dendro-GR

Enabling Adaptivity & Parallelism for Computational Relativity

Hari Sundar, Milinda Fernando
School of Computing, University of Utah

David Neilsen, Eric Hirschmann, Hyun Lim
Brigham Young University

Computational Challenges in Gravitational Wave Astronomy, IPAM 2019

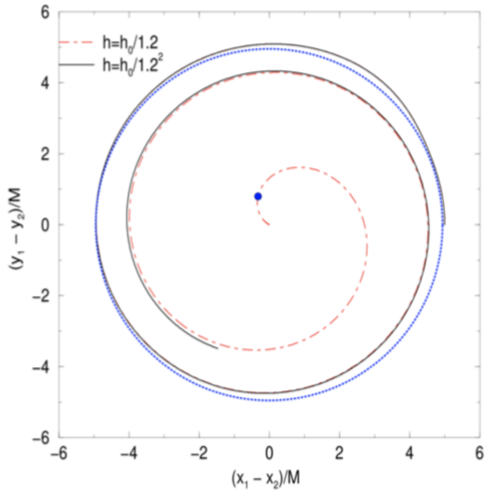
Introduction

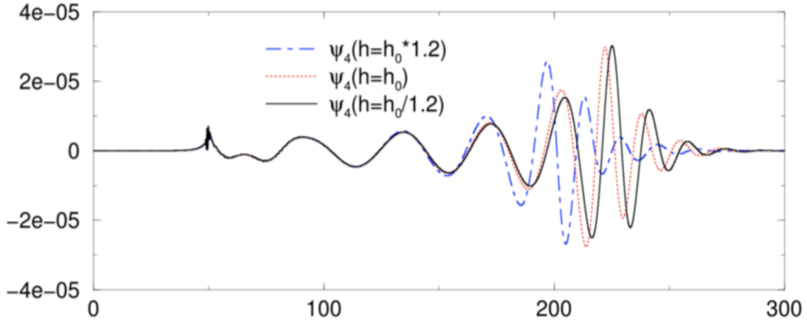
- Dendro-GR: New framework for Computational relativity
  - High-degree of spatial adaptivity
  - High levels of parallelism
- Intermediate Mass Ratio Inspirals (IMRIs)
- Wavelet Adaptive Multi-Resolution (WAMR)
- First BBH Evolutions

IMBHs and IMRIs

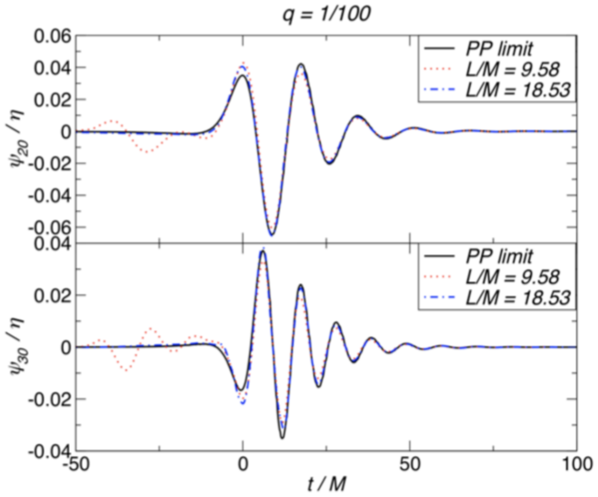
- Binaries with intermediate mass ratios $10 \lesssim q \lesssim 100$
- IMBHs
  - Collapse of Pop III stars
  - Mergers in stellar clusters
  - Accretion onto stellar mass BHs
  - Collapse of gas clouds in the early universe
- LIGO has detected remnants with masses $\sim 20 - 60\ M_{\odot}$
- Computational Challenge: Resolution

Mass ratio $q = 100$





- Lousto & Zlochower, PRL **106** 041101 (2011)

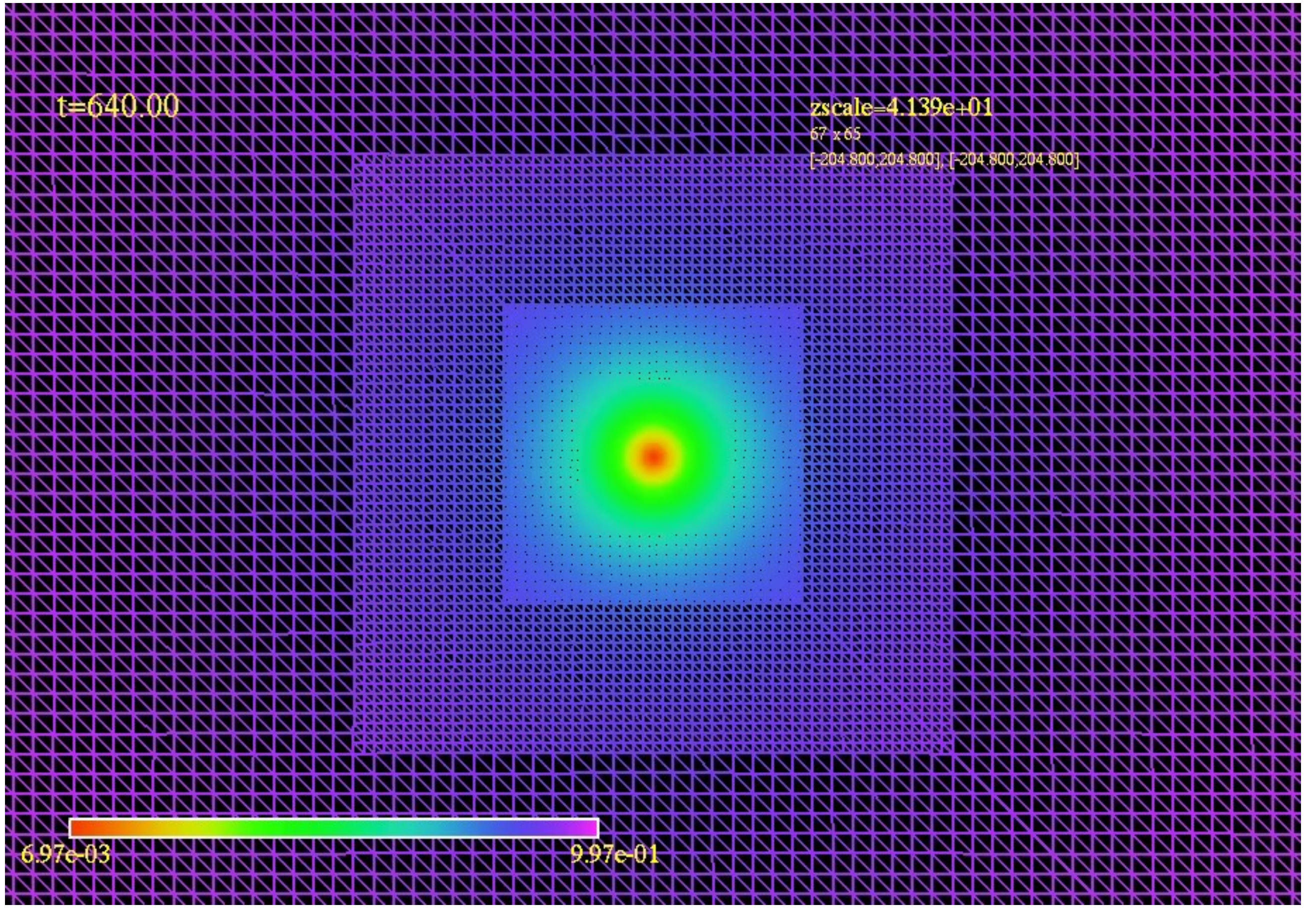


- Spherhake, Cardoso, Ott, Schnetter & Witek, PRD **84** 084038 (2011).

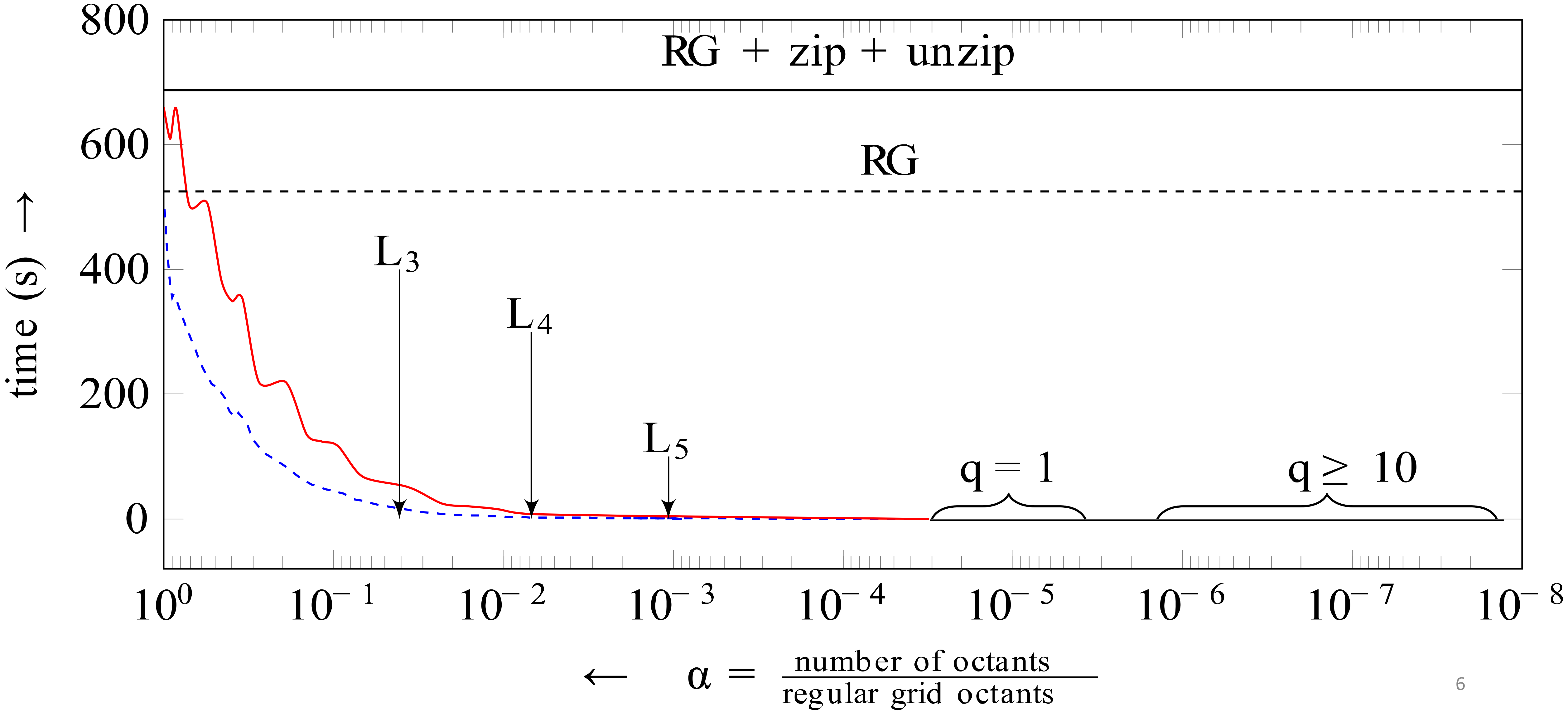
Numerical Relativity

- Conventional AMR uses nested boxes
- Boxes don’t naturally capture the geometry of binary black holes
- Numerical artefacts
- Computational Inefficiency

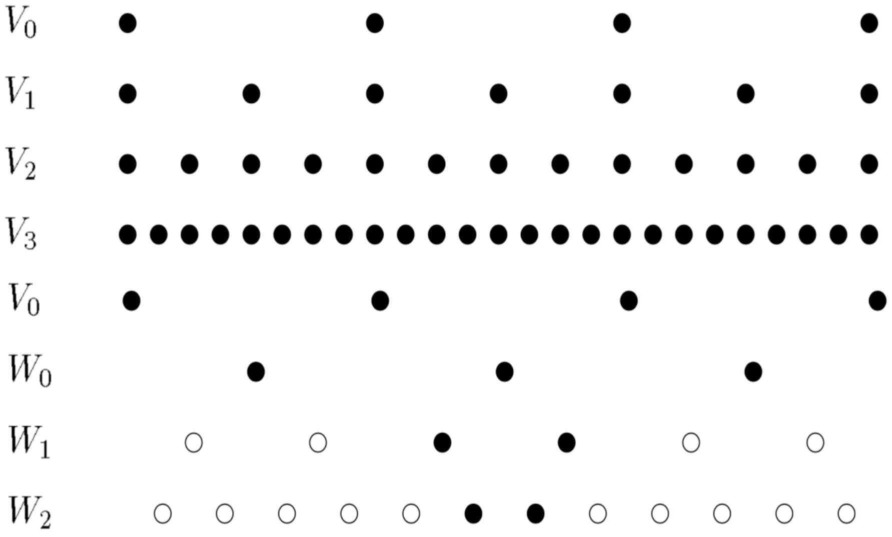
- **Need unstructured grids**
- **Need supercomputers**

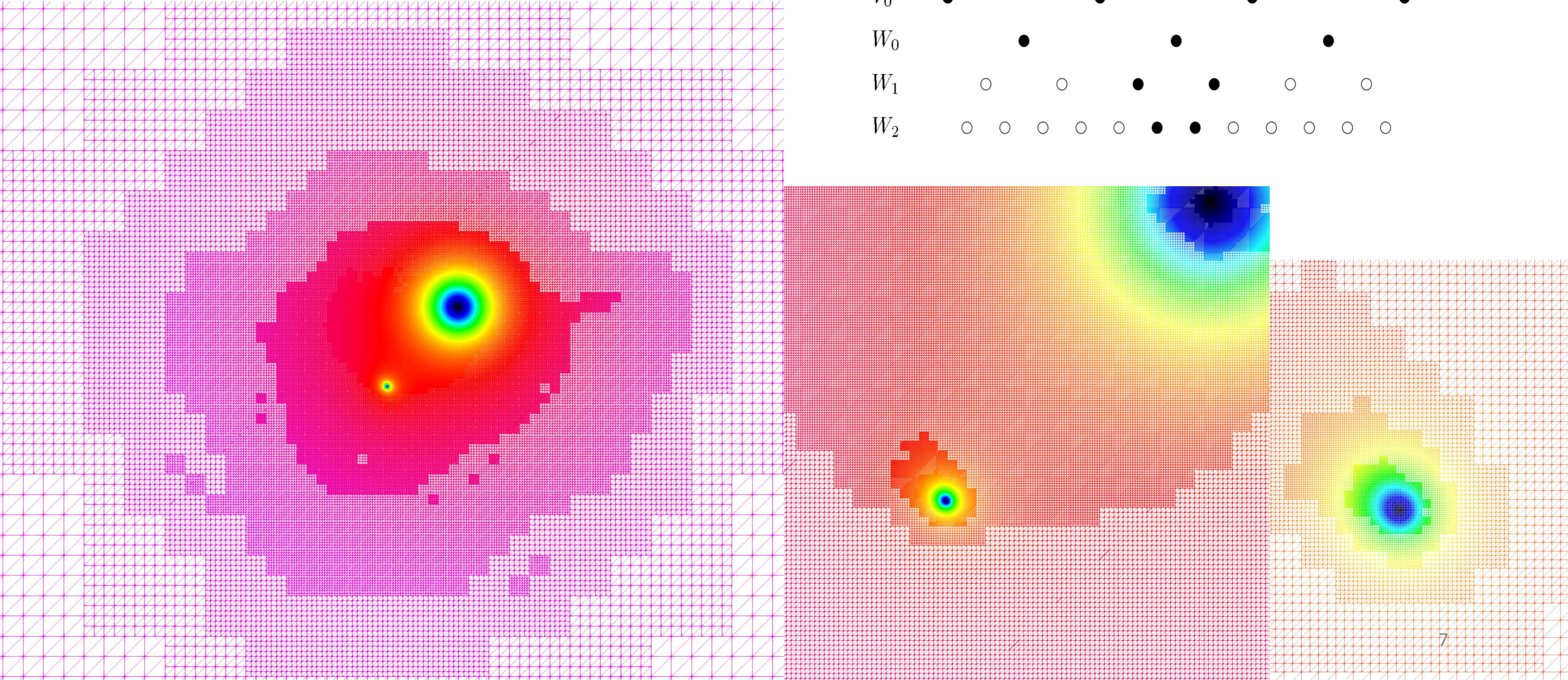


Why Block Adaptivity is not enough



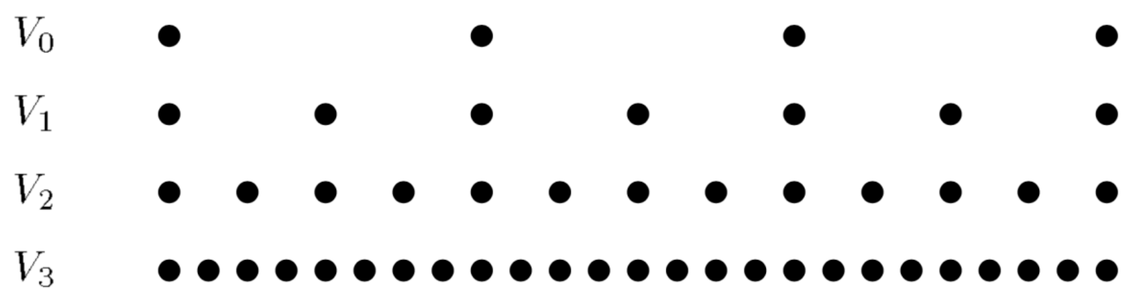
Octrees & Wavelets



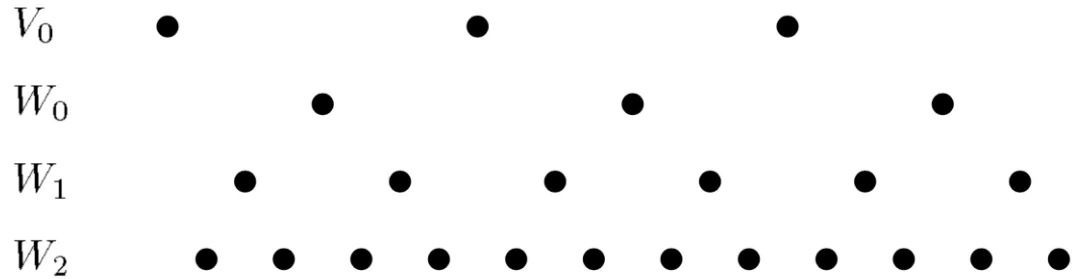


Wavelet Adaptive Multiresolution

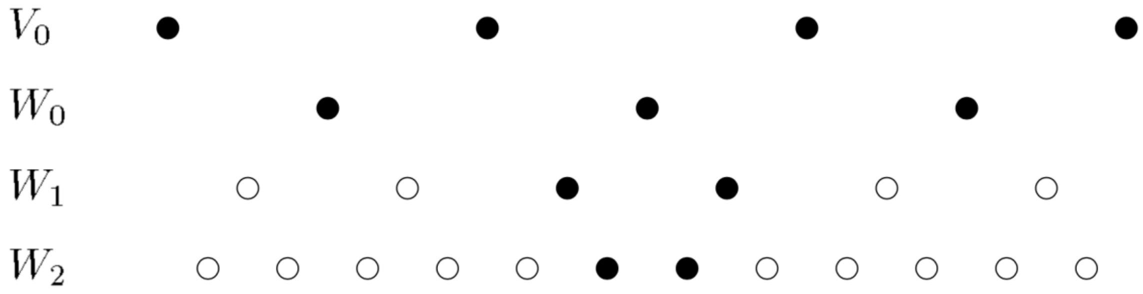
Scaling function

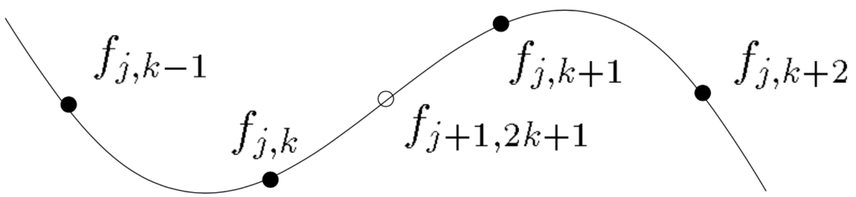


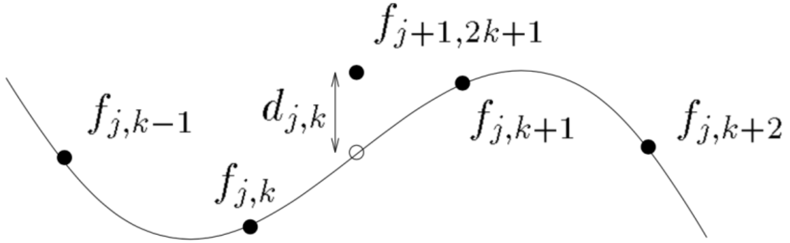
Wavelet basis

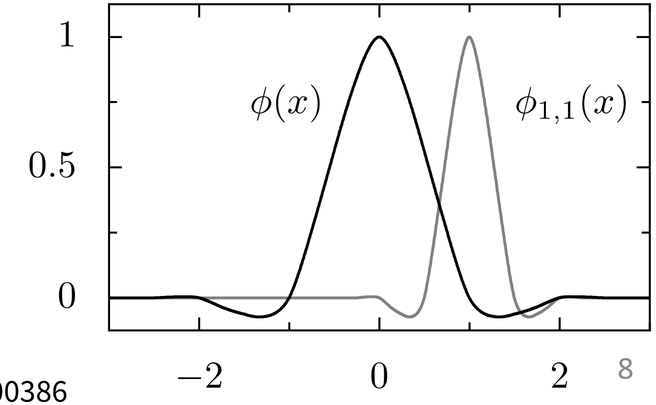


Sparse Representation





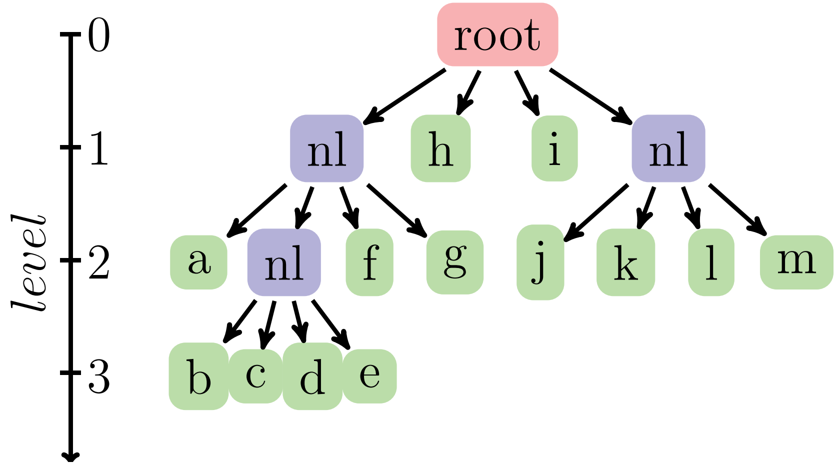


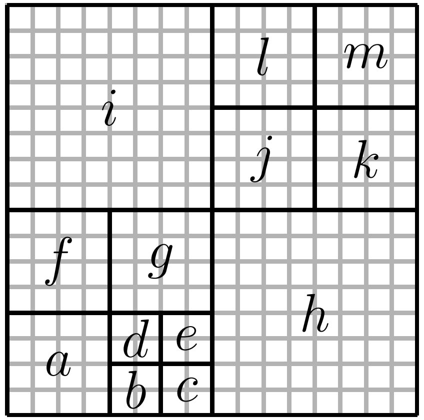


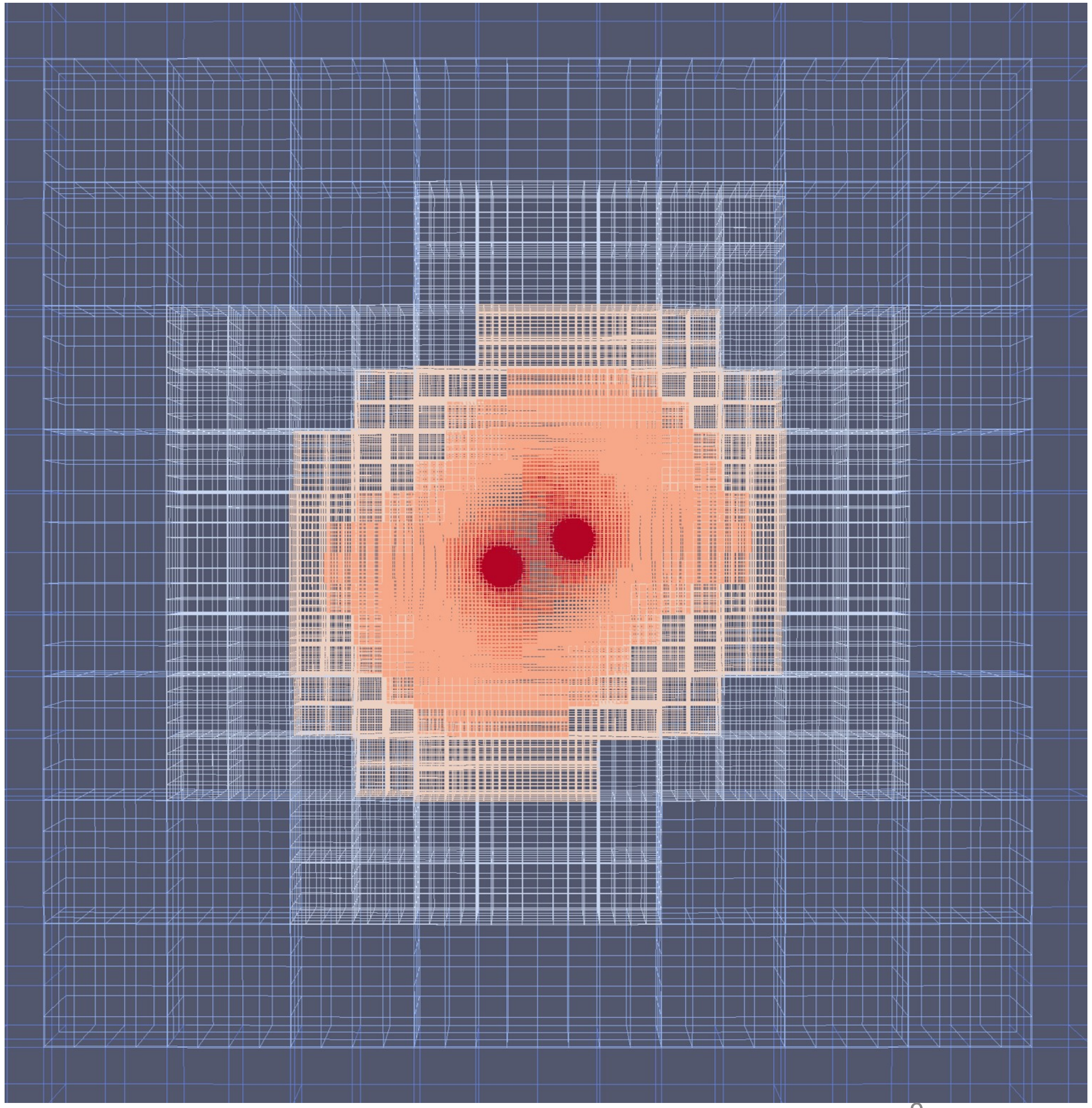
Figures from Holmström (1996) and arXiv:1512.00386

Octree-based AMR

- Axis-aligned subdivision of space
- In $2D$ each node has 4 children, 8 in $3D$
- Provides high-levels of adaptivity while enabling simple and efficient data-structures, especially in parallel

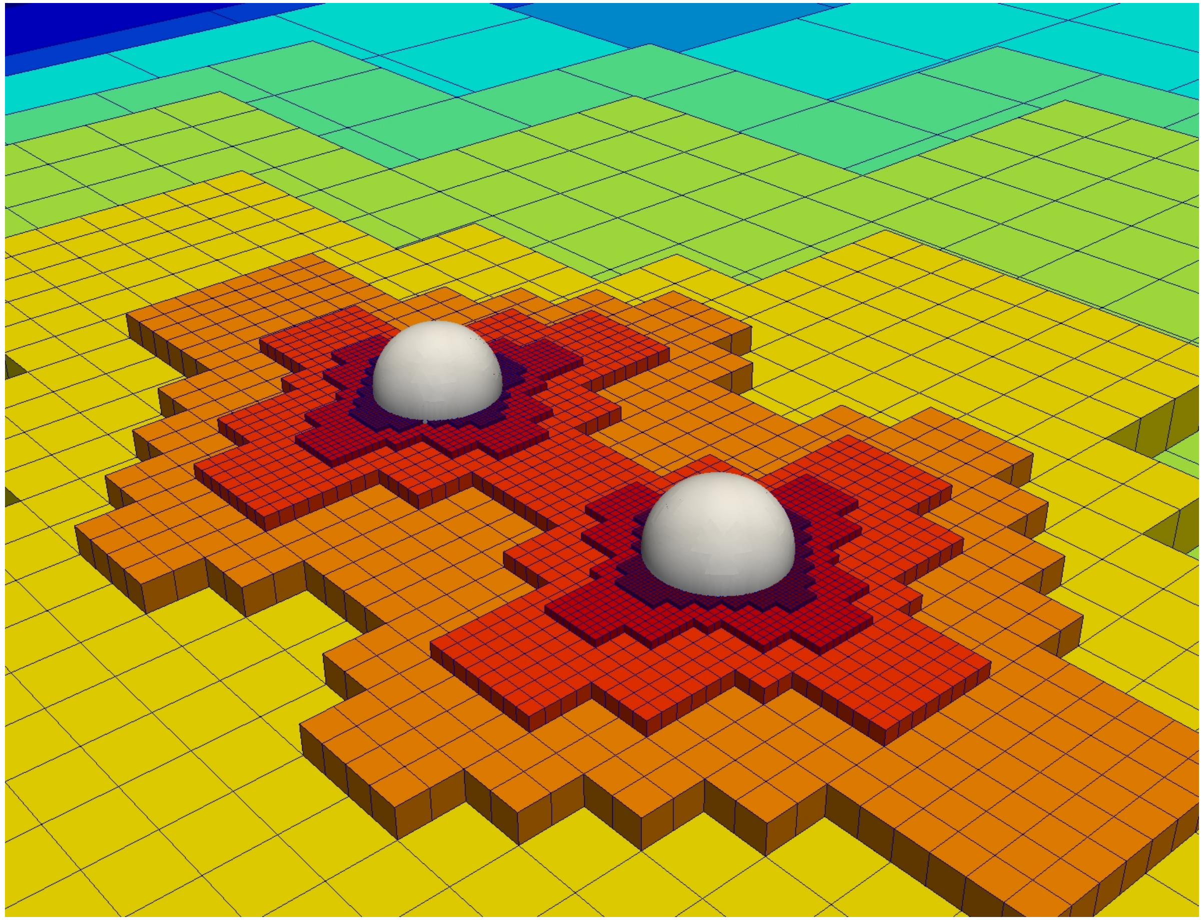




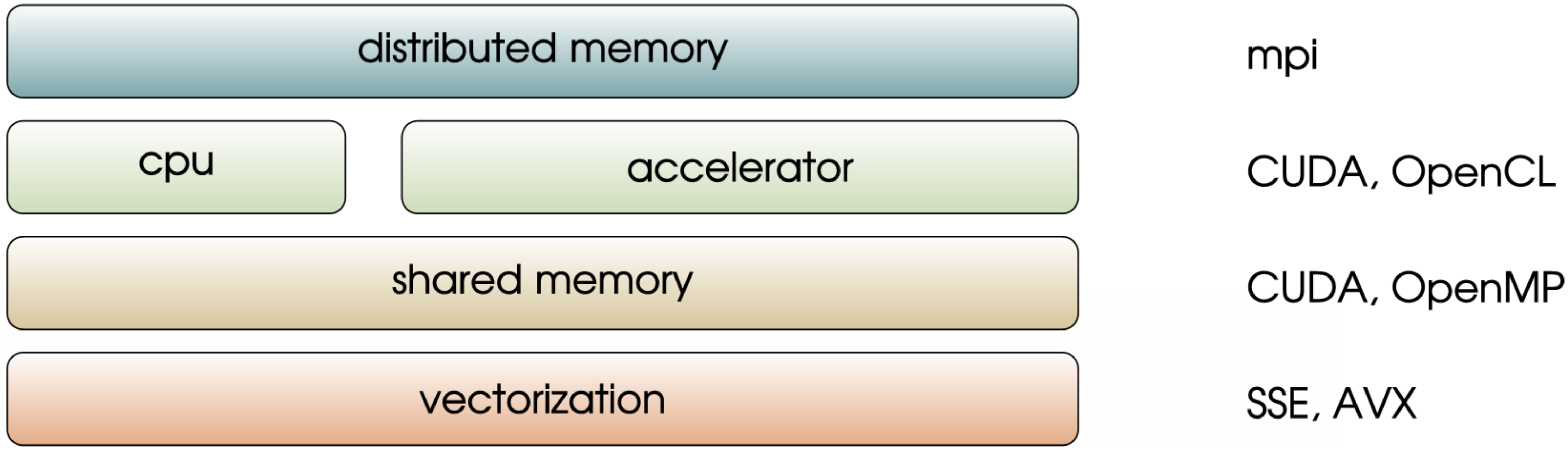


dendro-GR

- Wavelet adaptive multiresolution
- Unstructured Octree Grid
- High levels of fine-grained parallelism
- Automatic code-generation via symbolic interface
- Extensible
- Portable and highly-scalable on modern supercomputers

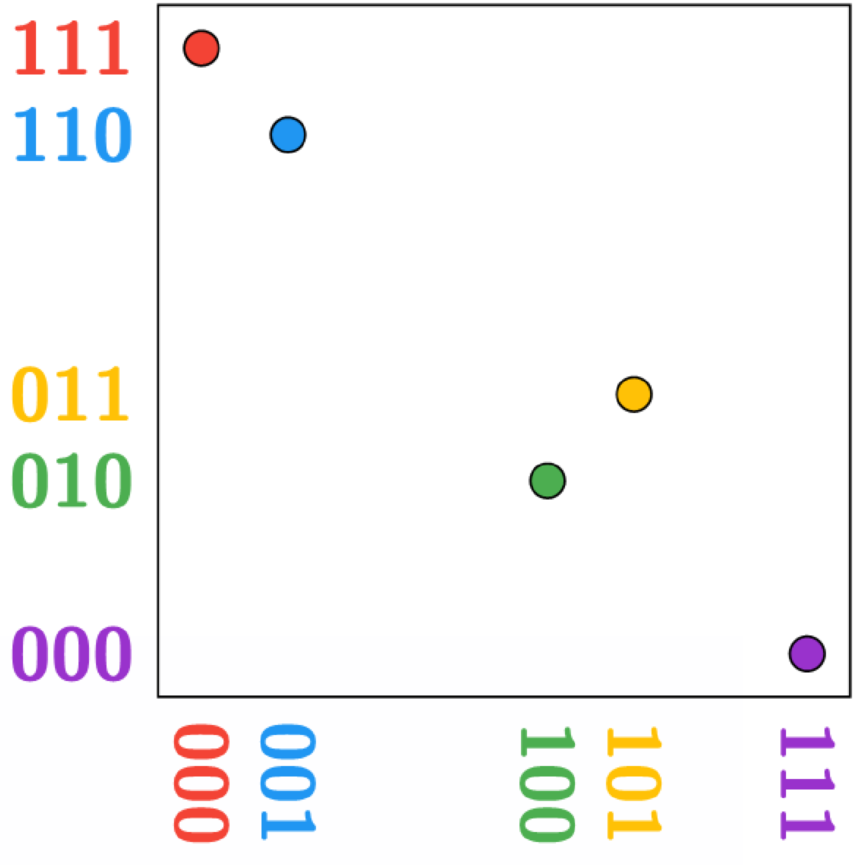


Parallelism



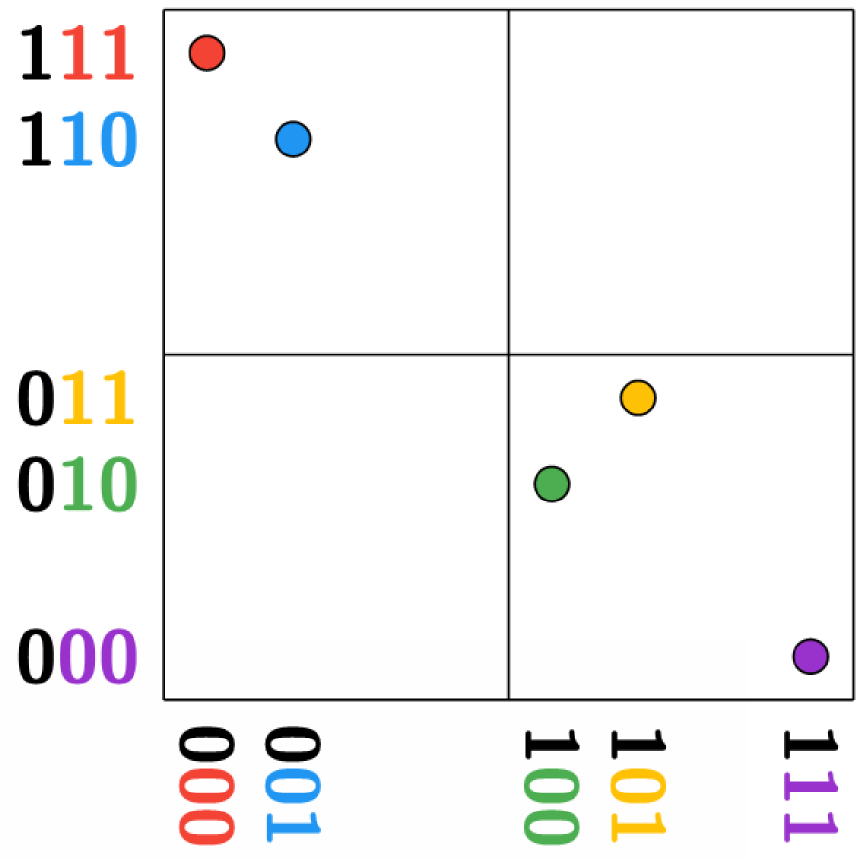
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored- linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering



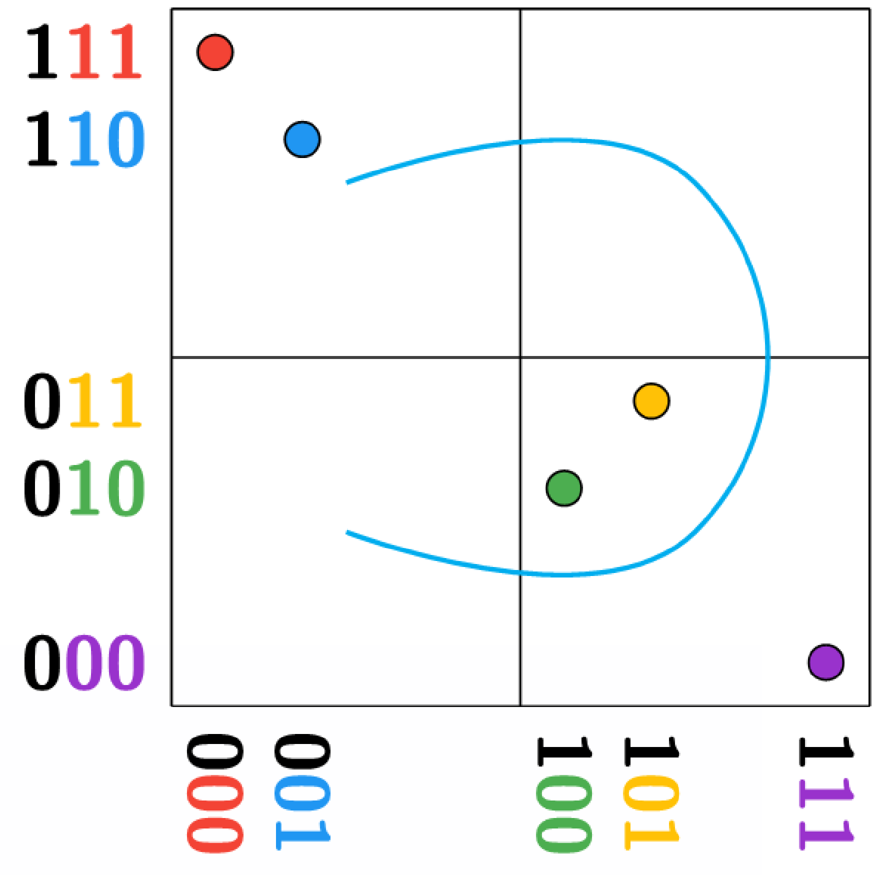
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored-linear octree
- Leaves are ordered according to Space filling Curves (SFC)
    - High spatial locality
    - Hilbert ordering
    - Morton Ordering



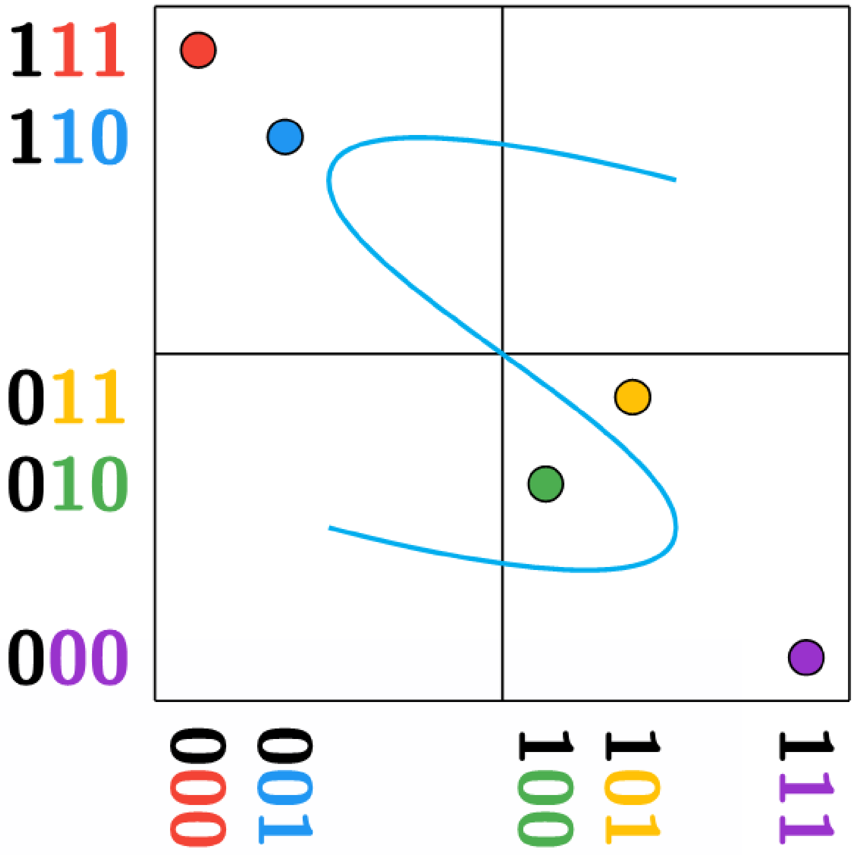
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored- linear octree
- Leaves are ordered according to Space filling Curves (SFC)
    - High spatial locality
    - Hilbert ordering
    - Morton Ordering



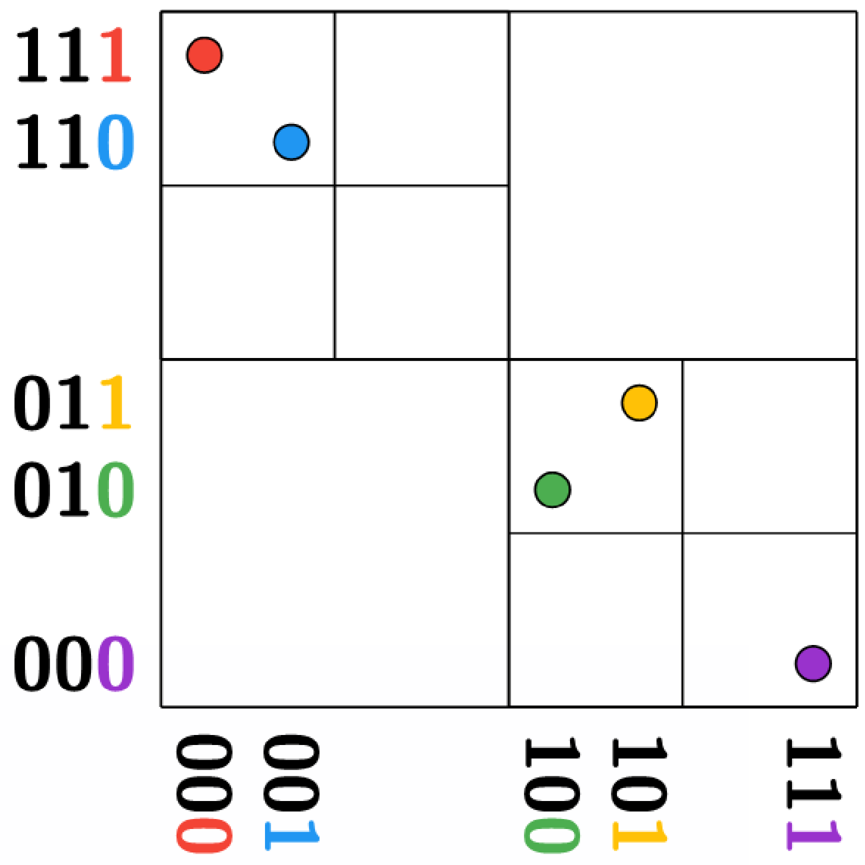
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored- linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering



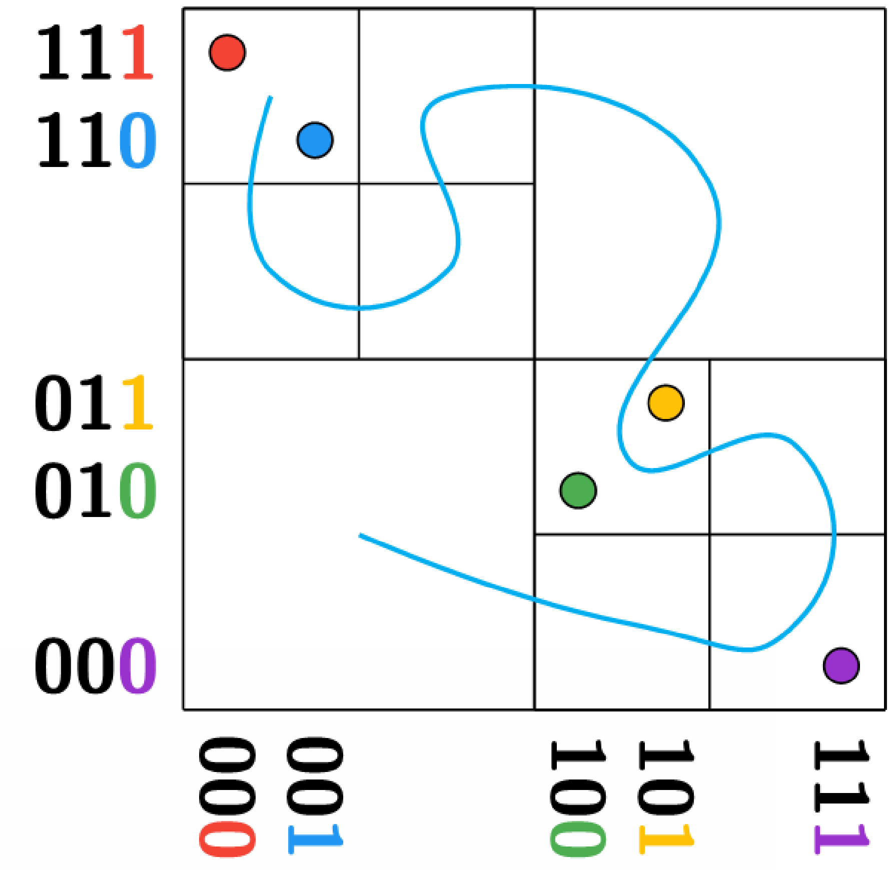
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored-linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering



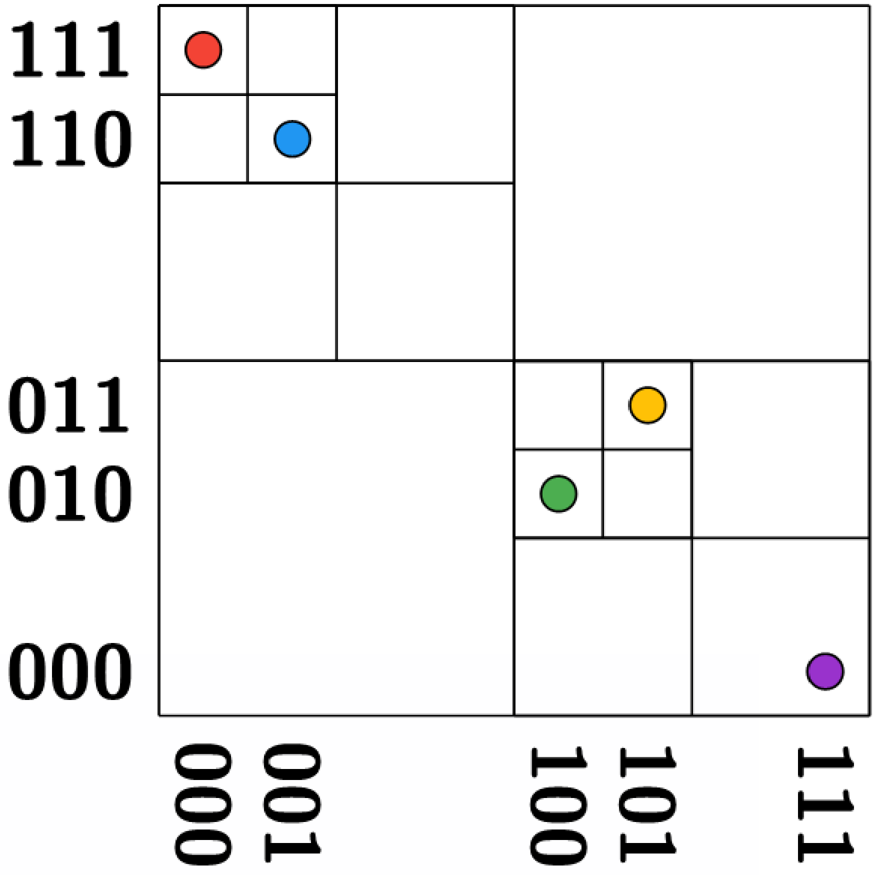
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored-linear octree
- Leaves are ordered according to Space filling Curves (SFC)
    - High spatial locality
    - Hilbert ordering
    - Morton Ordering



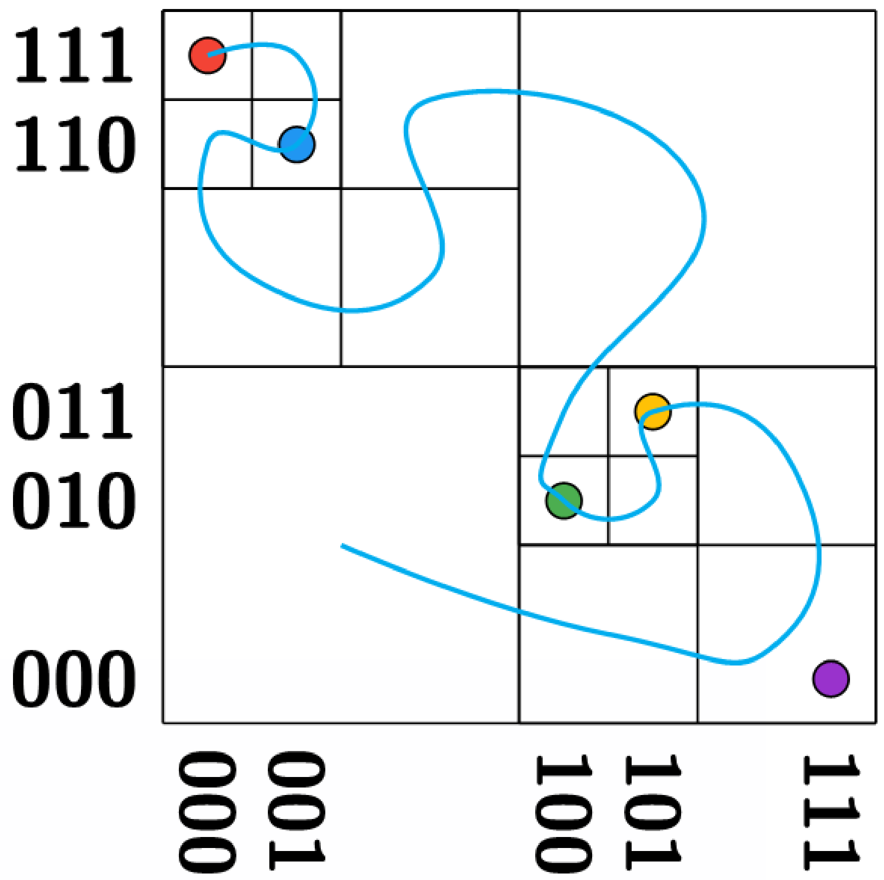
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored- linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering



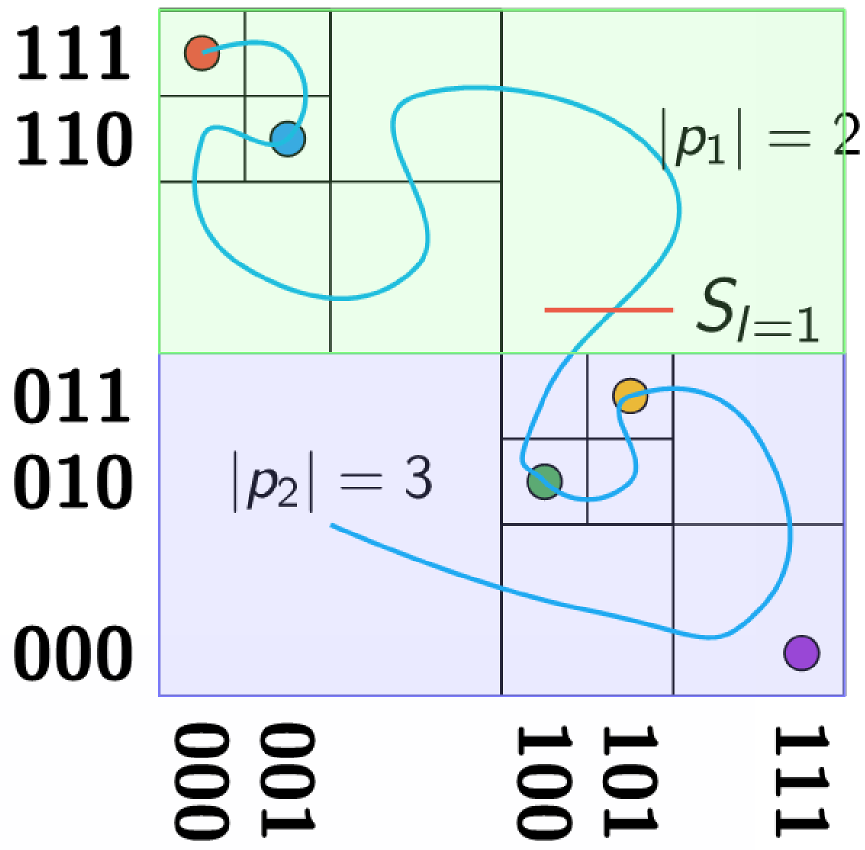
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored- linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering



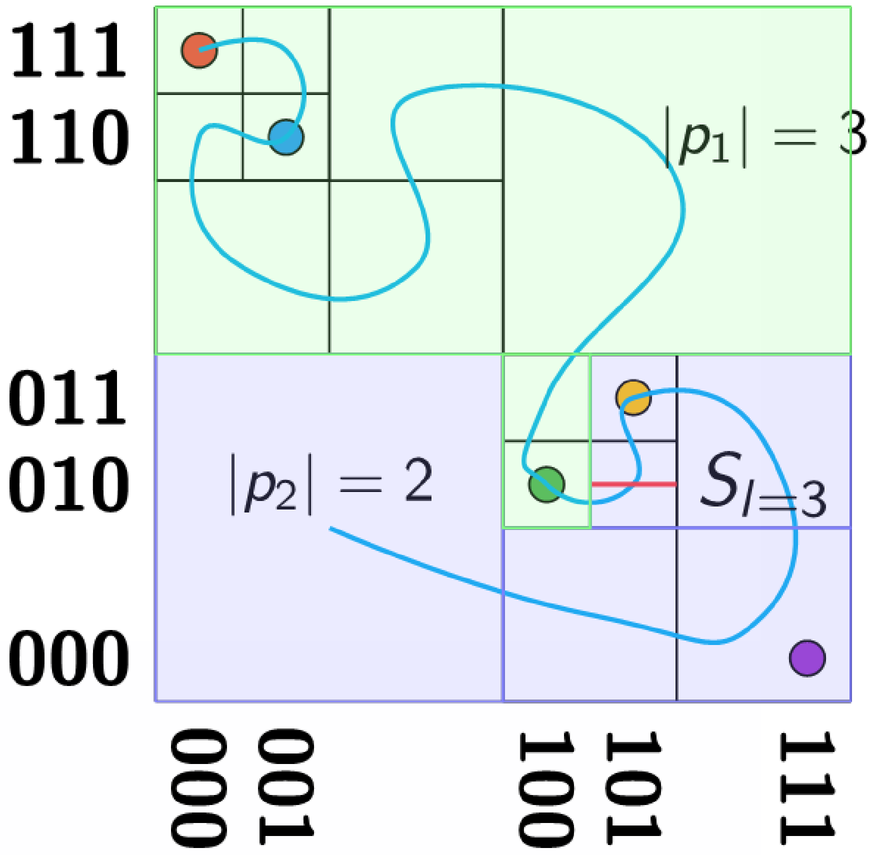
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored-linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering



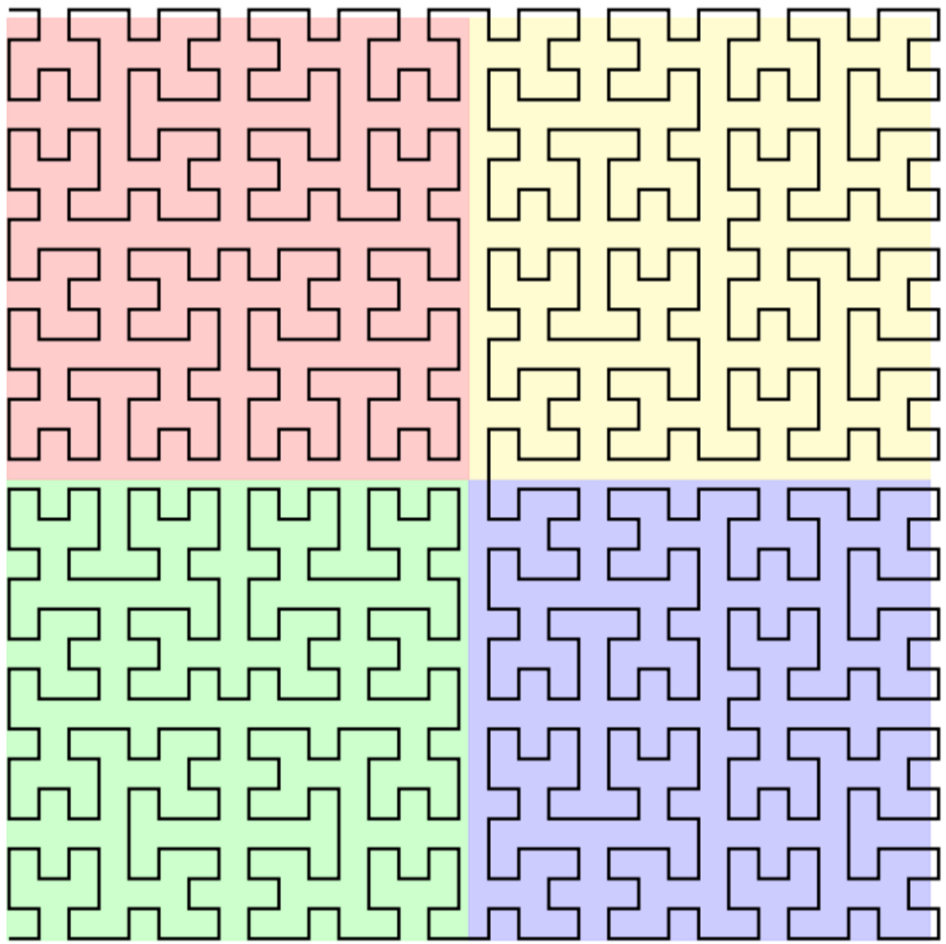
Octree Construction & Representation

- Top-down algorithm for constructing octree
- Only leaf nodes are stored- linear octree
- Leaves are ordered according to Space filling Curves (SFC)
  - High spatial locality
  - Hilbert ordering
  - Morton Ordering

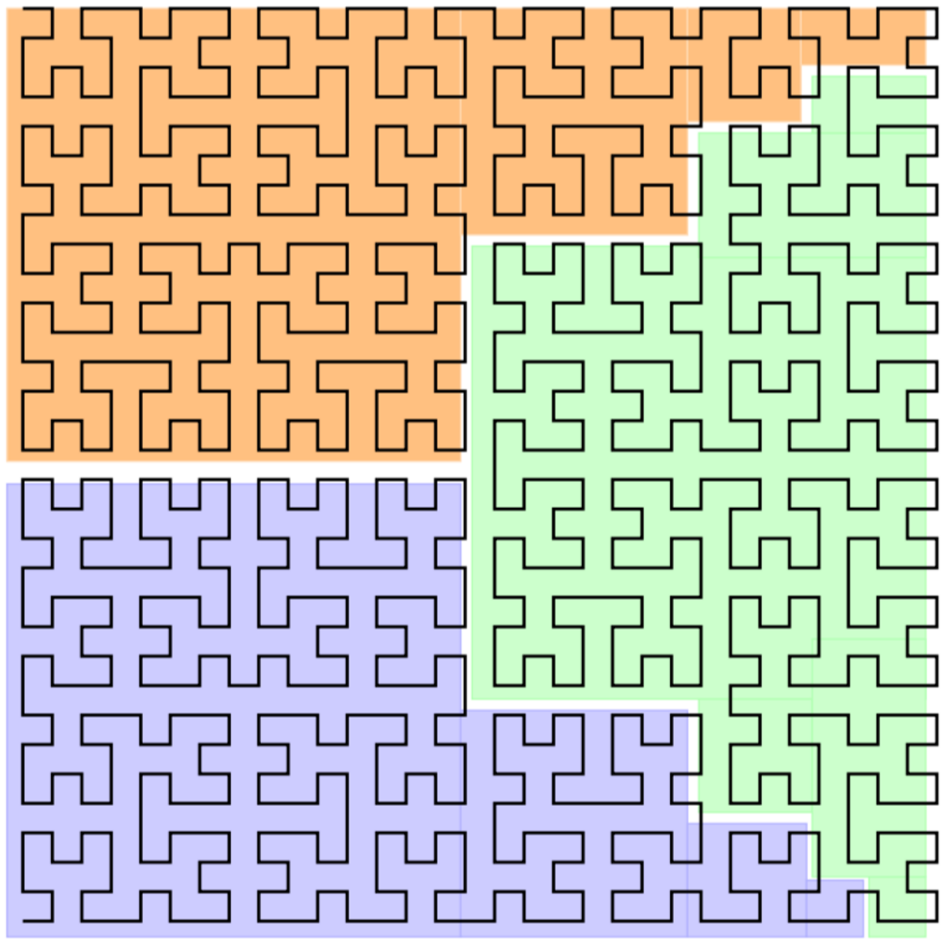


SFCs for Partitioning Data

$level{=}5,\ \#partitions{=}4$

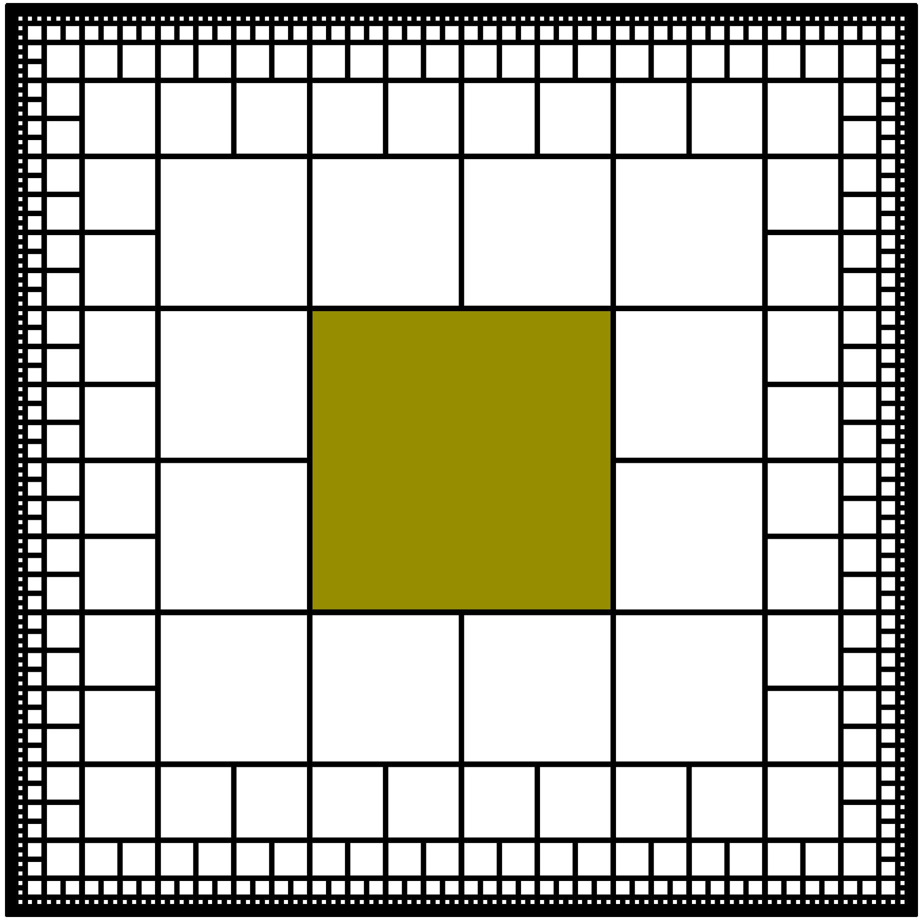


$level{=}5,\ \#partitions{=}3$



2:1 Balance constraint

- Simplifies mesh & neighborhood
- Does not sacrifice adaptivity
- Minimizes the need to interpolate data
- Minimizes data-dependencies



Computational Methods

- Relativistic Fluids
  - Finite difference HRSC Method
  - HLLE flux
  - MP5 reconstruction
- Einstein Equations
  - BSSN formulation
  - 4^{th} order finite differences
  - Kreiss-Oliger dissipation

Computational Methods

- Relativistic Fluids
  - Finite difference HRSC Method
  - HLLE flux
  - MP5 reconstruction

Actually…
nothing special

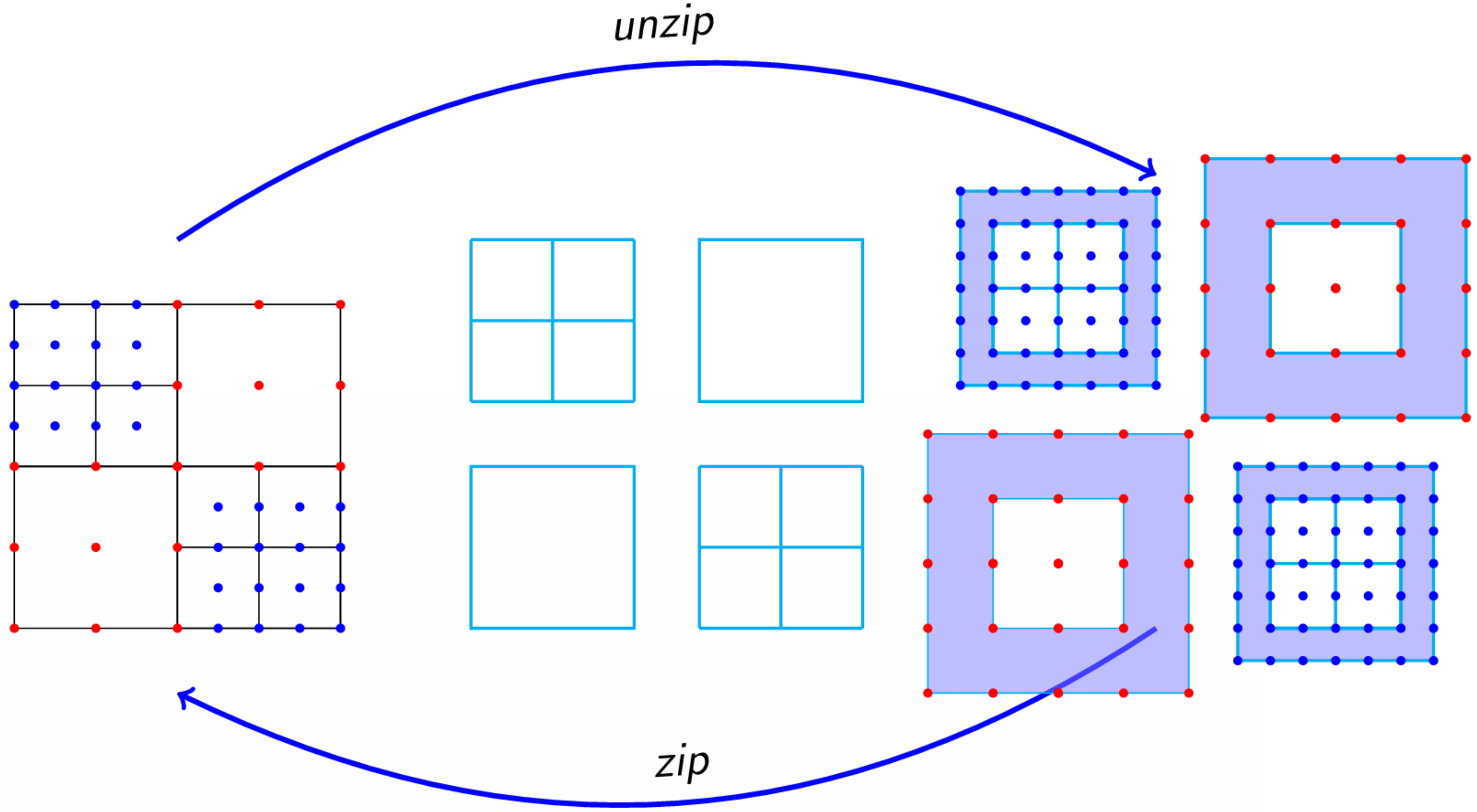
- Einstein Equations
  - BSSN formulation
  - 4th order finite differences
  - Kreiss-Oliger dissipation

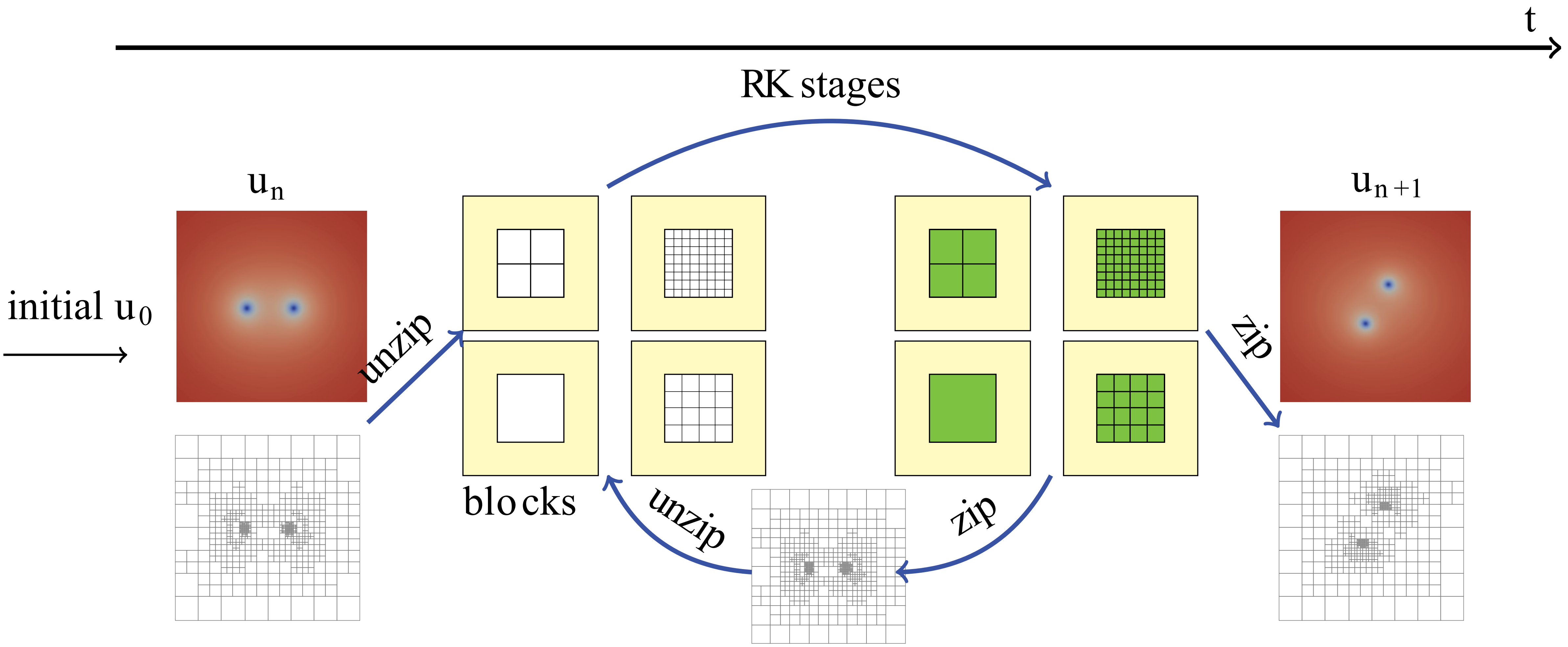
These are just the conventional numerical methods

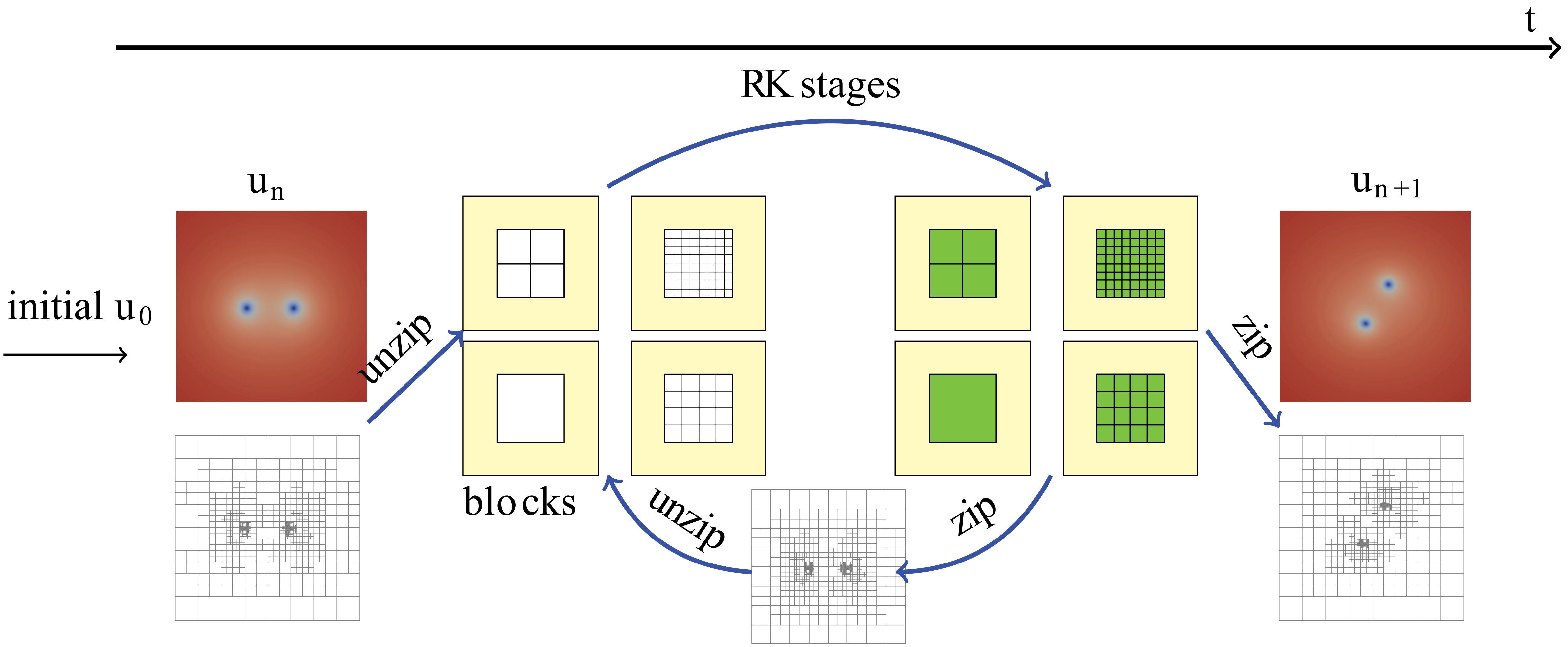
Finite differences & unstructured grids

- We need a regular grid to apply FD stencils
- This is not available everywhere for octree-refined grids

Additional operations are required before applying FD stencils




t
RK stages
u_n
u_{n+1}
initial u_0
unzip
blocks
unzip
zip
zip



BSSN equations applied at a block level.

Automatic Code Generation

BSSN Equations

$$\partial_t \alpha = \mathcal{L}_\beta \alpha - 2\alpha K$$

$$\partial_t \beta^i = \lambda_2 \beta^j \partial_j \beta^i + \frac{3}{4} f(\alpha) B^i$$

Dendro Code

```
from dendro import *

a_rhs = l1*dendro.lie(b, a) - 2*a*K

b_rhs = [ 3/4*f(a)*B[i] + l2*dendro.vec_j_ad_j(b, b[i])
                                  for i in dendro.e_i ]
```

Automatic Code Generation

```
# declare variables
a = dendro.scalar("alpha", "[pp]")
Gt = dendro.vec3("Gt", "[pp]")
gt = dendro.sym_3x3("gt", "[pp]")

dendro.set_metric(gt)
igt = dendro.get_inverse_metric()

a_rhs = l1*dendro.lie(b, a) - 2*a*K
..


outs = [a_rhs, b_rhs, gt_rhs, chi_rhs, At_rhs, K_rhs, Gt_rhs, B_rhs]
vnames = ['a_rhs', 'b_rhs', 'gt_rhs', 'chi_rhs', 'At_rhs', 'K_rhs', 'Gt_rhs', 'B_rhs’]
dendro.generate(outs, vnames, '[pp]')
```

Automatic Code Generation

```
// Dendro: {{{
// Dendro: original ops: 678611double DENDRO_0 = 2*alpha;
double DENDRO_1 = 0.75*alpha*lambda_f[1] + 0.75*lambda_f[0];
double DENDRO_2 = grad(0, beta0);
double DENDRO_3 = grad(1, beta1);
.
.
.
B_rhs0 = -B0*eta - DENDRO_952*lambda[3] + DENDRO_993 + lambda[2]*(beta0*agrad(0, B0) + beta1*agrad(1, B0) + beta2*agrad(2, B0));
B_rhs1 = -B1*eta + DENDRO_1003 - DENDRO_994*lambda[3] + lambda[2]*(beta0*agrad(0, B1) + beta1*agrad(1, B1) + beta2*agrad(2, B1));
B_rhs2 = -B2*eta - DENDRO_1004*lambda[3] + DENDRO_1006 + lambda[2]*(beta0*agrad(0, B2) + beta1*agrad(1, B2) + beta2*agrad(2, B2));
// Dendro: reduced ops: 4602
// Dendro: }}}
```

Automatic Code Generation - vectorization

```
// Dendro vectorized code: {{{
double v0 = 2.0;
double v1 = alpha[pp];
double v2 = dmul(v1, v0);
.
.
v14 = B2[pp];
v15 = eta;
v16 = dmul(v15, v14);
v17 = dmul(v16, negone);
v18 = DENDRO_989;
v19 = lambda[3];
v20 = dmul(v19, v18);
v21 = dmul(v20, negone);
v22 = dadd(v21, v17);
v23 = dadd(v22, v13);
v24 = dadd(v23, v0);
B_rhs2[pp] = v24;
// Dendro vectorized code: }}}
```

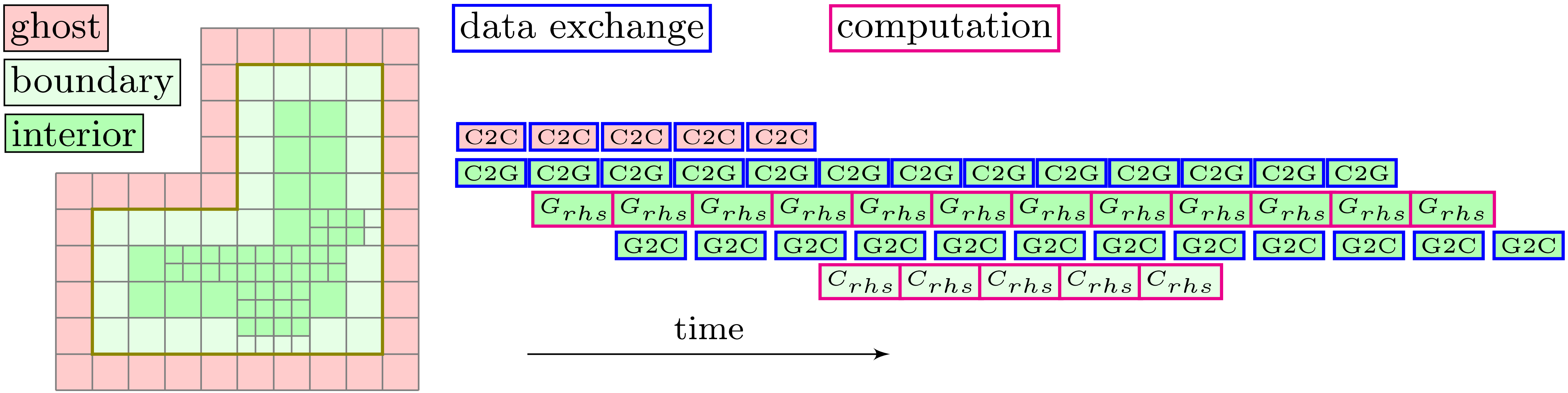
Automatic Code Generation - CUDA

```
//input vars begin
  double * K = __sm_base + 0;
  double * gt1 = __sm_base + 27;
  double * beta1 = __sm_base + 54;
  double * gt3 = __sm_base + 81;
  double * At1 = __sm_base + 108;
  double * gt5 = __sm_base + 135;
  double * alpha = __sm_base + 162;
  double * gt4 = __sm_base + 189;
  double * gt2 = __sm_base + 216;
  double * beta2 = __sm_base + 243;
  double * At3 = __sm_base + 270;
  double * At4 = __sm_base + 297;
  double * At0 = __sm_base + 324;
  double * At2 = __sm_base + 351;
  double * beta0 = __sm_base + 378;
  double * gt0 = __sm_base + 405;
  double * chi = __sm_base + 432;
  double * At5 = __sm_base + 459;
```

```
// deriv vars begin
  double * grad2_0_0_gt3 = __sm_base + 486;
  double * grad2_2_2_alpha = __sm_base + 513;
  double * grad2_1_2_gt1 = __sm_base + 540;
  double * grad_2_gt3 = __sm_base + 567;
// load data from global to shared memory
cuda::__loadGlobalToShared3D<double>(&__unzipInVar[cuda::VAR::U_K][offset],(double *) K,(const unsigned int *) ijk_lm,(const unsigned int *) alignedSz,(const unsigned int *) tile_sz);


if(!((threadIdx.x>(ijk_lm[1]-ijk_lm[0])) || (threadIdx.y>(ijk_lm[3]-ijk_lm[2]))) ){

    double x,y,z,r_coord,eta;

    unsigned int pp = 0*tile_sz[0]*tile_sz[1]+threadIdx.y*tile_sz[1]+threadIdx.x;

    for(unsigned int k=0;k<=(ijk_lm[5]-ijk_lm[4]);++k,pp+=tile_sz[0]*tile_sz[1]){
```

But modern clusters are heterogeneous!

Heterogeneous Architectures



- GPUs are very fast, but require SIMD (Single Instruction Multiple Data)
- CPUs handle inter-processor communication and boundary zones
- GPUs work on interior
- Computation and communication are interleaved

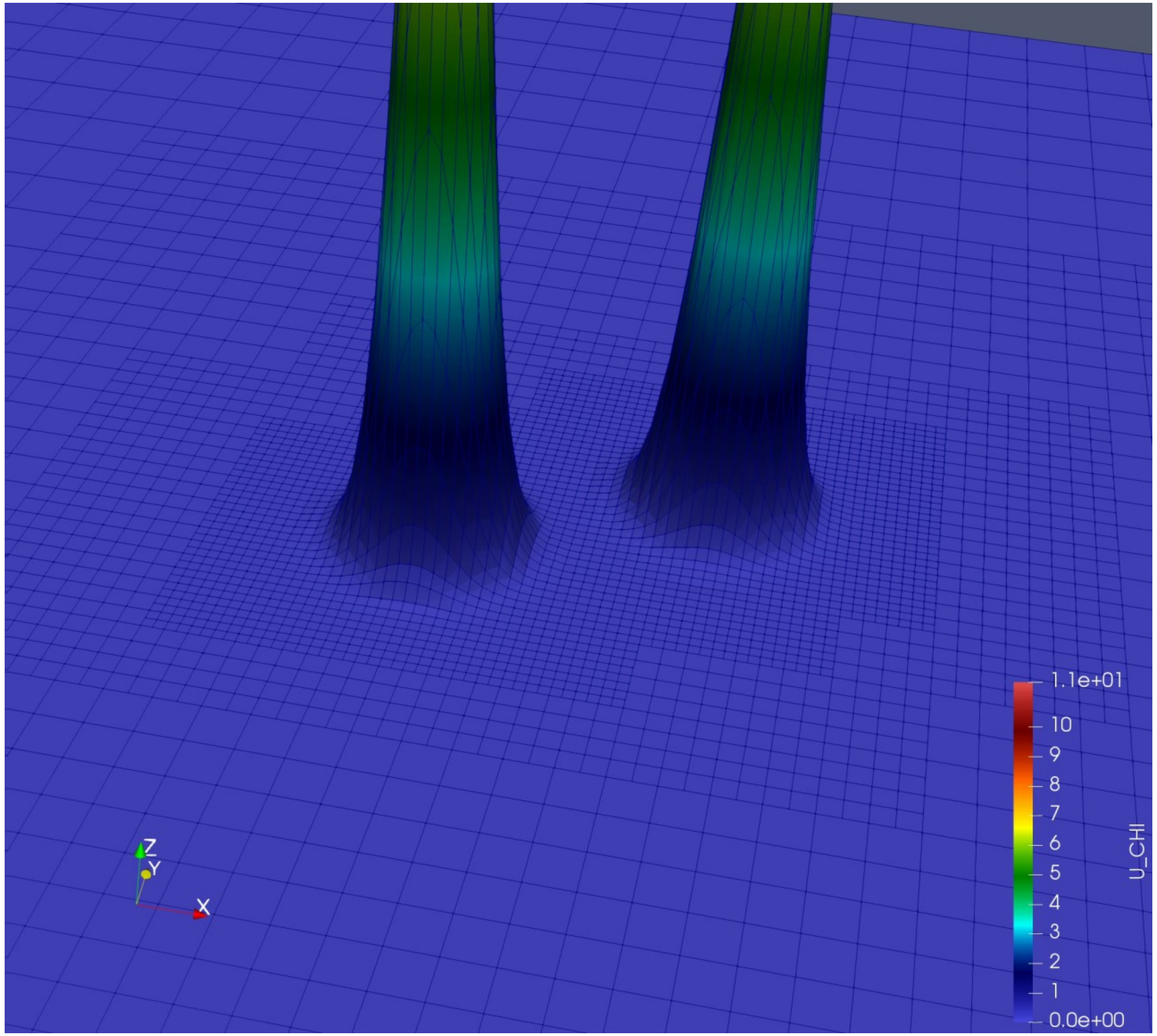
Experiments

Nonlinear Sigma Model

Connection to BH critical phenomena, Liebling (2004)

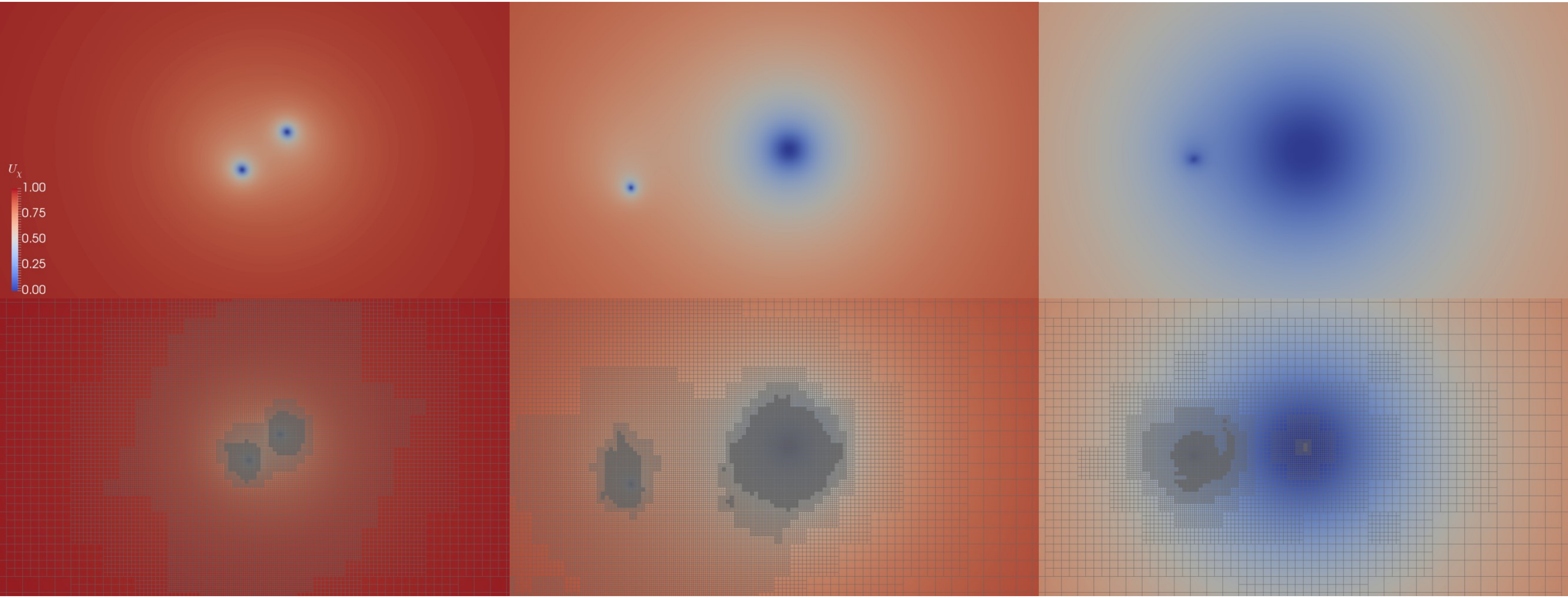
$$\partial_t^2 \phi - \nabla^2 \phi = -\frac{\sin 2\phi}{r^2}$$

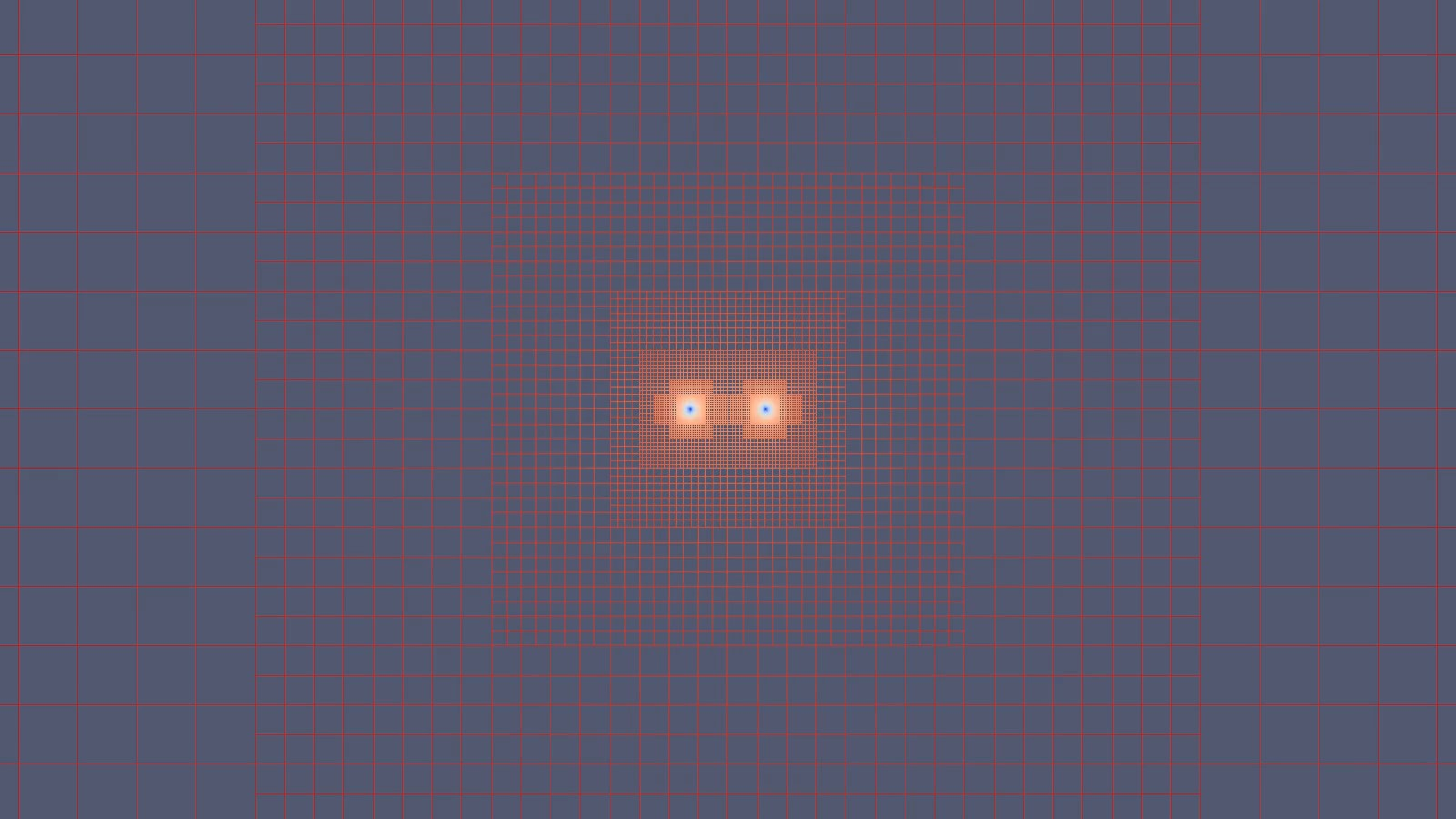
```
r = symbols('r')

# declare functions
chi = dendro.scalar("chi","[pp]")
phi = dendro.scalar("phi","[pp]")

phi_rhs = sum( d2(i,i,chi) for i in dendro.e_i )
            - sin(2*chi)/r**2

chi_rhs = phi
```



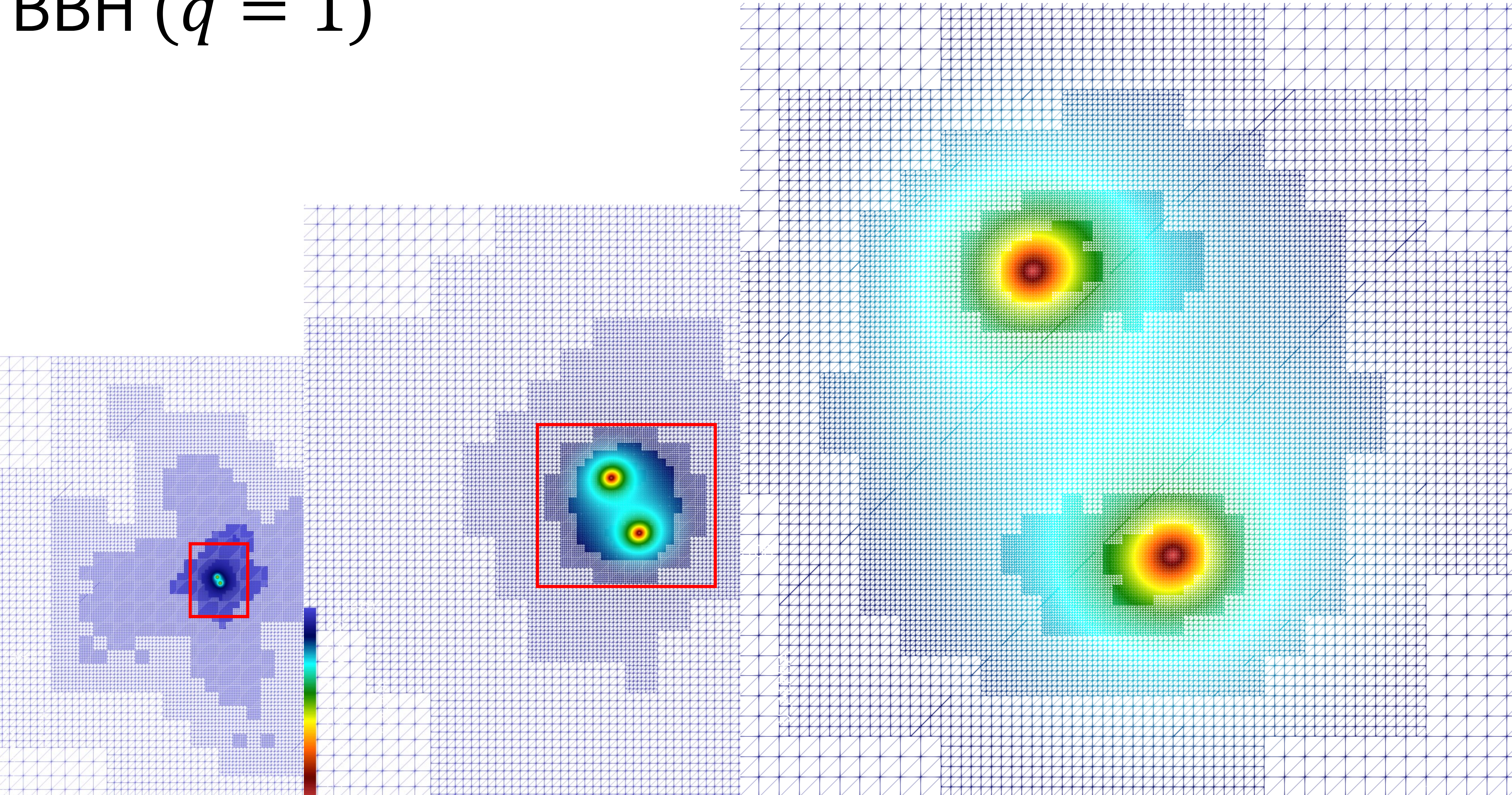
Binary Black Holes

1:1 | 10:1 | 100:1

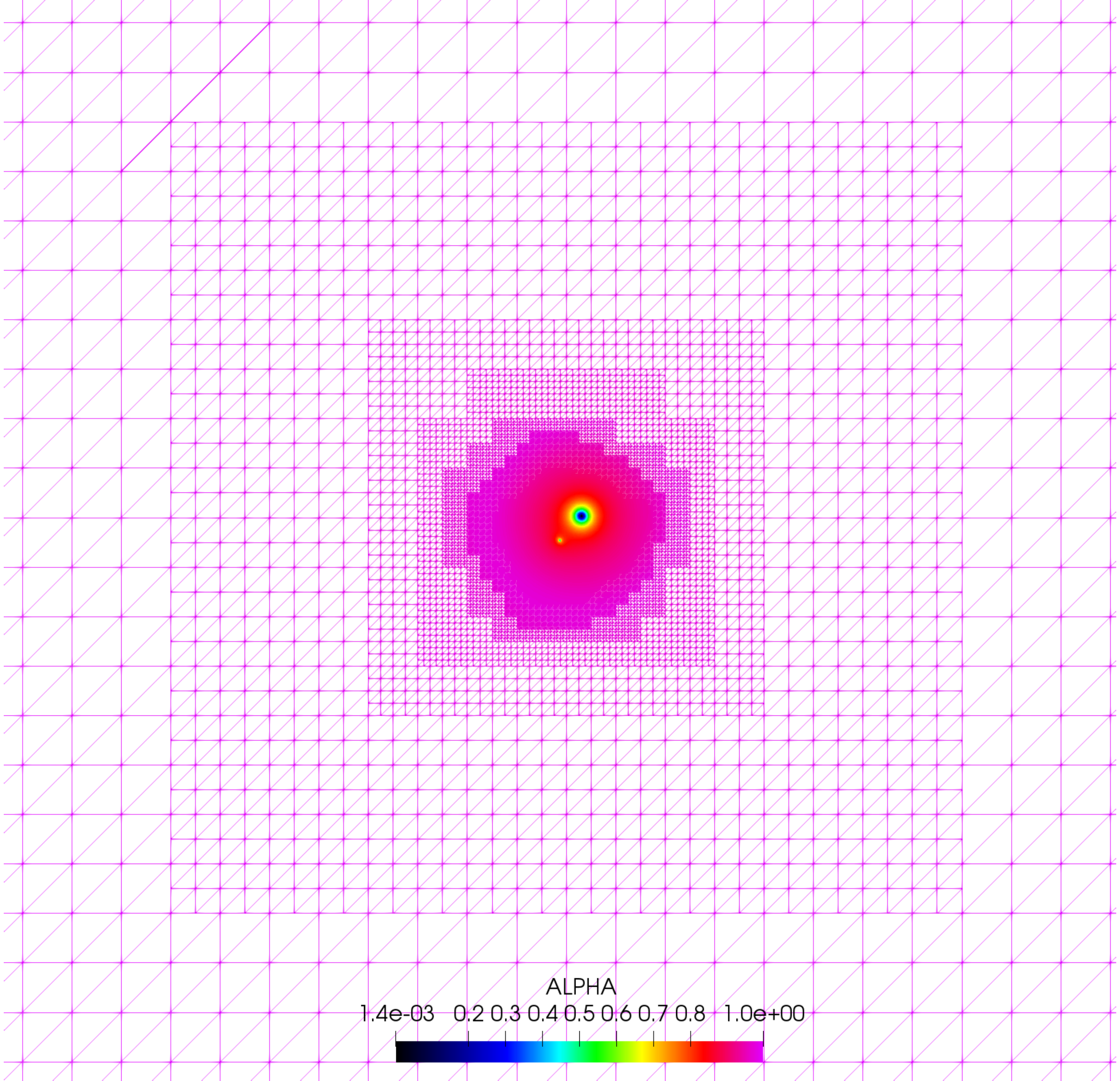




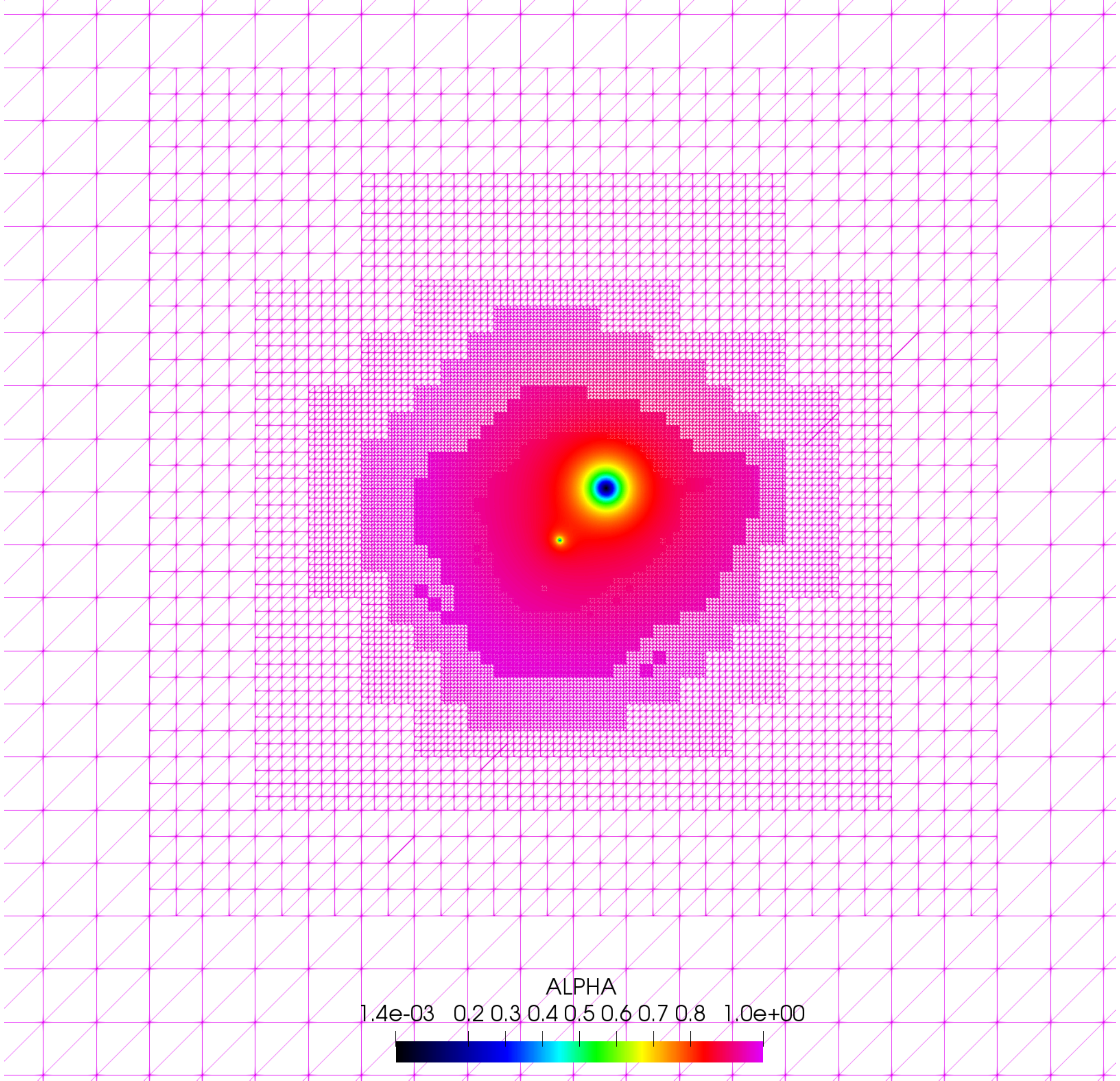
BBH ($q = 1$)



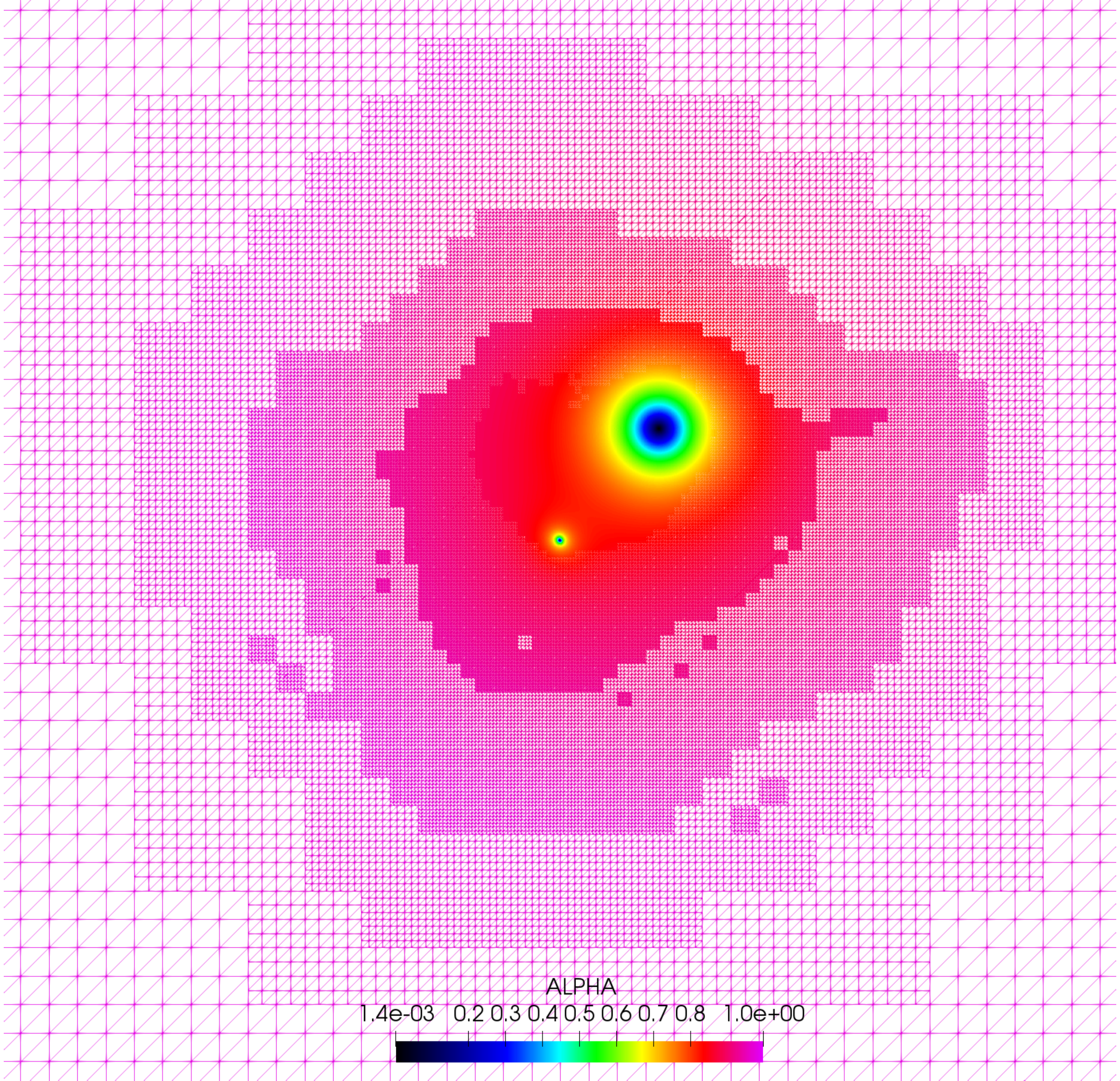
BBH ($q = 10$)



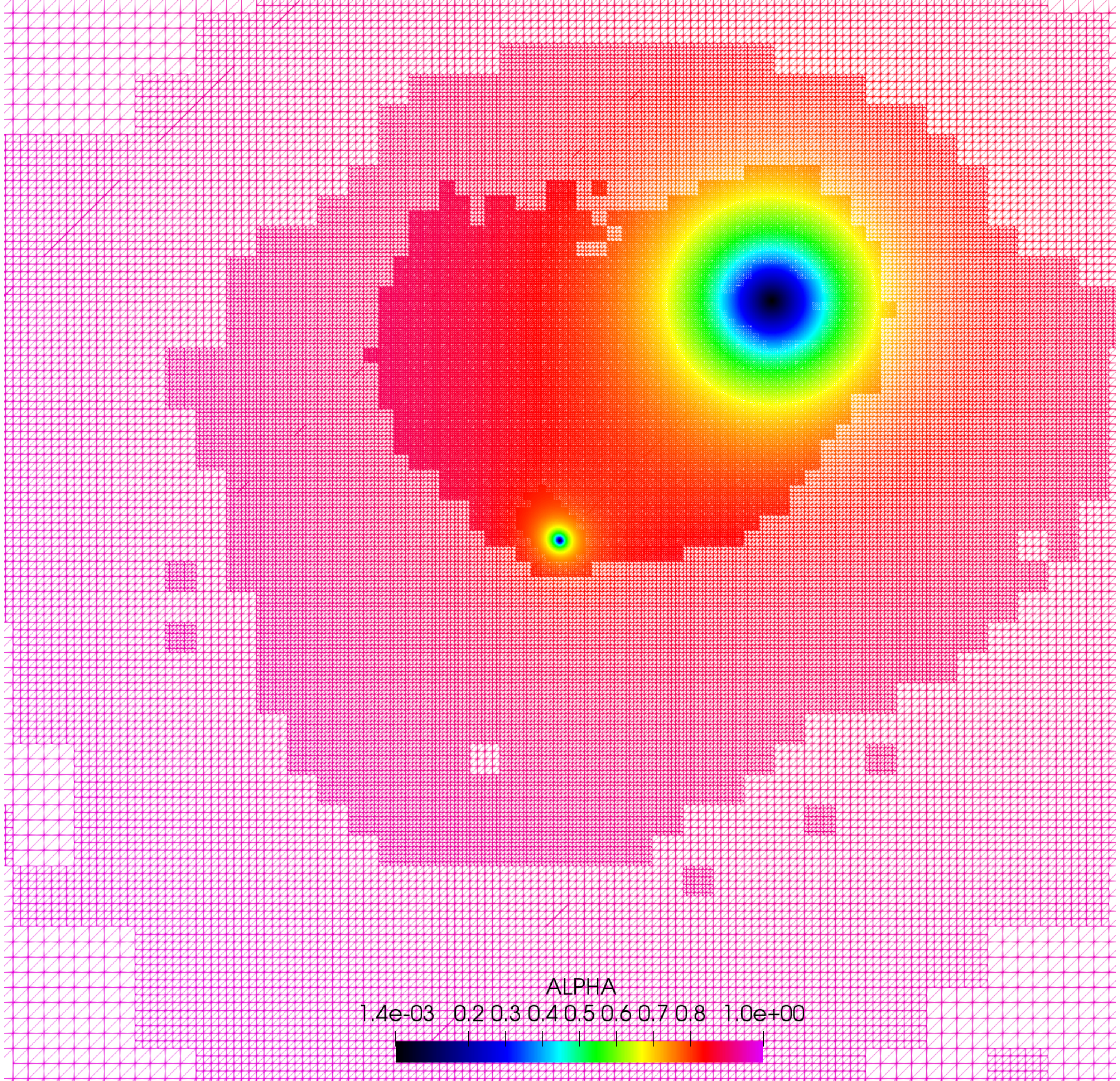
BBH ($q = 10$)



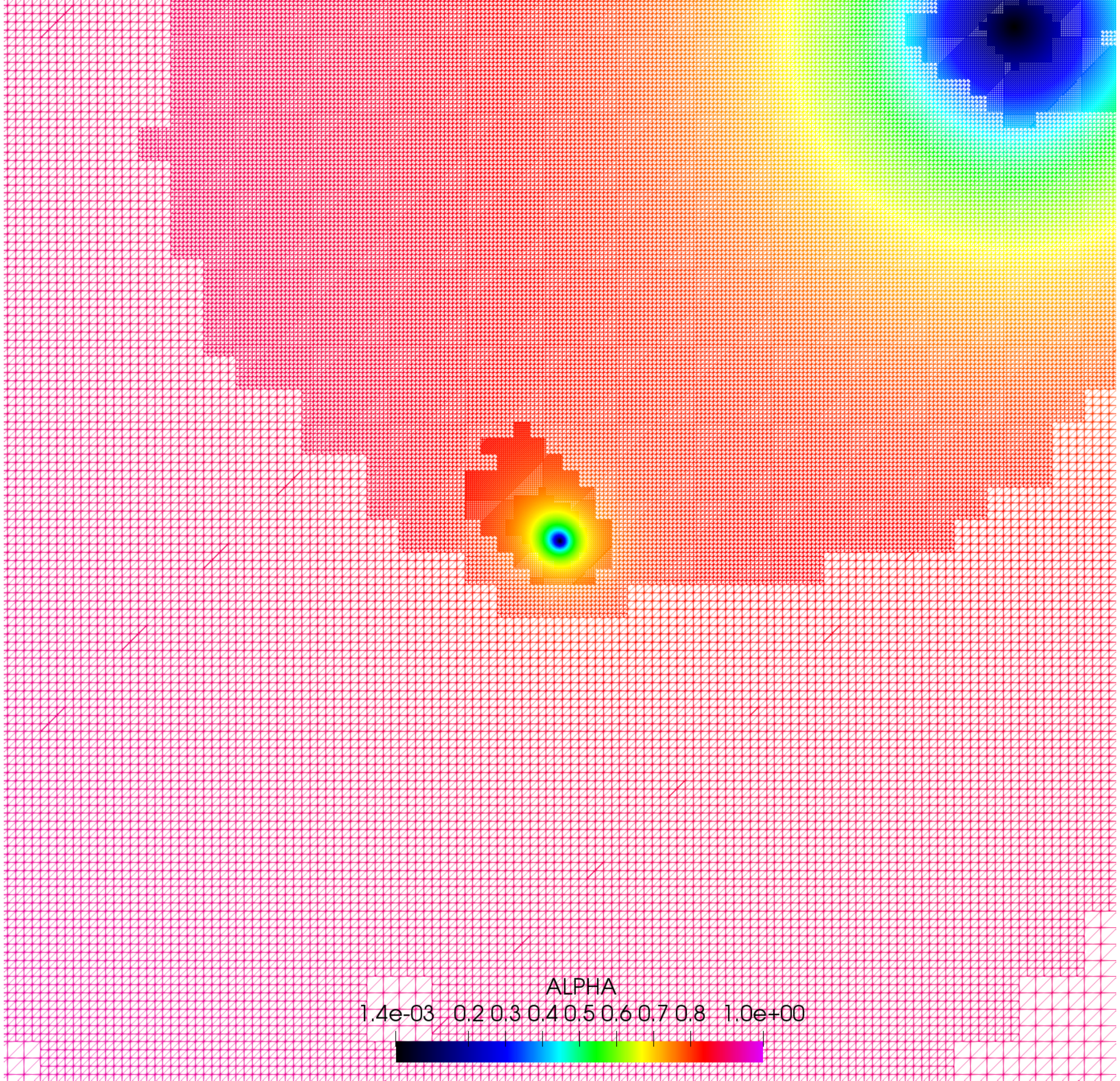
BBH ($q = 10$)



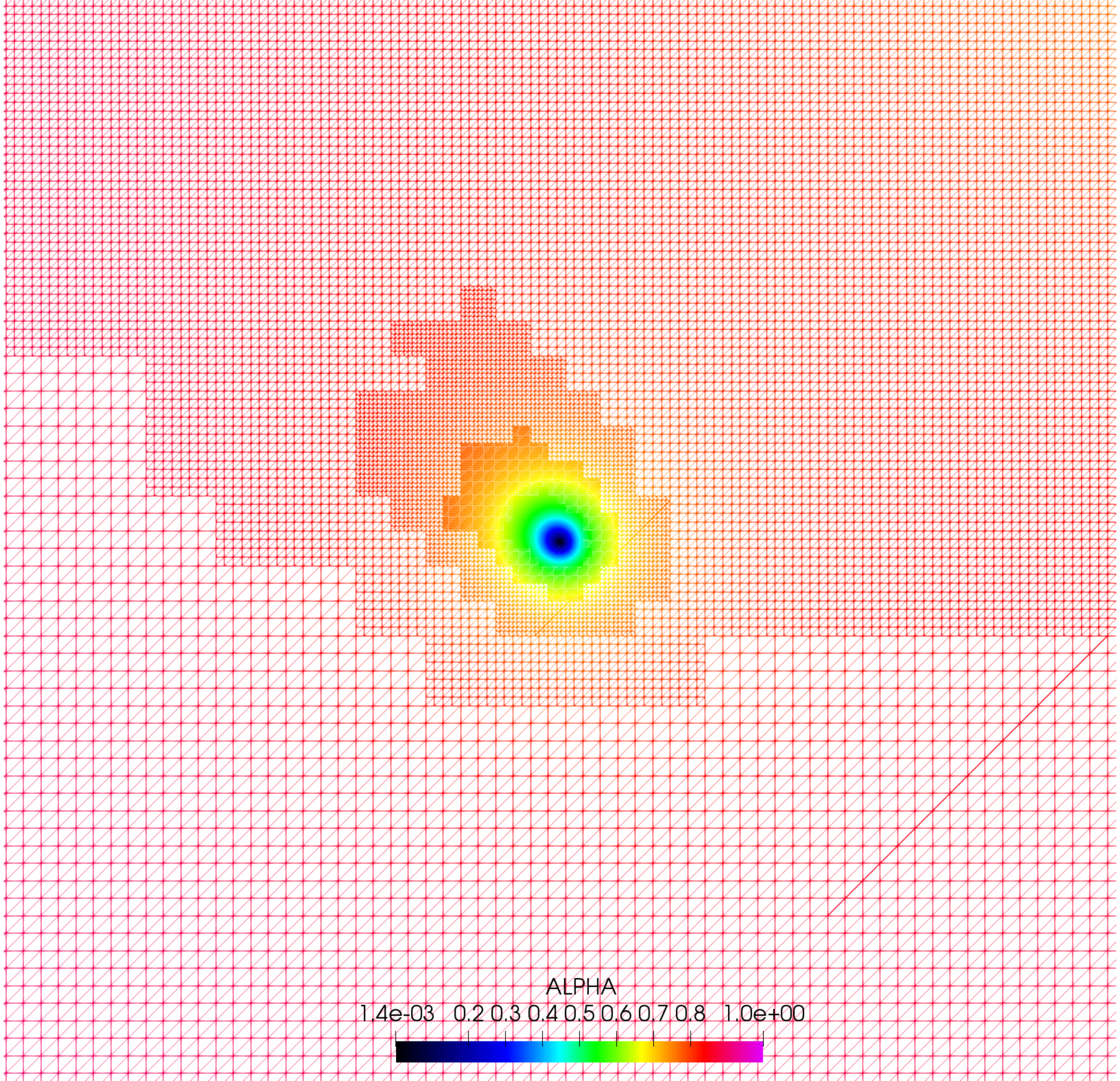
BBH ($q = 10$)



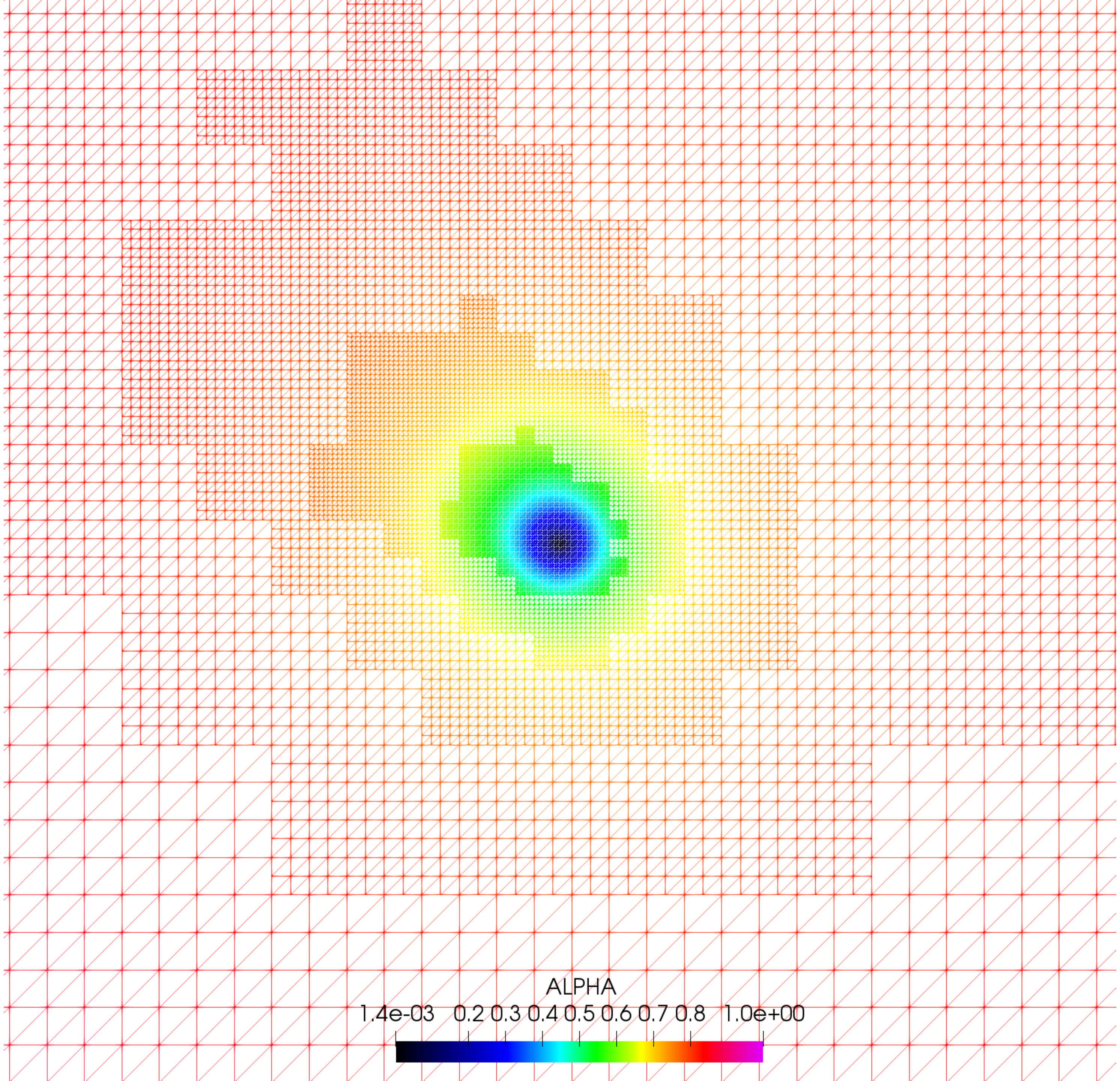
BBH $(q = 10)$



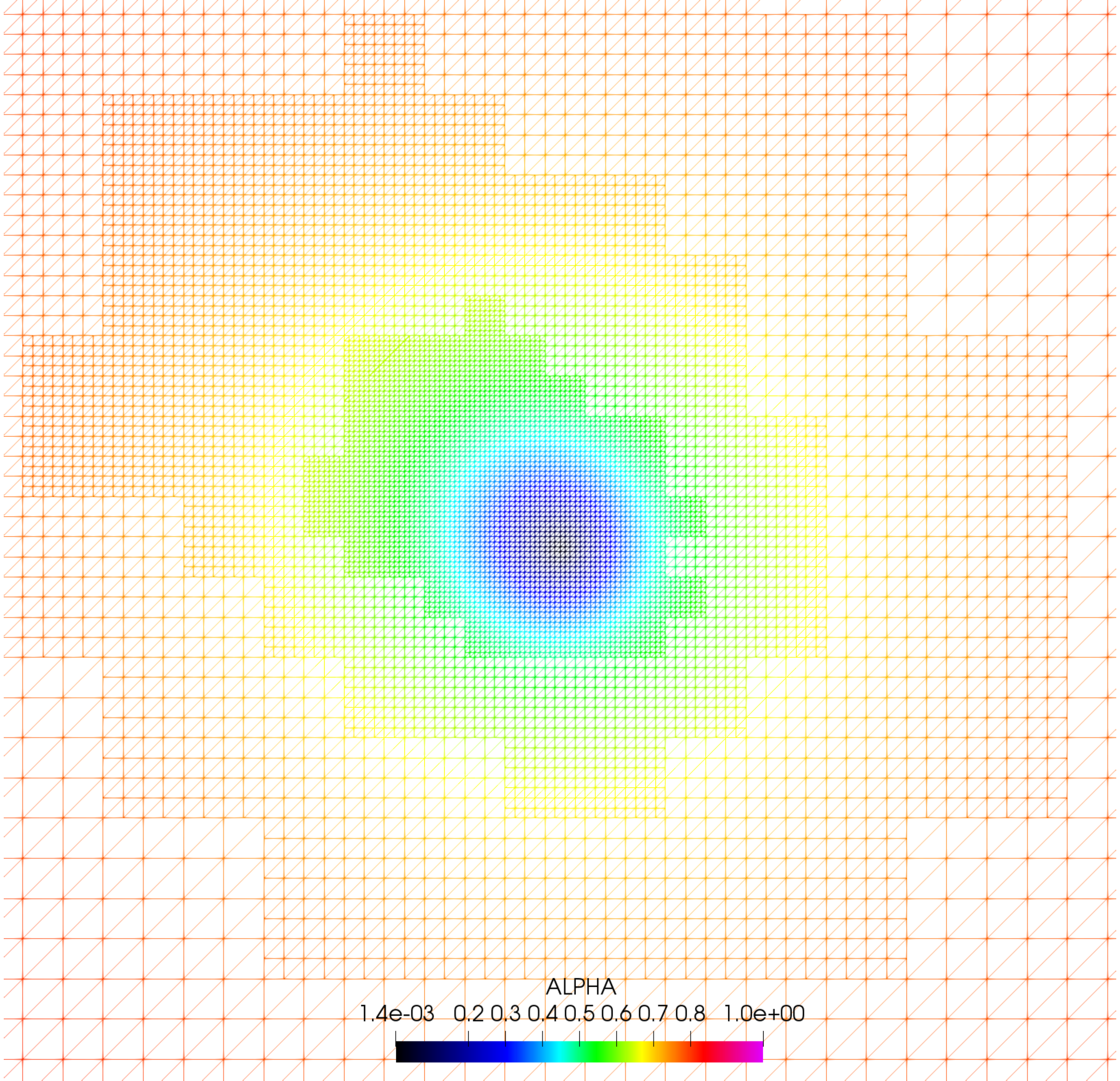
BBH ($q = 10$)



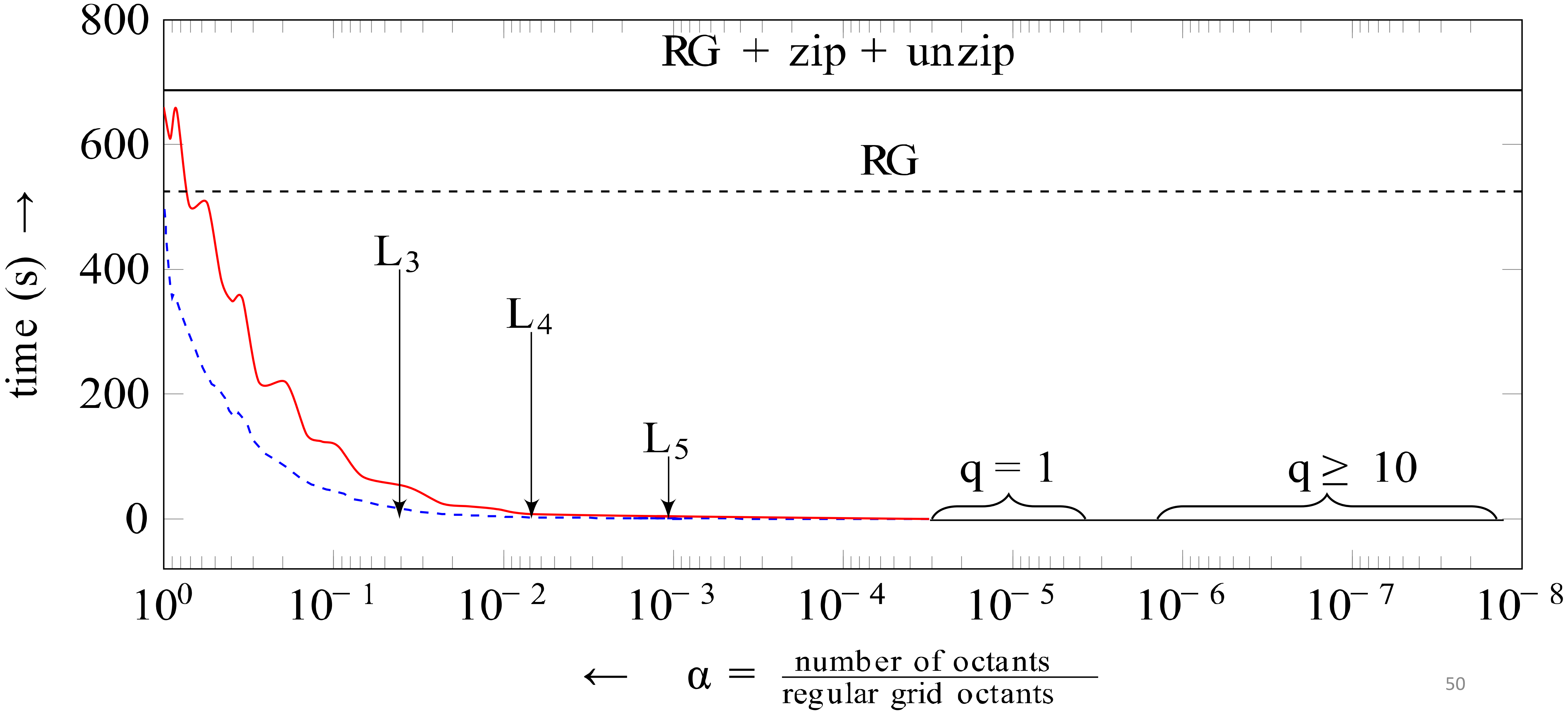
BBH ($q = 10$)



BBH ($q = 10$)



Need for true Adaptivity



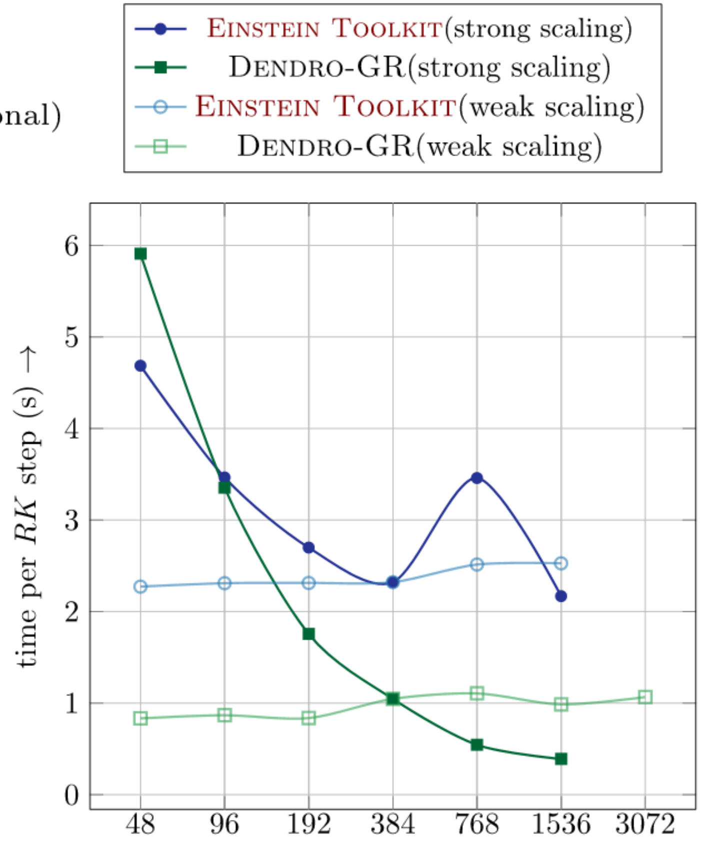
Performance & Scalability

- Evaluated at Stampede2 & Comet (XSEDE)
- Large-scale scalability on Titan at ORNL
  - Cray XK7 with 18,688 nodes with Nvidia K80s.

ET Comparison

Dendro-GR

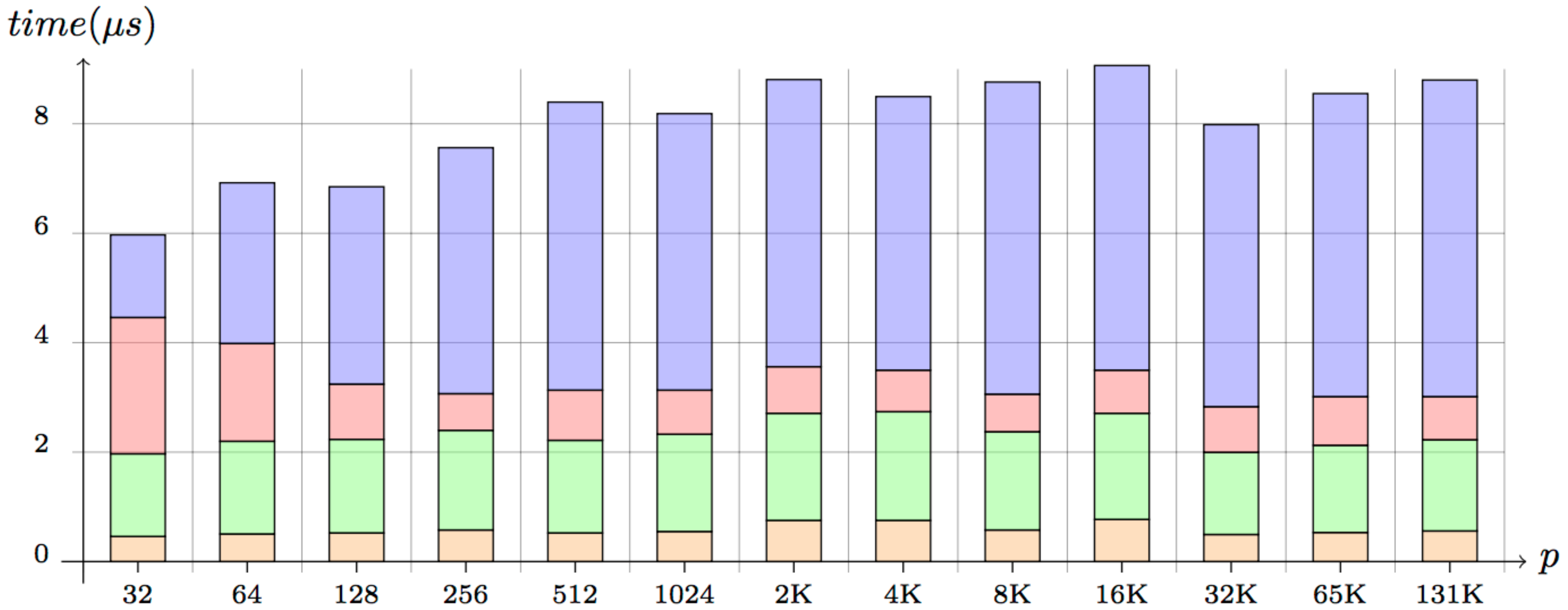
total dofs ↑ \ cores →	48	96	192	384	768	1536	3072	dofs (along the diagonal) per core ↑
$768M$	35.46	–	12.34	6.88	4.01	2.06	1.06	$250K$
$384M$	19.22	11.08	6.22	3.20	1.64	0.98	0.56	$125K$
$192M$	11.23	6.59	3.67	1.92	1.10	0.69	0.38	$62K$
$96M$	5.90	3.35	1.75	1.04	0.54	0.39	–	$31K$
$48M$	2.98	1.57	0.83	0.61	0.38	–	–	$15K$
$24M$	1.58	0.86	0.51	0.35	–	–	–	$8K$
$12M$	0.83	0.55	0.37	–	–	–	–	$4K$



Einstein Toolkit

total dofs ↑ \ cores →	48	96	192	384	768	1536	3072	dofs (along the di per core ↑
$768M$	23.38	12.65	7.64	4.93	3.71	2.95	–	$250K$
$384M$	12.44	7.51	4.87	3.48	2.92	2.52	–	$125K$
$192M$	7.13	4.76	3.45	2.74	2.51	2.31	–	$62K$
$96M$	4.68	3.46	2.69	2.32	3.45	2.16	–	$31K$
$48M$	3.36	2.68	2.31	2.08	2.12	2.08	–	$15K$
$24M$	2.67	2.31	2.08	1.94	2.07	2.04	–	$8K$
$12M$	2.27	2.06	1.93	1.88	2.03	2.07	–	$4K$

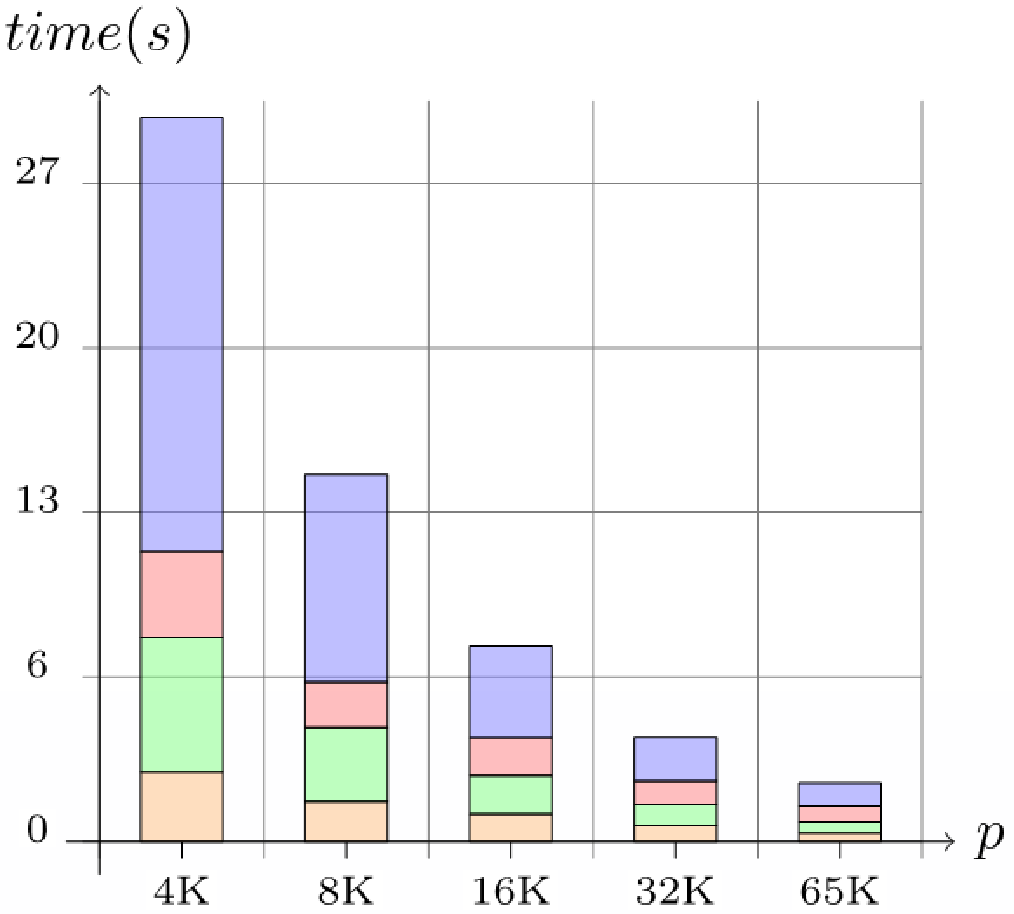
Weak Scaling



	32	64	128	256	512	1024	2K	4K	8K	16K	32K	65K	131K
communication	1.65	3.21	3.94	4.91	5.75	5.52	5.73	5.46	6.23	6.08	5.63	6.05	6.32
unzip	2.72	1.95	1.10	0.73	1.00	0.88	0.93	0.83	0.75	0.86	0.91	0.97	0.86
rhs	1.65	1.85	1.87	1.99	1.85	1.94	2.14	2.17	1.96	2.12	1.64	1.74	1.82
derivatives	0.50	0.55	0.57	0.63	0.57	0.60	0.82	0.82	0.63	0.84	0.54	0.58	0.61
Total time	6.5	7.6	7.5	8.2	9.2	8.9	9.6	9.3	9.6	9.9	8.7	9.3	9.6
Total dofs (zipped)	48.6M	92.9M	187.7M	342.1M	694.0M	1.4B	2.8B	5.6B	11.0B	21.7B	46.4B	93.9B	205.8B

Scaling test performed on Titan *with 18 levels of refinement.*

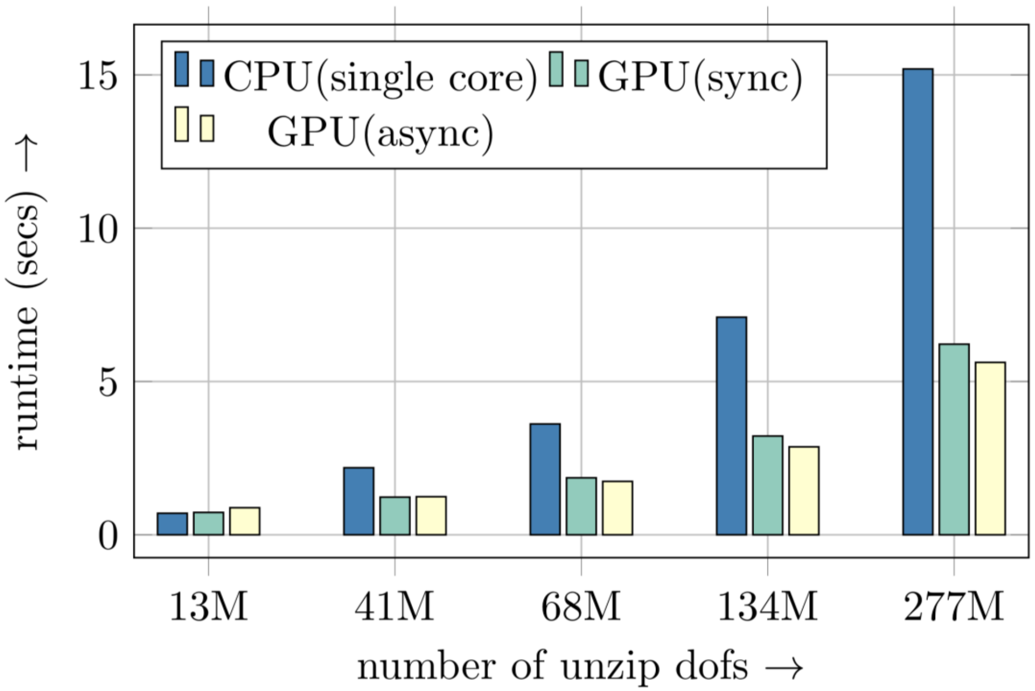
Strong Scaling



	4K	8K	16K	32K	65K
communication	18.31	8.76	3.86	1.85	0.99
unzip	3.62	1.92	1.59	1.00	0.66
rhs	5.68	3.12	1.64	0.88	0.45
derivatives	2.93	1.69	1.15	0.67	0.37
Total time (s)	30.5	15.5	8.2	4.4	2.5

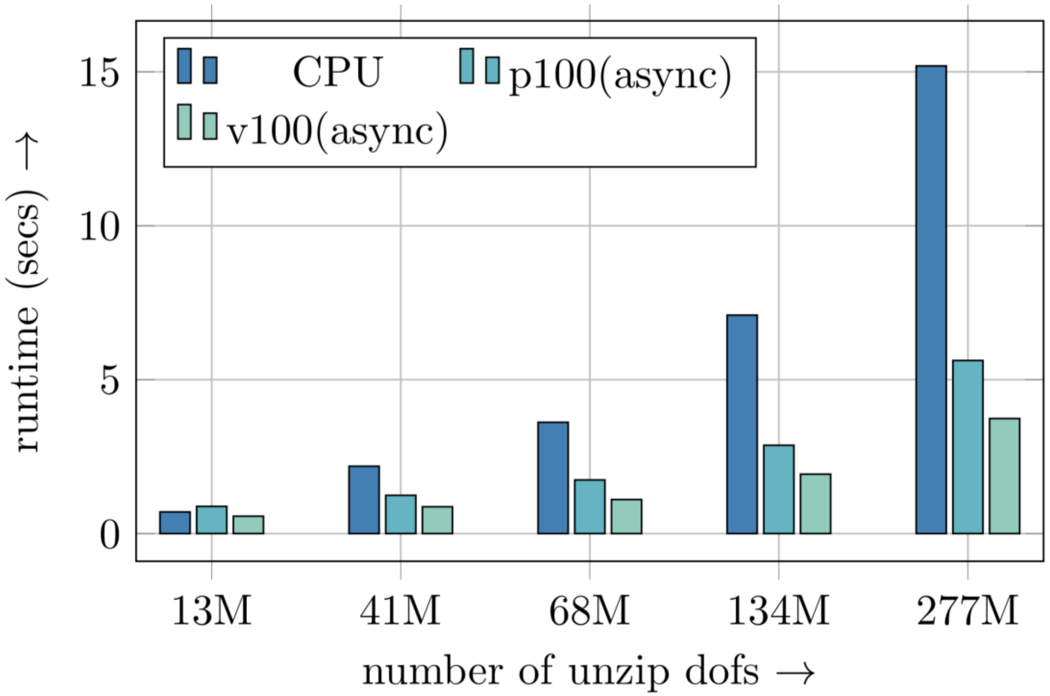
GPU Performance

Nvidia P100 GPUs



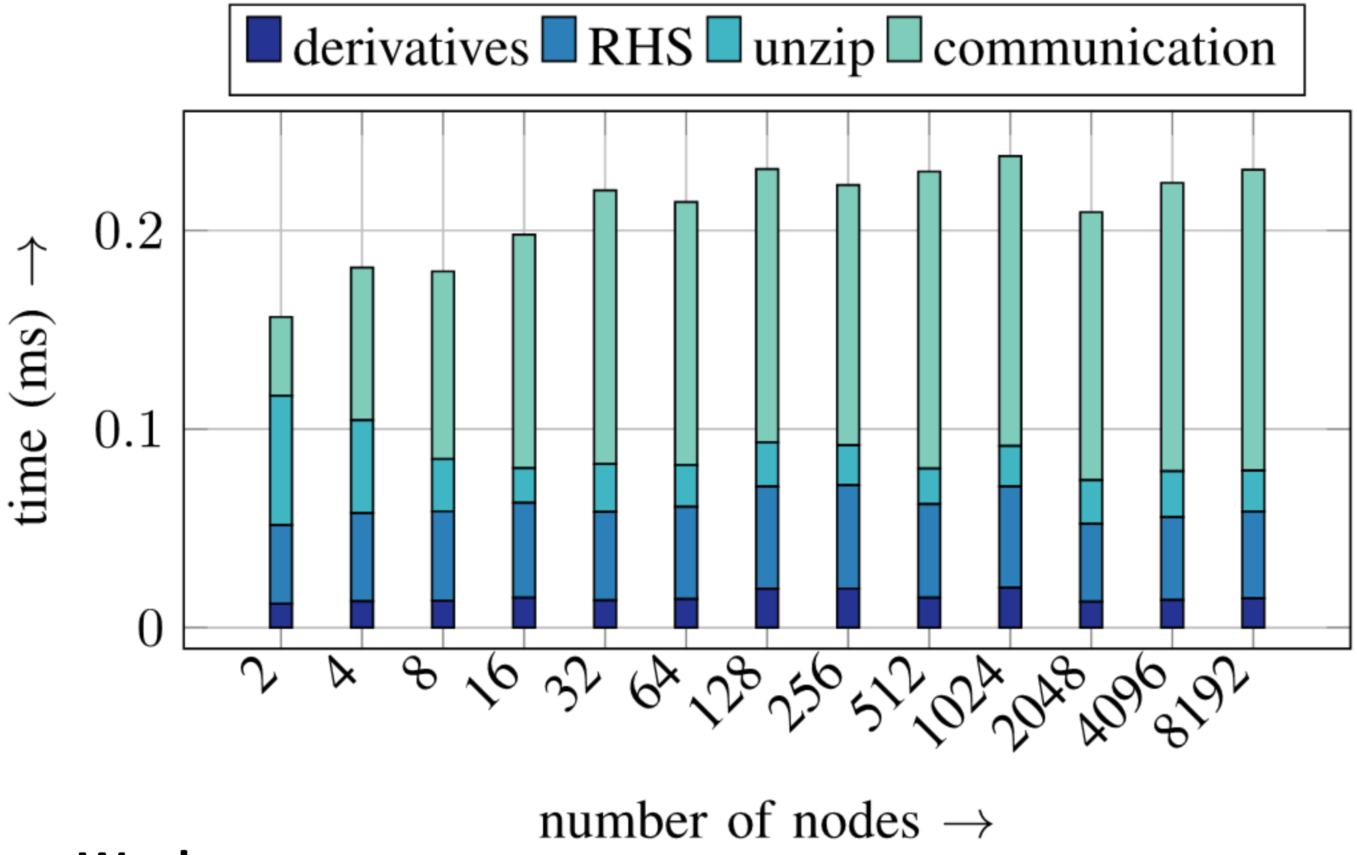
GPUs on Comet

Nvidia V100 GPUs

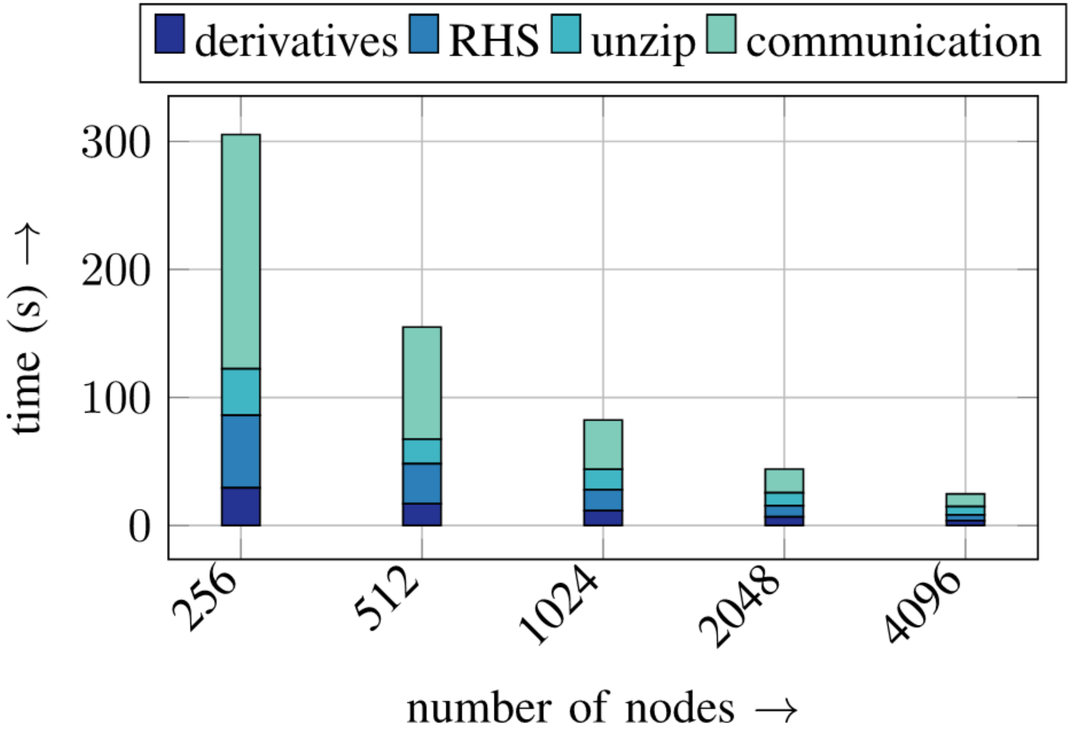


“Early Access” GPUs for Coral Sierra (LLNL)

Hybrid Code Scalability



Weak



Strong

Open Source

- Dendro-GR is available on Github.
  - https://github.com/paralab/Dendro-GR
  - `git clone git@github.com:paralab/Dendro-GR.git`
- Dendro-GR builds with CMake. Requires MPI and GSL. CUDA optional for GPU support.
- Public version
  - Wave Equation
  - Maxwell Equations* (Baumgarte's BSSN-like formulation)
  - BSSN Equations
- Support FEM in addition to FD
  - DG support coming soon
  - Extensively used for CFD
- For more details Fernando+ 1807.06128

Summary

- Dendro: Octree + Wavelet Adaptive Multiresolution (WAMR)
- Scaling to 10^5 cores *with refinement*
- Conventional finite difference/finite volume numerical methods
- Applications: Relativistic fluids and the BSSN equations
- Currently testing binary black hole simulations
- Future work
  - IMRIs
  - Neutron stars