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

\[
\alpha = \frac{\text{number of octants}}{\text{regular grid octants}}
\]

\[
q = 1 \quad q \geq 10
\]
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
• 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

- Distributed memory
- CPU
- Accelerator
- Shared memory
- Vectorization

- mpi
- CUDA, OpenCL
- CUDA, OpenMP
- SSE, AVX
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

![Octree Diagram](image-url)
SFCs for Partitioning Data

\textit{level=5, \#partitions=4} \hspace{1cm} \textit{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
  • 4th order finite differences
  • Kreiss-Oliger dissipation
Computational Methods

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

• Einstein Equations
  • BSSN formulation
  • 4\textsuperscript{th} order finite differences
  • Kreiss-Oliger dissipation

Actually... nothing special
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
\[ u_n \rightarrow \text{blocks} \rightarrow \text{RK stages} \rightarrow u_{n+1} \]

\[ \text{initial } u_0 \rightarrow \text{unzip} \rightarrow \text{blocks} \rightarrow \text{unzip} \rightarrow u_{n+1} \rightarrow \text{zip} \]
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

```python
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 ]
```
```
# 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]')
```
// Dendro: {{{
// Dendro: original ops: 678611

double 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));

// Dendro: reduced ops: 4602

// Dendro: }}}
Automatic Code Generation - vectorization

```c
// 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
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!
- 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

$\alpha = \frac{\text{number of octants}}{\text{regular grid octants}}$

$q = 1$

$q \geq 10$
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**

<table>
<thead>
<tr>
<th>total dofs</th>
<th>768M</th>
<th>384M</th>
<th>192M</th>
<th>96M</th>
<th>48M</th>
<th>24M</th>
<th>12M</th>
</tr>
</thead>
<tbody>
<tr>
<td>cores</td>
<td>48</td>
<td>96</td>
<td>192</td>
<td>384</td>
<td>768</td>
<td>1536</td>
<td>3072</td>
</tr>
<tr>
<td>dofs (along the diagonal)</td>
<td>35.46</td>
<td>12.34</td>
<td>6.88</td>
<td>4.01</td>
<td>2.06</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>per core</td>
<td>1.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**EINSTEIN TOOLKIT**

<table>
<thead>
<tr>
<th>total dofs</th>
<th>768M</th>
<th>384M</th>
<th>192M</th>
<th>96M</th>
<th>48M</th>
<th>24M</th>
<th>12M</th>
</tr>
</thead>
<tbody>
<tr>
<td>cores</td>
<td>48</td>
<td>96</td>
<td>192</td>
<td>384</td>
<td>768</td>
<td>1536</td>
<td>3072</td>
</tr>
<tr>
<td>dofs (along the diagonal)</td>
<td>250K</td>
<td>125K</td>
<td>62K</td>
<td>31K</td>
<td>15K</td>
<td>8K</td>
<td>4K</td>
</tr>
</tbody>
</table>

- **EINSTEIN TOOLKIT** (strong scaling)
- **DENDRO-GR** (strong scaling)
- **EINSTEIN TOOLKIT** (weak scaling)
- **DENDRO-GR** (weak scaling)
Weak Scaling

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

![Graph showing strong scaling with time in seconds on the y-axis and process count (p) on the x-axis. The graph includes bars for communication, unzip, rhs, derivatives, and total time, with different data points for 4K, 8K, 16K, 32K, and 65K processes.

<table>
<thead>
<tr>
<th></th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
<th>65K</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>18.31</td>
<td>8.76</td>
<td>3.86</td>
<td>1.85</td>
<td>0.99</td>
</tr>
<tr>
<td>unzip</td>
<td>3.62</td>
<td>1.92</td>
<td>1.59</td>
<td>1.00</td>
<td>0.66</td>
</tr>
<tr>
<td>rhs</td>
<td>5.68</td>
<td>3.12</td>
<td>1.64</td>
<td>0.88</td>
<td>0.45</td>
</tr>
<tr>
<td>derivatives</td>
<td>2.93</td>
<td>1.69</td>
<td>1.15</td>
<td>0.67</td>
<td>0.37</td>
</tr>
<tr>
<td>Total time (s)</td>
<td>30.5</td>
<td>15.5</td>
<td>8.2</td>
<td>4.4</td>
<td>2.5</td>
</tr>
</tbody>
</table>
GPU Performance

Nvidia P100 GPUs

Nvidia V100 GPUs

GPUs on Comet

“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](https://github.com/paralab/Dendro-GR)
  • git clone [git@github.com:paralab/Dendro-GR.git](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](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