Dendro-GR: Massively Parallel Simulations of Binary Black Hole Intermediate-Mass-Ratio Inspirals (Tutorial)

Milinda Fernando, Hari Sundar, Hyun Lim, Eric Hirschmann, David Neilsen

Einstein Toolkit Workshop, 2018
Outline

Getting Started

SFC based partitioning

Octree construction & Balancing

Wavelet Adaptive Mesh Refinement (WAMR) & \texttt{FUNCTIONToOCTREE}

Mesh generation

\texttt{NLSM} : Non-linear Sigma Model
  \texttt{NLSM} : Symbolic code generation
  \texttt{NLSM} : Runge-Kutta solver
Installation and getting started

• Downloading DENDRO-GR: You can visit
  https://github.com/paralab/Dendro-GR
  or clone DENDRO-GR
  git clone git@github.com:paralab/Dendro-GR.git
Building DENDRO-GR

CMake options

- **DIM_2(OFF):** Turn ON to run DENDRO-GR with 2D coordinates
Building DENDRO-GR

CMake options

- **DIM_2(OFF)**: Turn ON to run DENDRO-GR with 2D coordinates
- **HILBERT_ORDERING(ON)**: Turn OFF to use Morton curve based partitioning. (By default, DENDRO-GR uses Hilbert curve for SFC based partitioning.)
Building **DENDRO-GR**

**CMake options**

- **DIM_2(OFF):** Tun ON to run **DENDRO-GR** with 2D coordinates
- **HILBERT_ORDERING(ON):** Turn OFF to use Morton curve based partitioning. (By default, **DENDRO-GR** uses Hilbert curve for SFC based partitioning.)
- **SPLITTER_SELECTION_FIX(ON):** Turn OFF to perform splitter selection in a single stage. (related to SFC based partitioning.)
### Building DENDRO-GR

<table>
<thead>
<tr>
<th>CMake options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DIM_2(OFF)</strong>: Turn ON to run DENDRO-GR with 2D coordinates</td>
</tr>
<tr>
<td><strong>HILBERT_ORDERING(ON)</strong>: Turn OFF to use Morton curve based partitioning. (By default, DENDRO-GR uses Hilbert curve for SFC based partitioning.)</td>
</tr>
<tr>
<td><strong>SPLITTER_SELECTION_FIX(ON)</strong>: Turn OFF to perform splitter selection in a single stage. (related to SFC based partitioning.)</td>
</tr>
<tr>
<td><strong>NUM_NPES_THRESHOLD(INT)</strong>: Splitter selection number of stages $\log_2 N / \log_2 NUM_NPES_THRESHOLD$</td>
</tr>
</tbody>
</table>
CMake options

- **DENDRO_VTK_BINARY(ON)**: .PVTU,.VTU files are written in binary encoding.
CMake options

- **DENDRO_VTK_BINARY(ON):** .pvtu,.vtu files are written in binary encoding.
- **DENDRO_VTK_ZLIB_COMPRES (OFF):** if ON .pvtu,.vtu files are compressed while writing them.
Building **DENDRO-GR**

### CMake options

- **DENDRO_VTK_BINARY(ON)**: .PVTU,.VTU files are written in binary encoding.
- **DENDRO_VTK_ZLIB_COMPRES (OFF)**: if ON .PVTU,.VTU files are compressed while writing them.
- **RK_SOLVER_OVERLAP_COMM_AND_COMP(ON)**: Overlaps the computation with ghost exchange (communication)
Building **DENDRO-GR**

**CMake options**

- **DENDRO_VTK_BINARY(ON)**: .PVTU, .VTU files are written in binary encoding.
- **DENDRO_VTK_ZLIB_COMPRES (OFF)**: if ON, .PVTU, .VTU files are compressed while writing them.
- **RK_SOLVER_OVERLAP_COMM_AND_COMP(ON)**: Overlaps the computation with ghost exchange (communication)
Space filling curve based partitioning

Space filling curves

Space Filling Curve (SFC) is a surjective mapping from one dimensional space to multi-dimensional space.

- SFCs can be use to enforce an ordering in multi-dimensional spaces.
- Depending on the SFCs (i.e Hilbert, Morton, Peano etc) the quality (i.e communication cost, energy consumption) of the resulting partitions varies.
- **DENDRO-GR** supports both **HILBERT & MORTON** curves. (use **HILBERT_ORDERING** flag to switch between curves)
Hilbert Vs. Morton

- Compared to Morton curve, Hilbert curve produces better partitions in terms of communication cost.

Figure: Sparsity structure of the communication matrices of Hilbert (left) and Morton (right) based partition schemes for a total mesh size of 1B nodes with 4096 mpi tasks ran on TACC’s Stampede with a tolerance value of 0.3. Note that the two matrices have different sparsity structures and Hilbert is more sparse compared to Morton communication matrix.
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, SFC) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{recurse}(\tau_c) \)
8: return

\[ T(n) = O(nk) \text{ where } k \leq \log_2(n) \]
TreeSort (pseudo code)

1: $\tau \leftarrow \Gamma$
2: for $p_i \in \tau$ do →
3: $\tau_c \leftarrow \text{bucket}(p_i)$
4: $\text{reorder}(\tau_c, SFC)$
5: for $\tau_c$ of $\tau$ do
6: if $|\tau_c| > 1$ then
7: $\text{return} \text{recurse}(\tau_c)$
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \rightarrow \text{reorder}(\tau_c, SFC) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{recurse}(\tau_c) \)
8: return
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \rightarrow \text{reorder}(\tau_c, SFC) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{return} \)
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, SFC) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \rightarrow \text{recurse}(\tau_c) \)
8: return
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do →
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, \text{SFC}) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{recurse}(\tau_c) \)
return
TreeSort (pseudo code)

1: $\tau \leftarrow \Gamma$
2: for $p_i \in \tau$ do
3: $\tau_c \leftarrow \text{bucket}(p_i)$
4: $\rightarrow \text{reorder}(\tau_c, SFC)$
5: for $\tau_c$ of $\tau$ do
6: if $|\tau_c| > 1$ then
7: $\text{return} \text{recurse}(\tau_c)$
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: \( \text{for } p_i \in \tau \text{ do} \)
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, \text{SFC}) \)
5: \( \text{for } \tau_c \text{ of } \tau \text{ do} \)
6: \( \text{if } |\tau_c| > 1 \text{ then} \)
7: \( \rightarrow \text{recurse}(\tau_c) \)
8: \( \text{return} \)
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, \text{SFC}) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{recurse}(\tau_c) \)
8: return

// Example output

T(n) = O(nk) where k ≤ \( \log_2(n) \)
TreeSort (pseudo code)

1: $\tau \leftarrow \Gamma$
2: for $p_i \in \tau$ do
3: \hspace{1em} $\tau_c \leftarrow \text{bucket}(p_i)$
4: \hspace{1em} $\rightarrow \text{reorder}(\tau_c, SFC)$
5: for $\tau_c$ of $\tau$ do
6: \hspace{2em} if $|\tau_c| > 1$ then
7: \hspace{3em} \text{return}\hspace{2em} \text{recurse}(\tau_c)$

$T(n) = \mathcal{O}(nk)$ where $k \leq \log_2(n)$
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: \textbf{for} \( p_i \in \tau \) \textbf{do}
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, SFC) \)
5: \textbf{for} \( \tau_c \) of \( \tau \) \textbf{do}
6: \textbf{if} \( |\tau_c| > 1 \) \textbf{then}
7: \( \text{recurse}(\tau_c) \)
\textbf{return} \rightarrow

\[
T(n) = \mathcal{O}(nk) \text{ where } k \leq \log_2(n)
\]
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, SFC) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{recurse}(\tau_c) \)

return

\[
T(n) = \mathcal{O}(nk) \text{ where } k \leq \log_2(n)
\]
TreeSort (pseudo code)

1: \( \tau \leftarrow \Gamma \)
2: for \( p_i \in \tau \) do
3: \( \tau_c \leftarrow \text{bucket}(p_i) \)
4: \( \text{reorder}(\tau_c, SFC) \)
5: for \( \tau_c \) of \( \tau \) do
6: if \( |\tau_c| > 1 \) then
7: \( \text{recurse}(\tau_c) \)
return

\[ T(n) = \mathcal{O}(nk) \text{ where } k \leq \log_2(n) \]
TreeSort: Flexible SFC based partitioning

Number of boundary faces between partitions increase when we increase the refinement of the curve when partitioning.

$l=1, \lambda=2, s=16$

$l=2, \lambda=1.2, s=24$

$l=3, \lambda=1.05, s=28$

$l=4, \lambda=1.01, s=30$
Weak scaling results (TreeSort)

Figure: Total execution time for Hilbert & Morton curve based partitioning scheme with, $10^6$ grain size (minimum problem size of $80M$ & maximum problem size of $1.3T$ points) in ORNL’s Titan varying number of cores from 16 to 262144.
Strong scaling results (TreeSort)

Figure: Strong scaling results for Hilbert & Morton curve based partitioning scheme with problem size of $16 \times 10^6$ points, in ORNL’s Titan varying number of cores from 16 to 1024.
Octree construction & 2:1 balancing

- **Octree construction**: Can be implemented as a variation of TreeSort where we add missing octant when we reached the leaf level of the octree.

- **2:1 balancing**: Can be implemented as 3 passes of TreeSort where adding auxiliary octants for octants which violates the balance constraint.
TreeSort in Dendro-GR

SFC_treeSort can be used for sorting, constructing and balancing octrees by changing the options parameter.

- TS_REMOVE_DUPLICATES: To remove duplicates and perform sorting of octants (use 1 pass over input octants).
TreeSort in Dendro-GR

SFC_treeSort can be used for sorting, constructing and balancing octrees by changing the options parameter.

- **TS_REMOVE_DUPLICATES**: To remove duplicates and perform sorting of octants (use 1 pass over input octants).
- **TS_CONSTRUCT_OCTREE**: To sort and construct the octree (use 1 pass over input octants)
TreeSort in Dendro-GR

SFC_treeSort can be used for sorting, constructing and balancing octrees by changing the options parameter.

- **TS_REMOVE_DUPLICATES**: To remove duplicates and perform sorting of octants (use 1 pass over input octants).
- **TS_CONSTRUCT_OCTREE**: To sort and construct the octree (use 1 pass over input octants).
- **TS_BALANCE_OCTREE**: To remove duplicates and perform sorting of octants and perform construction and balancing. (use 3 pass over input octants)

```cpp
void SFC_treeSort(T* pNodes, DendroIntL n ,std::vector<T>& pOutSorted,std::vector<T>& pOutConstruct,std::vector<T>& pOutBalanced, unsigned int pMaxDepthBit,unsigned int pMaxDepth, T& parent, unsigned int rot_id,unsigned int k, unsigned int options)
```

Fernando, Sundar, Lim, Hirschmann, Neilsen
functionToOctree in Dendro-GR

- Uses the WAMR to determine the refinement pattern.
- Generate initial octree to capture a given function (i.e. specified as std::function)
Example: FUNCTION TO OCTREE

```cpp
// std::function
default_function<double(double,double,double)> g = (const double x, const double y, const double z) {
    return sin(2*M_PI*x)*sin(2*M_PI*y)*sin(2*M_PI*z);
};

// call function to octree
function20ctree(g, tmpNodes, m_uiMaxDepth, wavelet_tol, eOrder, comm);
// remove duplicates and octree construction
SFC::parSort::SFC_treeSort(tmpNodes, balOctree, balOctree, balOctree, partition_tol, m_uiMaxDepth, rootNode, ROOT_ROTATION, 1, TS_CONSTRUCT_OCTREE, 2, comm);
std::swap(tmpNodes, balOctree);
balOctree.clear();
// octree 2:1 balancing
SFC::parSort::SFC_treeSort(tmpNodes, balOctree, balOctree, balOctree, balOctree, partition_tol, m_uiMaxDepth, rootNode, ROOT_ROTATION, 1, TS_BALANCE_OCTREE, 2, comm);
tmpNodes.clear();
```
Example: \textbf{FUNCTIONToOctree}

\[ f(x), x \in \Omega \]

\[ T_c \in \Omega \]

\[ T_b \in \Omega \]
Mesh generation

- Build neighborhood data structures on the adaptive octree to perform numerical computations.
Mesh generation

- Build neighborhood data structures on the adaptive octree to perform numerical computations.
- Assumes that the input octree is 2:1 balanced.
Mesh generation

- Build neighborhood data structures on the adaptive octree to perform numerical computations.
- Assumes that the input octree is 2 : 1 balanced.
- Includes utility functionalities (zip/unzip, ghost nodal exchange) to write numerical methods such as finite difference or finite element computations.
Mesh generation

• Build neighborhood data structures on the adaptive octree to perform numerical computations.
• Assumes that the input octree is 2 : 1 balanced.
• Includes utility functionalities (zip/unzip, ghost nodal exchange) to write numerical methods such as finite difference or finite element computations.
• Communicator expansion/shrinking: Ability to dynamically change the number of cores used (depending on the problem size)
Mesh generation

• Build neighborhood data structures on the adaptive octree to perform numerical computations.
• Assumes that the input octree is 2:1 balanced.
• Includes utility functionalities (zip/unzip, ghost nodal exchange) to write numerical methods such as finite difference or finite element computations.
• Communicator expansion/shrinking : Ability to dynamically change the number of cores used (depending on the problem size)
• Re-mesh (with dynamic load balancing) and inter-grid transfer operations.
Mesh::Mesh(std::vector<ot::TreeNode> &in, unsigned int k_s=1, unsigned int pOrder=4, MPI_Comm comm, unsigned int grainSz, double ld_tol, unsigned int sf_k=2)
Zip & Unzip
Overview of RK solver

$u_0 \rightarrow u_n \rightarrow u_{n+1}$

RK stages

initial $u_0$  unzip  blocks  unzip  zip  $u_{n+1}$

blocks
Example: Solving NLSM using DENDRO-GR

Let $\chi(x, t)$ scalar function defined on $\Omega = [a, b]^3 \subset \mathbb{R}^3$

$$\chi_{tt} = \Delta \chi - \frac{\sin(2\chi)}{\|r\|_2^2}$$

(1)

Let’s rewrite above as,

$$\phi_t = \Delta \chi - \frac{\sin(2\chi)}{\|r\|_2^2}$$

$$\chi_t = \phi$$

with outgoing (open) boundary conditions,

$$\chi_t = -\chi_t^t \nabla \chi - k(\chi - \chi_0)$$

$$\|r\|_2$$

Ref: Liebling(2004), The nonlinear sigma model with Distributed adaptive mesh refinement
import dendro
from sympy import *

# initialize
r = symbols('r')

# declare functions
chi = dendro.scalar("chi","[pp]")
phi = dendro.scalar("phi","[pp]")
d = dendro.set_first_derivative('grad') # first argument is direction
d2s = dendro.set_second_derivative('grad2') # first 2 arguments are directions
d2 = dendro.d2

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

outs = [phi_rhs, chi_rhs]
vnames = ['phi_rhs', 'chi_rhs']
dendro.generate(outs, vnames, '[pp]')
// Dendro: {{{
// Dendro: original ops: 10
// Dendro: printing temp variables

// Dendro: printing variables
//--
phi_rhs[pp] = grad2_0_0_chi[pp] + grad2_1_1_chi[pp] + grad2_2_2_chi[pp]
   - sin(2*chi[pp])/pow(r, 2);
//--
chi_rhs[pp] = phi[pp];
// Dendro: reduced ops: 10
// Dendro: }}}}
NLSM: Generated code C code

// Dendro: {{{
// Dendro: original ops: 10
// Dendro: printing temp variables

// Dendro: printing variables
//--
phi_rhs[pp] = grad2_0_0_ch[i][pp] + grad2_1_1_ch[i][pp] + grad2_2_2_ch[i][pp]
    - sin(2*chi[pp])/pow(r, 2);
//--
chi_rhs[pp] = phi[pp];
// Dendro: reduced ops: 10
// Dendro: }}}

• Note that in BSSN code generation original number of operations 667,747 reduced to 4484 by common sub-expression elimination.
Runge-Kutta Solver

1: \( M \leftarrow \text{initialize mesh} \)
2: \( u = (\phi, \chi) \leftarrow \text{initialize variables (} M \text{)} \)
3: while \( t < T \) do
   4: for \( r = 1 : 4 \) do
      5: \( B, \hat{u} \leftarrow \text{Unzip}_\text{async}(M, u) \)
      6: for \( b \in B \) do
         7: Compute derivatives
         8: Compute \( \hat{u}_{rhs}(b) \)
         9: \( u_{rhs} \leftarrow \text{Zip}(M, B, \hat{u}_{rhs}) \)
      10: RK update
      11: \( t \leftarrow t + dt \)
   12: if need remesh \( M \) then
      13: \( M' \leftarrow \text{remesh}(M) \)
      14: \( u' \leftarrow \text{Intergrid}\_\text{Transfer}(M, M', u) \)

▷ Machine generated code
▷ Machine generated code
▷ based on WAMR
Runge-Kutta Solver

1: $M \leftarrow$ initialize mesh
2: $u = (\phi, \chi) \leftarrow$ initialize variables ($M$)
3: while $t < T$ do
4:     for $r = 1 : 4$ do
5:         $B, \hat{u} \leftarrow \text{Unzip}_\text{async}(M, u)$
6:     for $b \in B$ do
7:         Compute derivatives
8:         Compute $\hat{u}_{r\text{hs}}(b)$
9:         $u_{r\text{hs}} \leftarrow \text{Zip}(M, B, \hat{u}_{r\text{hs}})$
10:    RK update
11:    $t \leftarrow t + dt$
12: if need remesh $M$ then
13:    $M' \leftarrow \text{remesh}(M)$
14:    $u' \leftarrow \text{Intergid}_\text{Transfer}(M, M', u)$
Runge-Kutta Solver

1: \[ M \leftarrow \text{initialize mesh} \]
2: \[ u = (\phi, \chi) \leftarrow \text{initialize variables} \ (M) \]
3: \[ \textbf{while} \ t < T \ \textbf{do} \]
4: \[ \textbf{for} \ r = 1 : 4 \ \textbf{do} \]
5: \[ B, \hat{u} \leftarrow \text{Unzip\_async}(M, u) \]
6: \[ \textbf{for} \ b \in B \ \textbf{do} \]
7: \[ \text{Compute derivatives} \]
8: \[ \text{Compute } \hat{u}_{\text{rhs}}(b) \]
9: \[ u_{\text{rhs}} \leftarrow \text{Zip}(M, B, \hat{u}_{\text{rhs}}) \]
10: \[ \text{RK update} \]
11: \[ t \leftarrow t + dt \]
12: \[ \textbf{if} \ \text{need remesh} \ M \ \textbf{then} \]
13: \[ M' \leftarrow \text{remesh}(M) \]
14: \[ u' \leftarrow \text{Intergid\_Transfer}(M, M', u) \]

- Machine generated code
- Machine generated code
- Based on WAMR
Runge-Kutta Solver

1: $M \leftarrow$ initialize mesh
2: $u = (\phi, \chi) \leftarrow$ initialize variables ($M$)
3: while $t < T$ do →
4:   for $r = 1 : 4$ do
5:     $B, \hat{u} \leftarrow$ Unzip\_async($M, u$)
6:       for $b \in B$ do
7:       Compute derivatives
8:       Compute $\hat{u}_{rhs}(b)$
9:       $u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)
10:      RK update
11:     $t \leftarrow t + dt$
12:   if need remesh $M$ then → based on WAMR
13:     $M' \leftarrow$ remesh($M$)
14:     $u' \leftarrow$ Intergrid\_Transfer($M, M', u$)
Runge-Kutta Solver

1. $M \leftarrow$ initialize mesh
2. $u = (\phi, \chi) \leftarrow$ initialize variables ($M$)
3. while $t < T$ do
4.  
5.  
6.  
7.  
8.  
9.  
10.  
11. if need remesh $M$ then
12.  
13.  
14.  

# Machine generated code

$B, \hat{u} \leftarrow$ Unzip$_{async}(M, u)$

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)

$\rightarrow B, \hat{u} \leftarrow$ Unzip$_{async}(M, u)$

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)

$\rightarrow B, \hat{u} \leftarrow$ Unzip$_{async}(M, u)$

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)

$\rightarrow B, \hat{u} \leftarrow$ Unzip$_{async}(M, u)$

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)

$\rightarrow B, \hat{u} \leftarrow$ Unzip$_{async}(M, u)$

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)

$\rightarrow B, \hat{u} \leftarrow$ Unzip$_{async}(M, u)$

$\hat{u}_{rhs}(b)$

$u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)
Runge-Kutta Solver

1: \[ M \leftarrow \text{initialize mesh} \]
2: \[ u = (\phi, \chi) \leftarrow \text{initialize variables (} M \text{)} \]
3: \[ \text{while } t < T \text{ do} \]
4: \[ \text{for } r = 1 : 4 \text{ do} \]
5: \[ B, \hat{u} \leftarrow \text{Unzip}_\text{async}(M, u) \rightarrow \]
6: \[ \text{for } b \in B \text{ do} \]
7: \[ \text{Compute derivatives} \]
8: \[ \text{Compute } \hat{u}_{\text{rhs}}(b) \]
9: \[ u_{\text{rhs}} \leftarrow \text{Zip}(M, B, \hat{u}_{\text{rhs}}) \]
10: \[ \text{RK update} \]
11: \[ t \leftarrow t + dt \]
12: \[ \text{if need remesh } M \text{ then} \] \[ \text{based on WAMR} \]
13: \[ M' \leftarrow \text{remesh}(M) \]
14: \[ u' \leftarrow \text{Intergid\_Transfer}(M, M', u) \]
Runge-Kutta Solver

1: $M \leftarrow \text{initialize mesh}$
2: $u = (\phi, \chi) \leftarrow \text{initialize variables (} M \text{)}$
3: while $t < T$ do
4: for $r = 1 : 4$ do
5: $\hat{M}, \hat{u} \leftarrow \text{Unzip\_async}(M, u)$
6: for $b \in \hat{M}$ do
7: Compute derivatives ▶ Machine generated code
8: Compute $\hat{u}_{rhs}(b)$ ▶ Machine generated code
9: $u_{rhs} \leftarrow \text{Zip}(M, B, \hat{u}_{rhs})$
10: RK update
11: $t \leftarrow t + dt$
12: if need remesh $M$ then ▶ based on WAMR
13: $M' \leftarrow \text{remesh}(M)$
14: $u' \leftarrow \text{Intergid\_Transfer}(M, M', u)$
Runge-Kutta Solver

1: \[ M \leftarrow \text{initialize mesh} \]
2: \[ u = (\phi, \chi) \leftarrow \text{initialize variables} \ (M) \]
3: \[ \textbf{while } t < T \textbf{ do} \]
4: \[ \textbf{for } r = 1 : 4 \textbf{ do} \]
5: \[ B, \hat{u} \leftarrow \text{Unzip\_async}(M, u) \]
6: \[ \textbf{for } b \in B \textbf{ do} \]
7: \[ \text{Compute derivatives} \]
8: \[ \rightarrow \text{Compute } \hat{u}_{rhs}(b) \]
9: \[ u_{rhs} \leftarrow \text{Zip}(M, B, \hat{u}_{rhs}) \]
10: \[ \text{RK update} \]
11: \[ t \leftarrow t + dt \]
12: \[ \textbf{if need remesh } M \textbf{ then} \]
13: \[ M' \leftarrow \text{remesh}(M) \]
14: \[ u' \leftarrow \text{Intergid\_Transfer}(M, M', u) \]
Runge-Kutta Solver

1: $M \leftarrow$ initialize mesh
2: $u = (\phi, \chi) \leftarrow$ initialize variables ($M$)
3: while $t < T$ do
4:   for $r = 1 : 4$ do
5:       $B, \hat{u} \leftarrow$ Unzip\_async($M, u$)
6:       for $b \in B$ do
7:           Compute derivatives
8:           Compute $\hat{u}_{\text{rhs}}(b)$  \hspace{1cm} \triangleright \text{Machine generated code}
9:       $u_{\text{rhs}} \leftarrow$ Zip($M, B, \hat{u}_{\text{rhs}}$)
10: RK update
11: $t \leftarrow t + dt$
12: if need remesh $M$ then  \hspace{1cm} \triangleright \text{based on WAMR}
13:   $M' \leftarrow$ remesh($M$)
14:   $u' \leftarrow$ Intergid\_Transfer($M, M', u$)
Runge-Kutta Solver

1: $M \leftarrow$ initialize mesh
2: $u = (\phi, \chi) \leftarrow$ initialize variables ($M$)
3: while $t < T$ do
4:     for $r = 1 : 4$ do
5:         $B, \hat{u} \leftarrow \text{Unzip}_\text{async}(M, u)$
6:         for $b \in B$ do
7:             Compute derivatives
8:             Compute $\hat{u}_{rhs}(b)$
9:             $u_{rhs} \leftarrow \text{Zip}(M, B, \hat{u}_{rhs})$
10:             $\rightarrow$ RK update
11:     $t \leftarrow t + dt$
12:     if need remesh $M$ then
13:         $M' \leftarrow \text{remesh}(M)$
14:         $u' \leftarrow \text{Intergid\_Transfer}(M, M', u)$

Machine generated code

$\rightarrow$ based on WAMR
Runge-Kutta Solver

1. $M \leftarrow$ initialize mesh
2. $u = (\phi, \chi) \leftarrow$ initialize variables ($M$)
3. while $t < T$ do
4.   for $r = 1 : 4$ do
5.     $B, \hat{u} \leftarrow$ Unzip\_async($M, u$)
6.     for $b \in B$ do
7.       Compute derivatives
8.       Compute $\hat{u}_{rhs}(b)$
9.     $u_{rhs} \leftarrow$ Zip($M, B, \hat{u}_{rhs}$)
10.   RK update
11.   $t \leftarrow t + dt$
12. if need remesh $M$ then
13.     $M' \leftarrow$ remesh($M$)
14.     $u' \leftarrow$ Intergid\_Transfer($M, M', u$)

$\triangleright$ Machine generated code
$\triangleright$ Machine generated code
$\triangleright$ based on WAMR
Runge-Kutta Solver

1: \( M \leftarrow \text{initialize mesh} \)

2: \( u = (\phi, \chi) \leftarrow \text{initialize variables (} M) \)

3: while \( t < T \) do

4: for \( r = 1 : 4 \) do

5: \( B, \hat{u} \leftarrow \text{Unzip\_async}(M, u) \)

6: for \( b \in B \) do

7: Compute derivatives \( \triangleq \text{Machine generated code} \)

8: Compute \( \hat{u}_{rhs}(b) \) \( \triangleq \text{Machine generated code} \)

9: \( u_{rhs} \leftarrow \text{Zip}(M, B, \hat{u}_{rhs}) \)

10: RK update

11: \( t \leftarrow t + dt \) \( \rightarrow \)

12: if need remesh \( M \) then \( \triangleright \text{based on WAMR} \)

13: \( M' \leftarrow \text{remesh}(M) \)

14: \( u' \leftarrow \text{Intergid\_Transfer}(M, M', u) \)
Runge-Kutta Solver

1: $M \leftarrow \text{initialize mesh}$
2: $u = (\phi, \chi) \leftarrow \text{initialize variables (} M \text{)}$
3: while $t < T$ do
4:     for $r = 1 : 4$ do
5:         $B, \hat{u} \leftarrow \text{Unzip\_async}(M, u)$
6:             for $b \in B$ do
7:                 Compute derivatives $\text{\triangledown}$
8:             Compute $\hat{u}_{\text{rhs}}(b)$ $\triangledown$
9:         $u_{\text{rhs}} \leftarrow \text{Zip}(M, B, \hat{u}_{\text{rhs}})$
10:        RK update
11:     $t \leftarrow t + dt$
12:     if need remesh $M$ then $\triangledown$
13:         $M' \leftarrow \text{remesh}(M)$ $\triangledown$
14:         $u' \leftarrow \text{Intergid\_Transfer}(M, M', u)$
Runge-Kutta Solver

1. \( M \leftarrow \text{initialize mesh} \)
2. \( u = (\phi, \chi) \leftarrow \text{initialize variables} (M) \)
3. \[ \text{while } t < T \text{ do} \]
4. \[ \quad \text{for } r = 1 : 4 \text{ do} \]
5. \[ \quad B, \hat{u} \leftarrow \text{Unzip}_\text{async}(M, u) \]
6. \[ \quad \text{for } b \in B \text{ do} \]
7. \[ \quad \text{Compute derivatives} \] \>
8. \[ \quad \text{Compute } \hat{u}_{\text{rhs}}(b) \] \>
9. \[ u_{\text{rhs}} \leftarrow \text{Zip}(M, B, \hat{u}_{\text{rhs}}) \]
10. \[ \text{RK update} \]
11. \[ t \leftarrow t + dt \]
12. \[ \text{if need remesh } M \text{ then} \] \>
13. \[ M' \leftarrow \text{remesh}(M) \] \>
14. \[ \rightarrow u' \leftarrow \text{Intergid}\_\text{Transfer}(M, M', u) \] \>

\( \triangleright \) Machine generated code

\( \triangleright \) Machine generated code

\( \triangleright \) based on WAMR
// initial octree
function2Octree(f_init,nlsm::NLSM_NUM_VARS,varIndex,interpVars,tmpNodes,m_uiMaxDepth,nlsm::NLSM_WAVELET_TOL,nlsm::NLSM_ELE_ORDER,comm);

// 2:1 balancing of octrees.
SFC::parSort::SFC_treeSort(tmpNodes,balOct,balOct,balOct,nlsm::NLSM_LOAD_IMB_TOL,m_uiMaxDepth,
root,ROOT_ROTATION,1,TS_BALANCE_OCTREE,nlsm::NLSM_SPLIT_FIX,commActive);

// Mesh generation
ot::Mesh * mesh=new
ot::Mesh(balOct,1,nlsm::NLSM_ELE_ORDER,comm,nlsm::NLSM_DENDRO_GRAIN_SZ,nlsm::NLSM_LOAD_IMB_TOL,nlsm::NLSM_SPLIT_FIX);

// Initialize the rk solver.
ode::solver::RK4_NLSM
rk_nlsm(mesh,nlsm::NLSM_RK45_TIME_BEGIN,nlsm::NLSM_RK45_TIME_END,nlsm::NLSM_RK45_TIME_STEP_SIZE);

// solve.
rk_nlsm.rkSolve();
Linear wave equation
Non-linear sigma model (single Gaussian distribution)
Non-linear sigma model (two Gaussian distributions)
Questions ?
Thank You.