Documentation of OpenPME Code Generation

TU Dresden, Chair of Compiler Construction
Landfried Kraatz
2020-21-05
Version 1.0

Abstract

This document describes the process of generating executable C++ code using the OpenFPM library for the DSL OpenPME.

This documentation is an addition to the the OpenPME DSL.

Contents

1 Notations

Description	Notation
Vector	$\vec{x}$
Vector moved by -1 in y-dimension	$\vec{x}_{y-1}$
Field	f
Field f at position $\vec{x}$	$f(\vec{x})$
Field f at position $\vec{x}$, moved by -1 in y-dimension	$f(\vec{x}_{y-1})$
Component x of field f	f_x
Component x of field f at position $\vec{x}$	$f_x(\vec{x})$
Arbitrary component c of field f (in 3D: $c \in \{x, y, z\}$)	f_c
Spacing between mesh-nodes of mesh m in x-dimension	$h_x(m)$

2 Core language

2.1 Pre-processing

2.1.1 add_cellList

All instances of Interact are collected. For each Interact, a CellList variable is declared at the beginning of the Simulation body, associated with the particle container of the Interact instance.

2.2 Reduction rules

2.2.1 reduce_Evolve

Transforms Evolve into ParticleLoop. Evolve is a high-level abstraction of a ParticleLoop. Evolve.self_particle is mapped to ParticleLoop.node, Evolve.container is mapped to ParticleLoop.iterable, Evolve.body is mapped to ParticleLoop.body.

2.2.2 reduce_Interact

Iterates through all neighbors of a particle and applies the interaction. The iteration through all particles is done by a ParticleLoop. The iteration through all neighbors is done by a NeighborhoodLoop. The variable computing the distance between the two particles is labeled, so it can be referred to from elsewhere.

2.2.3 reduce_Gradient3D

Applicable for 3D meshes. Transforms the gradient expression into the corresponding algebraic term. The gradient is calculated component-wise, the component is specified by Gradient.dim. For field f discretized on mesh m for an arbitrary component c, the following equation is applied:

$$\begin{aligned}\nabla f_c = &\frac{f_c(\vec{x}_{x+1}) - f_c(\vec{x}_{x-1})}{2 * h_x(m)} \\ &+ \frac{f_c(\vec{x}_{y+1}) - f_c(\vec{x}_{y-1})}{2 * h_y(m)} \\ &+ \frac{f_c(\vec{x}_{z+1}) - f_c(\vec{x}_{z-1})}{2 * h_z(m)}\end{aligned}$$

2.2.4 reduce_GradientWithFactor3D

Applicable for 3D meshes. Transforms the GradientWithFactor expression into the corresponding algebraic term. For the GradientWithFactor expression, a field can be specified as a factor. This is a workaround, as long as vector algebra is not fully integrated into the language. The gradient is calculated component-wise, the component is specified by Gradient.dim. For field f discretized on mesh m for an arbitrary component c and field g as factor, the following equation is applied:

$$\begin{aligned}(g\nabla)f_c = g_x(\vec{x}) * \frac{f_c(\vec{x}_{x+1}) - f_c(\vec{x}_{x-1})}{2 * h_x(m)} \\ + g_y(\vec{x}) * \frac{f_c(\vec{x}_{y+1}) - f_c(\vec{x}_{y-1})}{2 * h_y(m)} \\ + g_z(\vec{x}) * \frac{f_c(\vec{x}_{z+1}) - f_c(\vec{x}_{z-1})}{2 * h_z(m)}\end{aligned}$$

2.2.5 reduce_Laplace3D

Applicable for 3D meshes. Transforms the Laplace-operator expression into the corresponding algebraic term. The Laplace-operator is calculated component-wise, the component is specified by Laplace.dim. For field f discretized on mesh m for an arbitrary component c, the following equation is applied:

$$\begin{aligned}\triangle f_c = \frac{f_c(\vec{x}_{x+1}) + f_c(\vec{x}_{x-1})}{h_x(m)^2} + \frac{f_c(\vec{x}_{y+1}) + f_c(\vec{x}_{y-1})}{h_y(m)^2} + \frac{f_c(\vec{x}_{z+1}) + f_c(\vec{x}_{z-1})}{h_z(m)^2} \\ -2 * (f_c(\vec{x}) * (\frac{1}{h_x(m)^2} + \frac{1}{h_y(m)^2} + \frac{1}{h_z(m)^2}))\end{aligned}$$

2.3 Switch rules

2.3.1 switch_Curl3D

Applicable for 3D meshes. Transforms the curl expression into the corresponding algebraic term. The curl is calculated component-wise, the component is specified by Curl.dim. Each component of the curl is calculated by a different algebraic term. For field f discretized on mesh m for the components x, y and z, the following equations are applied:

$$\nabla \times f_x = (f_z(\vec{x}_{y+1}) - f_z(\vec{x}_{y-1})) * \frac{0.5}{h_y(m)} - (f_y(\vec{x}_{z+1}) - f_y(\vec{x}_{z-1})) * \frac{0.5}{h_z(m)}$$

$$\nabla \times f_y = (f_x(\vec{x}_{z+1}) - f_x(\vec{x}_{z-1})) * \frac{0.5}{h_z(m)} - (f_z(\vec{x}_{x+1}) - f_z(\vec{x}_{x-1})) * \frac{0.5}{h_x(m)}$$

$$\nabla \times f_z = (f_y(\vec{x}_{x+1}) - f_y(\vec{x}_{x-1})) * \frac{0.5}{h_x(m)} - (f_x(\vec{x}_{y+1}) - f_x(\vec{x}_{y-1})) * \frac{0.5}{h_y(m)}$$

2.3.2 switch_ExpressionStatement

Delegates to specific rules on ExpressionStatement, including

- switch_ExpressionStatement_Mesh_Assignment
- switch_ExpressionStatement_Particle_Assignment
- switch_ExpressionStatement_ParticlePosition_Assignment

2.3.3 switch_ExpressionStatement_Mesh_Assignment

Is applied to an ExpressionStatement when: 1. it contains an AssignmentExpression, 2. the left side of the ExpressionStatement is a MeshAccess, 3. the MeshAccess.variableReference refers to a Mesh (FieldContainer). The ExpressionStatement is transformed into a MeshLoop. The original ExpressionStatement is mapped to the body of the MeshLoop. Every MeshAccess in the ExpressionStatement.expression is transformed to reference to the iterator of the MeshLoop.

2.3.4 switch_ExpressionStatement_Particle_Assignment

Is applied to an ExpressionStatement when: 1. it contains an AssignmentExpression, 2. the left side of the ExpressionStatement is a ParticleAccess, 3. the ParticleAccess.variableReference refers to a Particle (FieldContainer). The ExpressionStatement is transformed into a ParticleLoop. The original ExpressionStatement is mapped to the body of the ParticleLoop. Every ParticleAccess and ParticlePositionAccess in the ExpressionStatement.expression is transformed to reference to the iterator of the ParticleLoop.

2.3.5 switch_ExpressionStatement_ParticlePosition_Assignment

Is applied to an ExpressionStatement when: 1. it contains an AssignmentExpression, 2. the left side of the ExpressionStatement is a ParticlePositionAccess, 3. the ParticlePositionAccess.variableReference refers to a Particle (FieldContainer). The ExpressionStatement is transformed into a ParticleLoop. The original ExpressionStatement is mapped to the body of the ParticleLoop. Every ParticleAccess and ParticlePositionAccess in the ExpressionStatement.expression is transformed to reference to the iterator of the ParticleLoop.

3 Cpp language

3.1 Pre-processing

3.1.1 add_particle_containers

For every ParticleLoop, ParticleLoop.iterable is copied to ParticleLoop.node.container, and to every descendant SkipIteration.container and AccessInDimension.access.containerReference.

For every NeighborhoodLoop, NeighborhoodList.iterable is copied to NeighborhoodLoop.particle.container and every descendant SkipIteration.container.

3.1.2 add_property_id

Sets a unique ID for each property in every FieldContainer.

3.1.3 add_map_resync_in_ploop

ResyncGhost and MapVectorDist (particle-mapping) have to be inserted before accessing particles, in case particle positions were changed since the last resync and particle-mapping. This script inserts ResyncGhost and MapVectorDist after every ParticleLoop containing a write-access to a particle position, i.e. an AssignmentExression with ParticlePositionAccess as left-expression. The generated code is not optimal for multiple consecutive write-accesses, because only the last resync and particle-mapping is necessary in that case.

3.1.4 generate_access_in_dimension_particleaccess

ExpressionStatements containing a ParticleAccess or a ParticlePositionAccess are multiplicated according to the dimension of the accessed property. Every contained ParticleAccess is converted into an AccessInDimension. VariableReference.ndim and ParticlePositionAccess.ndim are set according to AccessInDimension.ndim.

3.1.5 generate_access_in_dimension_meshaccess

ExpressionStatements containing a MeshAccess are multiplicated according to the dimension of the accessed property. Every contained MeshAccess is converted into an AccessInDimension. VariableReference.ndim is set according to AccessInDimension.ndim.

3.2 Reduction rules

3.2.1 reduce_TimeLoop

Transforms TimeLoop into for-loop. The iteration-counter time˙step is initialized with TimeLoop.start and the condition is set to $time\_step < TimeLoop.stop$. TimeLoop.body is copied to the for-loop body.

3.2.2 reduce_ParticleLoop_VectorDist

Transforms ParticleLoop into while-loop. The while-loop iterates over an iterator, which is received from the method getDomainIterator() of ParticleLoop.iterator. A unique name is generated for the iterator-variable. The particle that is processed in each iteration is stored in a separate variable. ParticleLoop.body is copied to the while-loop body.

3.2.3 reduce_NeighborhoodLoop

Transforms NeighborhoodLoop into while-loop. The while-loop iterates over an iterator, which is copied from NeighborhoodLoop.iterator. The neighbor-particle that is processed in each iteration is stored in a separate variable. NeighborhoodLoop.particle is mapped to the neighbor-particle variable. NeighborhoodLoop.body is copied to the while-loop body.

3.2.4 reduce_MeshLoop

Transforms MeshLoop into while-loop. The while-loop iterates over an iterator, which is received from the method getDomainIterator() of MeshLoop.iterator. A unique name is generated for the iterator-variable. The mesh-node that is processed in each iteration is stored in a separate variable. MeshLoop.body is copied to the while-loop body.

3.2.5 reduce_StencilMeshLoop

Transforms StencilMeshLoop into while-loop. This is an experimental feature and requires revision. The while-loop iterates over an iterator, which is received from the method getDomainIteratorStencil() of StencilMeshLoop.iterator. A unique name is generated for the iterator-variable. StencilMeshLoop.body is copied to the while-loop body. The mesh-node and the neighboring nodes that are processed in each iteration are stored in separate variables. The corresponding concept in openPME is the MoveKey concept. So far, there exists no mapping of MoveKey using a stencil. That implies, the neighborboring nodes have to be referenced using their variable name *positionX*.

3.2.6 reduce_SkipIteration

Transforms SkipIteration into two statements. The first increments the iterator of the for loop, which is retrieved from SkipIteration.container. The second is the C++ continue-statement.

3.2.7 reduce_MapVectorDist

Particles have to be re-mapped after moving, which is done by the *map()* method. This rule transforms MapVectorDist into the corresponding C++ statement. The FieldContainer reference is copied from MapVectorDist.vector.

3.2.8 reduce_ResyncGhost

Ghost layers have to be synced after particles moved, which is done by the *ghost_get(property1, ...)* method. This rule transforms ResyncGhost into the corresponding C++ statement. The FieldContainer is copied from ResyncGhost.fieldContainer, the properties are copied from ResyncGhost.properties.

3.2.9 reduce_CreateCellList

A CellList has to be initialized before usage, which is done by the *getCellList()* method. This rule transforms CreateCellList into the corresponding C++ statement. The FieldContainer is copied from CreateCellList.vector, the cutoff radius refers to r_cut in weave˙Initialization, the dimension is copied from Initialization.dimension.

3.2.10 reduce_UpdateCellList

If particles have changed position, it is faster to call *updateCellList()* instead of creating a new cell list. This rule transforms UpdateCellList into the corresponding C++ statement. The FieldContainer is copied from the corresponding CreateCellList.vector, the reference to the cell list variable declaration is copied from UpdateCellList.cellList.

3.2.11 reduce_CreateNeighborList

Creates an iterator for all neighbors of a particle. Callee is the CellList reference. The iterator is returned by the method *getNNIterator*. The cell list is created based on the position of the particle. The position is copied from CreateNeighborList.position.

3.2.12 reduce_IfStatement

The condition, true-block and false-block are copied from the *IfStatement* node.

3.2.13 reduce_SumPrintVariable

The statement is used to sum a variable over all processing units and print the result. For summing up the values, a vcluster object is created. The reference to the variable is copied from SumPrintVariable.ref.

3.2.14 reduce_WriteParticles

This statement writes the particles from a FieldContainer to a *vtk* file for visualization. Before writing, the ghost layer of the particle container is deleted. Then the particles are written to the file named *particles*. In the end the ghost layer is re-applied to the container.

3.2.15 reduce_Remesh

This statement creates particles at the nodes of a mesh. Particle position and properties have to be written for each dimension. The different write accesses are inserted using a MAP_SRCL macro. As we need the references to the particles and mesh FieldContainers, these are transfered inside the MAP_SRCL macro using VAR macros. Same applies to the property of the FieldContainers.

3.2.16 reduce_Copy

Copies all nodes from a mesh (Copy.source) to another mesh (Copy.target).

3.2.17 reduce_WriteMesh

This statement writes the mesh nodes from a FieldContainer to a *vtk* file named *mesh* for visualization.

3.2.18 reduce_Load

Loads data from a *vtk* file to a FieldContainer, specified by Load.container. The file name is copied from Load.file.

3.2.19 reduce_SolvePoisson

Declares and executes a linear solver to solve the Poisson equation. The Poisson equation is solved for a mesh specified in SolvePoisson.sourceMesh. The result is copied to SolvePoisson.targetMesh. The solver is executed for each component of the mesh using a for loop.

3.2.20 reduce_Spacing

Returns the spacing between nodes of a mesh. The spacing is retrieved by the method *spacing()* of the mesh iterator.

3.3 Switch rules

3.3.1 switch_VariableDeclaration

Maps to a variable declaration. Name, type and initialization are copied using COPY_SRC.

3.3.2 switch_VariableReference

There are three cases for mapping a VariableReference. The access to a vector component is mapped to the *get()* method. If comparing two instances of a vector_dist, the index is retrieved by the *getKey()* method. The standard case is resolved to an unchecked reference.

3.3.3 switch_ParticlePositionAccess

The position is retrieved by the method *getPos()*. Callee is the vector_dist container, the particle is passed as an argument. A special case is a reference to a NeighborList. Here the container is copied from the CreateCellList initialization of the variable. Accessing one component of the position is done by an index expression.

3.3.4 switch_BinaryExpression

Works as collection of all rules for subclasses of BinaryExpression. Mapped concepts are AssignmentExpression, PlusAssignmentExpression, MultiplicationExpression, SubtractionExpression, AdditionExpression, DivisionExpression, PowerExpression, EqualsExpression, GreaterExpression and ModuloExpression. In each rule, left and right expression are copied to the target concept.

3.3.5 switch_Type

Works as collection of all rules for subclasses of Type. Mapped concepts are FloatType, DoubleType, CellListType, PointType, NeighborListType and IntegerType. For the Point type, Initialization.dimension is mapped as template argument.

3.3.6 switch_TypeOfSimulation

Delegates to the respective weaving rule for simulations of type MeshBased, ParticleBased or Hybrid.

3.3.7 switch_FieldContainer

Maps particle or mesh containers. If the container is in the role of TypeOfSimulation.particle or .mesh, it is mapped to a declaration of the vector_dist or grid_dist. Otherwise (when used in the simaulation body) it is mapped to an UncheckedReference.

3.3.8 switch_DiffOpDiscretizationScheme

This rule is deprecated.

3.3.9 switch_Loop

Delagates to the different subclasses of Loop. Subclasses are TimeLoop, ParticleLoop, MeshLoop and StencilMeshLoop.

3.3.10 switch_AccessInDimension

Maps to the different cases of an access to a property of a particle or mesh node. A ParticleAccess to a vector property must be accessed using the *getProp()* method. Otherwise, the property is accessed using the *get()* method. To access neighbor mesh nodes, the *move()* method is used. Whether or not to insert the *move()* is determined using an IF macro.

3.3.11 switch_interpolation_scheme

Instantiates the interpolation object. Currently only MP4 interpolation is supported.

3.3.12 switch_Interpolate

Determines interpolation direction. Particle → Mesh interpolation uses the *p2m()* method. The reverse direction uses the *m2p()* method.

3.3.13 switch_ExpressionStatement

Inserts an ExpressionStatement and copies the expression, which is mapped during the copy process.

3.4 Weaving rules

3.4.1 weave_Initialization

Inserts initialization statements into the body of the main function. Declares the cutoff radius, domain size(box), boundary conditions and ghost.

3.4.2 weave_TypeOfSimulation

Inserts the simulation inside the body of the main function. Delegates to switch_TypeOfSimulation using the SWITCH macro.

3.4.3 weave_Discrete

Inserts the simulation body of a particle-based simulation. Declares particle containers, initialization of particles and the statements in the body.

3.4.4 weave_Hybrid

Inserts the simulation body of a hybrid simulation. Declares particle containers, mesh containers, initialization of particles and the statements in the body. Furthermore, the linear solver for the Poisson equation is initialized, if used in the simulation.

3.4.5 weave_MeshBased

Inserts the simulation body of a mesh-based simulation. Declares mesh containers, initialization of particles and the statements in the body.

3.4.6 weave_InitParticle

Initialize particles by placing them on a regular grid and setting properties to zero. Maps of all property using a MAP_SRCL macro. The reference to the particle container is passed inside the macro using a VAR macro.

3.5 CALL templates

3.5.1 poisson_prerequisites

Initializes the linear solver for solving the Poisson equation. A mesh *psi* is declared. The Poisson equation is imposed in *poisson* variable. An array *phi_solution_poisson* is declared for storing the result of the solver. The actual solver is instanciated in the variable *poisson_solver*.

3.5.2 struct_poisson

Initializes the struct *poisson_nn* used by the linear solver for the Poisson equation.