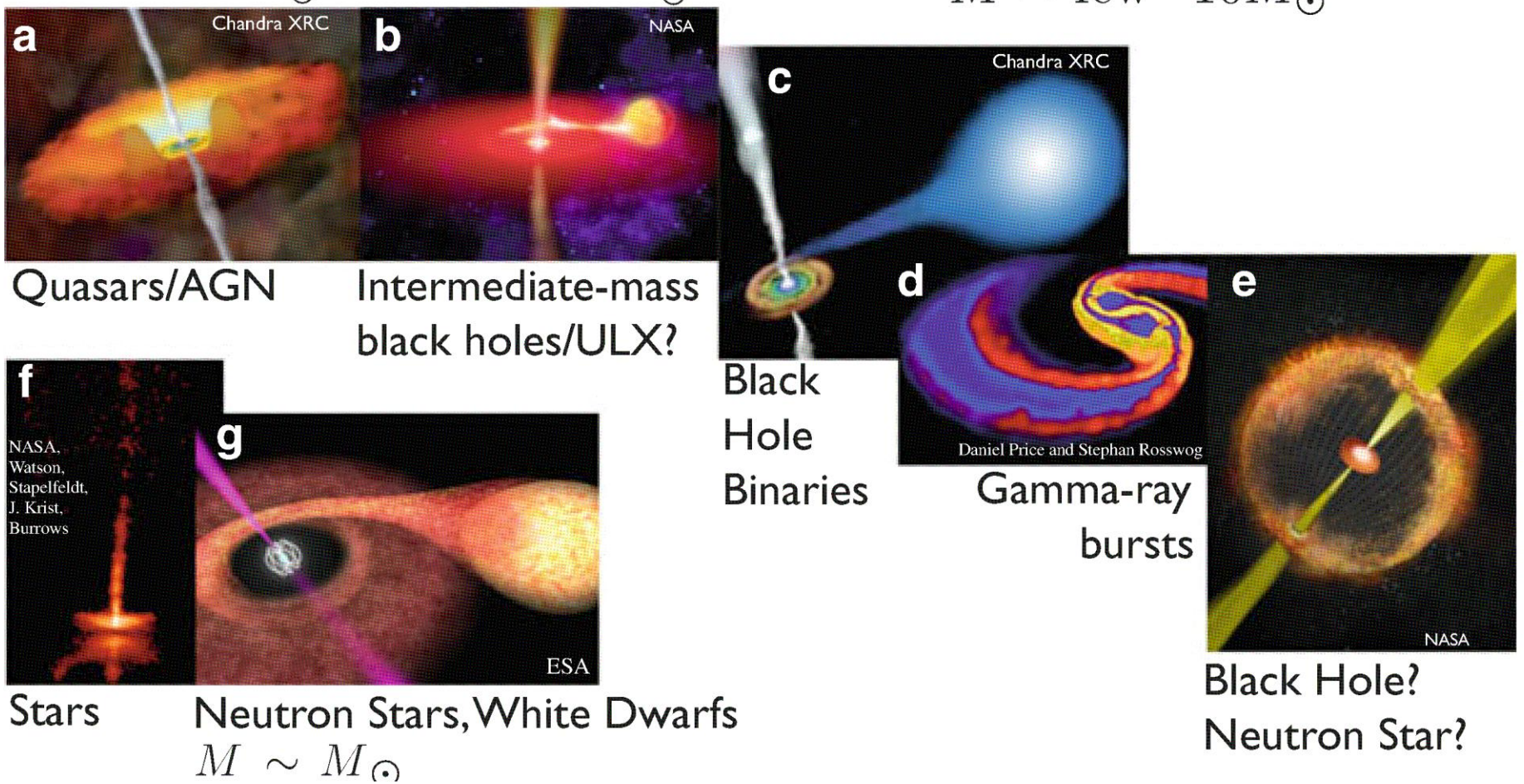
Lev Barash[1], Alexander Tchekhovskoy[2]

[1] ИТФ им. Л.Д. Ландау
[2] UC Berkeley

**Таруса, ИКИ РАН, 18 ноября 2015 года**

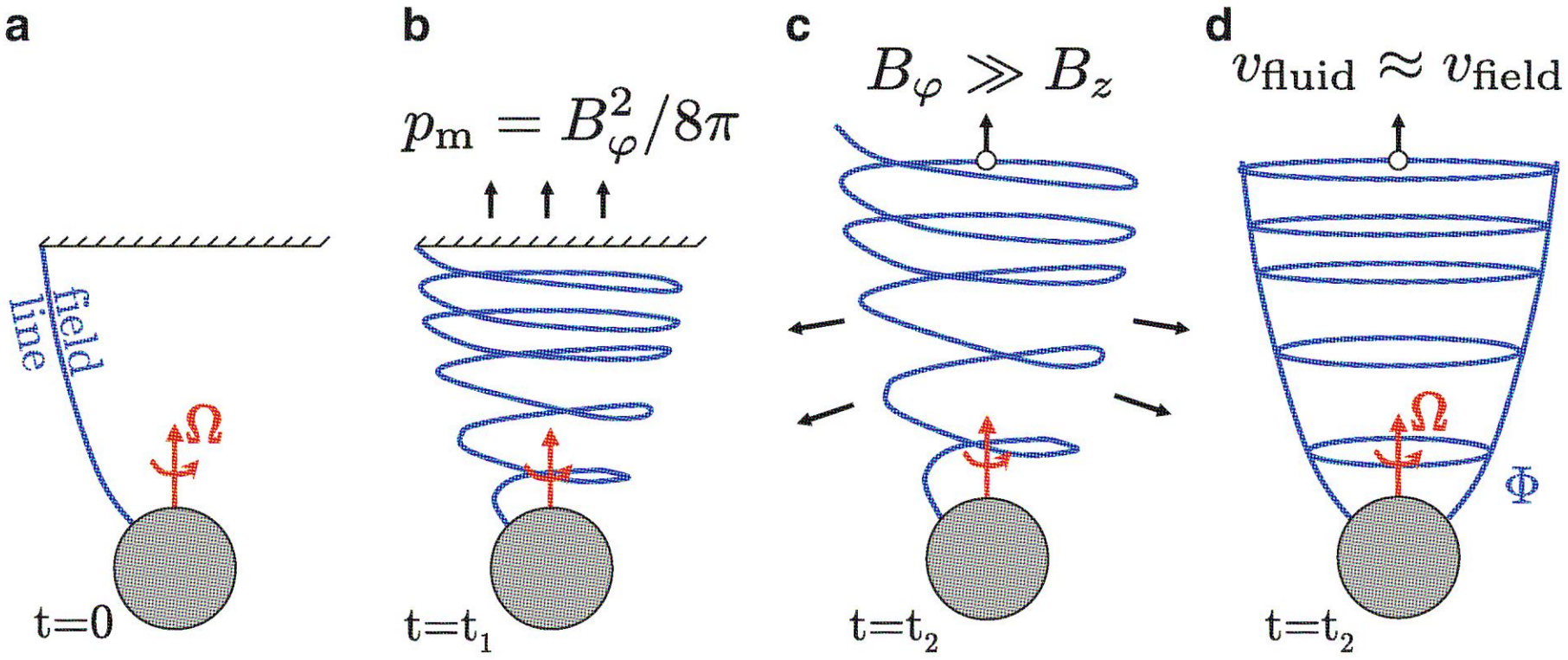
# Jets: Beautiful & Challenging






Black holes of all sizes produce jets. BHs come in two broad categories: supermassive, with masses ranging between millions and billions of solar masses, and stellar-mass BHs, with masses ranging from a few to tens of solar masses. Supermassive BHs are found at the centers of AGN (panel (**a**)), and stellar-mass BHs are found in binary systems (panel (**c**)), or formed as a result of binary neutron star mergers (panel (**d**)) and core collapse of massive stars that is thought to give rise to GRBs (panel (**e**)).

# Illustration of jet formation by magnetic fields



## Introduction

Recent years have seen great progress in understanding astrophysical systems, their structure and dynamics. Examples of such systems include active galaxies that contain accreting black holes at their centers that sometimes produce relativistic jets. Because such systems are highly nonlinear and involve dynamically important magnetic fields, they are notoriously difficult to describe analytically. Therefore, large-scale 3D numerical simulations have been instrumental at advancing our understanding of such systems. In this work, we describe an improvement to a numerical algorithm implemented in a massively parallel code, which solves time-dependent general relativistic magnetized fluid dynamics in 3D on a space-time of a spinning black hole. This code is based on an open-source 2D serial code HARM2D (High Accuracy Relativistic Magnetohydrodynamics, Gammie et al., ApJ 579, p. 444-457, 2003). Using a code of this type, important results have been obtained in the context of black hole accretion and jets. However, the cost of such simulations is enormous: each of them took about 1 million core-hours on modern CPUs. To make further progress in our understanding of such systems requires substantial increase in the amount of available computational resources, which, however, are scarce.

# The algorithm

- shock-capturing Godunov-based scheme
- evolves the GRMHD equations of motion in a conservative form

For instance, for the 1D case, the equations take a simple vector form:

$$\frac{\partial \mathbf{U}(\mathbf{p})}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{p})}{\partial x}, \tag{1}$$

where $\mathbf{U}$ is the vector of "conserved" quantities, e.g., energy density, momentum density, particle number density, $\mathbf{F}$ is the corresponding vector of fluxes, and $\mathbf{p}$ is the vector of primitive quantities, e.g., fluid pressure, velocity. To evolve this vector equation, at each time step the right-hand side is evaluated, and this is used to compute the value of the conserved quantity $\mathbf{U}$ at the new time.

# The algorithm

The fundamental GRMHD equations as used in the simulation are the particle number conservation equation; the four energy momentum equations, written in a coordinate basis and using the MHD stress energy tensor; and the induction equation, subject to the no-monopoles constraint.

Vector of conserved variables: $\boldsymbol{U} \equiv \sqrt{-g}(\rho u^t, T^t_t, T^t_i, B^i)$.

Primitive variables: $\boldsymbol{P} = (\rho, u, v^i, B^i)$.

Here $\rho$ is rest-mass density, $B^i$ is magnetic field density, $v^i$ is the 3-velocity, $u$ is internal energy density, $u^i$ is 4-velocity, $T$ is the energy-momentum tensor.

# Advantages of the algorithm

It is one of the premier GRMHD codes in use today and has already led to several important results on black hole accretion disks and relativistic jets.

- the code uses Kerr-Schild coordinates which allow the inner computational grid boundary to be inside the black hole horizon. As a result, the inner boundary is causally disconnected from the accretion flow and jet, so that the black hole is properly treated as an event horizon.
- Unlike other codes, it can be used with arbitrary coordinates.
- The code is energy-conserving to machine precision. Energy conserving schemes correctly capture (as gas internal energy) whatever magnetic field energy is lost via reconnection or turbulent energy is lost at the grid scale. While all codes generate a finite numerical (truncation) error, non-conservative codes can generate or lose arbitrary amounts of energy when making such errors. In contrast, when a conservative code makes a truncation error, the error is limited by the requirement to conserve energy – a manifestly physical requirement.
- the code has been especially designed to accurately handle strongly magnetized regions. This allows us to simulate highly magnetized jets, as in gamma-ray bursts.

# The algorithm

`step_ch()` – handles the sequence of making the time step, the fixup of unphysical values, and the setting of boundary conditions; also sets the dynamically changing time step size;

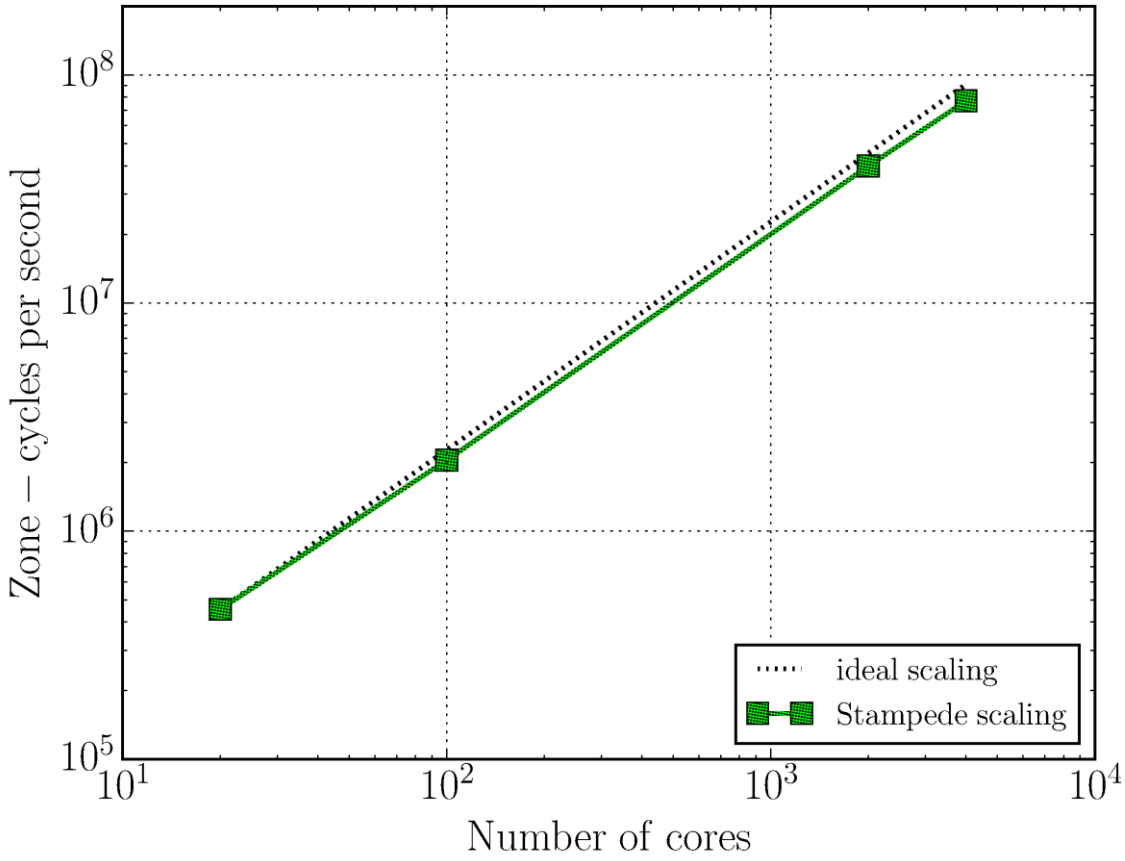
`advance()` – responsible for what happens during a time step update, including the flux calculation, the constrained transport calculation, the finite difference form of the time integral, and the calculation of the primitive variables from the updated conserved variables; also handles the `fix_flux()` call that sets the boundary condition on the fluxes. `advance()` is called from `step_ch()`.

`fluxcalc()` – sets the numerical fluxes, evaluated at the cell boundaries using the slope limiter. Also performs the flux-averaging used to preserve the $\boldsymbol{\nabla}\cdot\boldsymbol{B} = 0$ constraint. `fluxcalc()` is called from `advance()`.

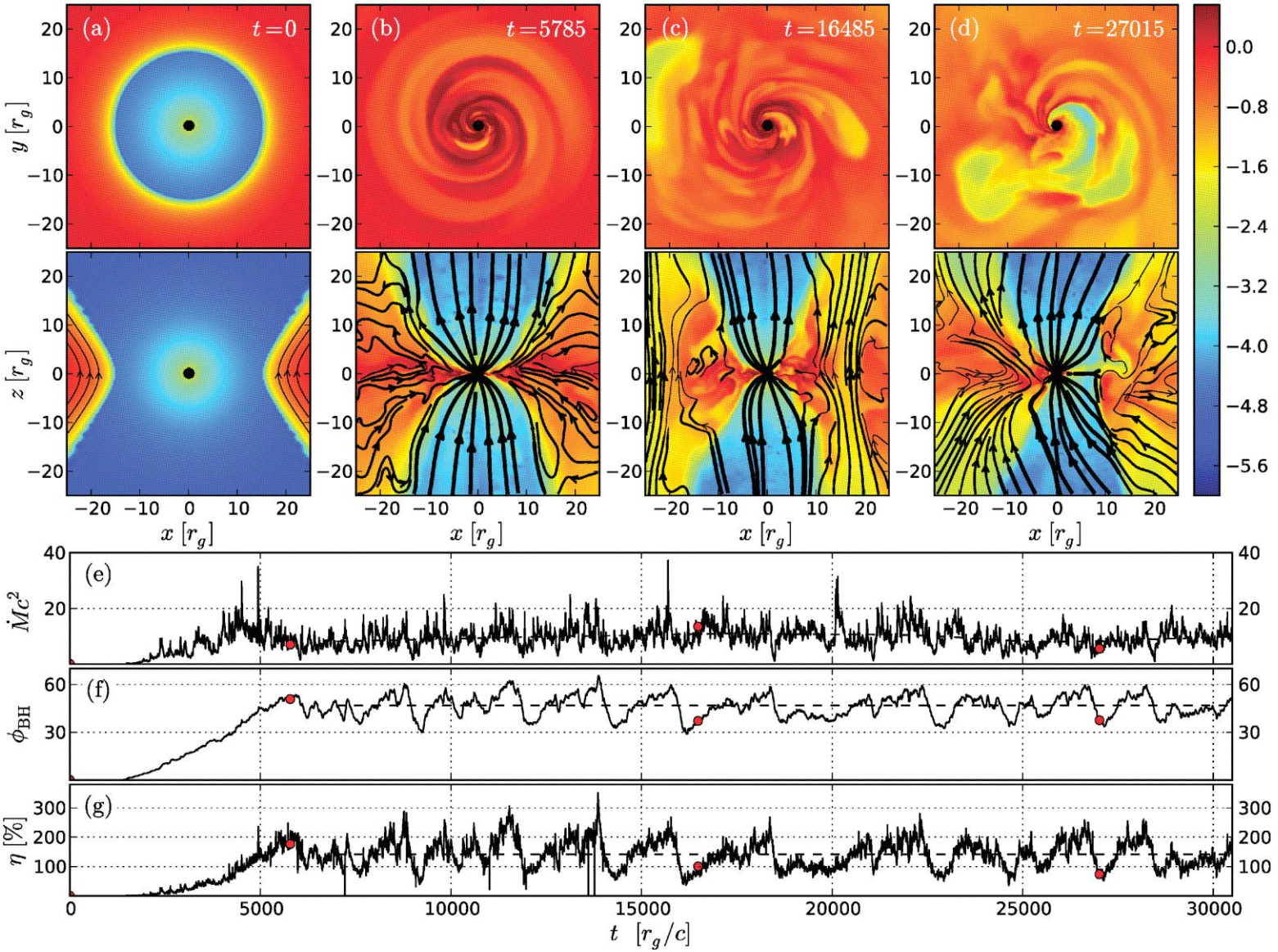
`eVol()` – evolves the electron entropy and updates it due to heating and electron conduction `eVol()` is called from `step_ch()`.

# MPI code performance and scaling

Fig.3: The `HARM` code shows excellent scaling on the `Stampede` supercomputer. The plot shows weak scaling for a tile size of $16^3$ cells when compiled using an Intel compiler with aggressive optimization options. The speed of the code on a single core is about $20,000$ zone-cycles per second.




# Simulation of Black Hole Disk-Jet Connection




видео

## Employing GPU capabilities

Here, we adopt an alternative approach: leveraging the power of Graphical Processing Units, or GPUs. Essentially high-end graphics cards allow to carry out enormous amount of computing and are a highly competitive to CPUs. We have used the OpenACC programming approach. It allows, with minor modifications of a CPU-based source code, to obtain a highly portable code which can offload the computations to a GPU. As we explain below, this gives us a factor of 10 speedup compared to the CPU version, allowing a dramatic increase in the computing abilities.

## OpenACC directive-based programming model

OpenACC is a directive-based high level programming model targeting a CPU + accelerator system, intentionally similar in many ways to OpenMP. It is designed to do for accelerator programming systems what OpenMP does for multicore systems. It hides or virtualizes those features of the system that can be managed automatically by the system without performance penalty, and exposes those features that must be managed by the programmer.

**Advantages:** OpenACC has demonstrated support for multiple devices, and there is some initial evidence for performance portability across device types. The same code can be successfully compiled and used for different platforms, on pure CPUs or using accelerators of different types, etc.

**Disadvantages:** OpenACC is still relatively young. There are as yet no available open-source implementations of the full language. Some critical features are still under development, and there are some differences in how features are implemented by different compilers, making portability across vendors an issue. There are still some limitations. For example, using global arrays are not supported in the device code for PGI OpenACC, so pointers of global arrays should be transferred through function parameters.

# Categories of OpenACC APIs

- Accelerator Parallel Region / Kernels Directives
- Loop Directives
- Data Declaration Directives
- Data Regions Directives
- Cache directives
- Wait / update directives
- Runtime Library Routines
- Environment variables

# OpenACC Kernels Directive

- Defines a region of a program that is to be compiled into a sequence of kernels for execution on the accelerator
- Each loop nest will be a different kernel
- Kernels launched in order in device
- Specified by:
  - #pragma acc kernels [clause [,clause]…] new-line

    *structured block*

# OpenACC Loop Directive

- Used to describe what type of parallelism to use to execute the loop in the accelerator.
- Can be used to declare loop-private variables, arrays and reduction operations.
- Specified by:
  - #pragma acc loop [clause [,clause]…] new-line

    *for loop*

# OpenACC Update Directive

- Used within a data region to update / synchronize the values of the arrays on both the host or accelerator
- Specified by:

  **#pragma acc update [clause [,clause]…] new-line**

- The clauses for the *!$acc update* directive are:
  - **host (list)**
  - **device (list)**
  - **if (condition)**
  - **async [( scalar-integer-expression)]**

**OpenACC directives which were most commonly used by us:**

```
#pragma acc routine
#pragma acc kernels present(var1,var2,...)
#pragma acc loop gang vector collapse(3) reduction(+:myvar) independent
#pragma acc enter data create(var1,var2,...)
#pragma acc exit data delete(var1,var2,...)
#pragma acc update host(var1,var2,...)
#pragma acc update device(var1,var2,...)
```

# OpenACC performance and speed-up

Time in seconds needed for the first 40 steps
for the lattice $512 \times 512 \times 1$.
CPU: Intel Xeon E5-2650v3, GPU: nVidia Tesla K40.

| | | time(all) | time(advance) | time(fluxcalc) |
|---|---|---|---|---|
| gcc | -O3: | 222.58 | 50.72 | 30.53 |
| pgcc -ta=host | -O3: | 196.27 | 51.54 | 28.87 |
| pgcc -ta=tesla | -O3: | 24.73 | 9.87 | 6.58 |