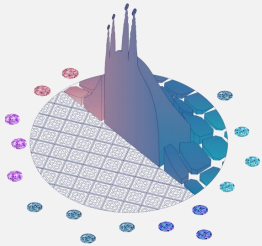
TNL-SPH: Open-source modular SPH solver for modern distributed computing platforms based on GPU accelerators

Tomáš HALADA
L. Beneš, J. Klinkovský, T. Oberhuber

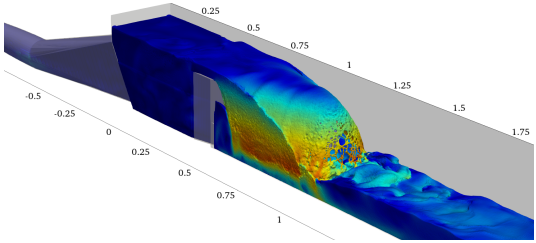
19th SPHERIC World Conference

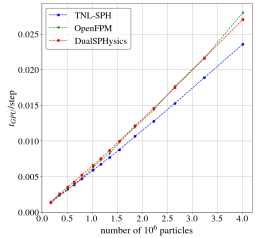
17-19 June 2025

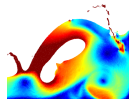


But why?

- We needed **multi-GPUs and distributed computing** for large scale 3D simulations in complex geometries
- **Particles are general computational algorithms** - we want general particle framework not only for SPH *(high order particle methods - LABFM, DC-PSE, GFDM, LDD,... image representation, optimization, MD,..)*
- Aim for better abstraction, modularity, generality and clean codebase, while trying to tip the performance







TEMPLATE NUMERICAL LIBRARY

TNL-SPH design

LAYER 1: TNL: Architecture dependent primitives

Memory management, parallel algorithms,...

LAYER 2: Particles structure

Implementation of particle coordinates, neighbor search, particle distributed communication

LAYER 3: Particles solver

Control of the simulations, connection of particles and model variables

LAYER 4: Numerical schemes

Definition of particle fields and interaction functions

LAYER 1: TNL

Template Numerical Library (TNL) - unified code for different hardware platforms, written in **C++20**, header only

- **Usage of lambda functions and parallel algorithms:**

```
const float h = 0.5;

auto v_view = v.getView();
auto func = [ = ] __cuda_callable__( int i ) mutable
{
   v_view[ i ] = i + h;
};
TNL::Algorithms::parallelFor< Device >( 0, v.getSize(), func );
```

Unified memory management, flexible parallel reduction, parallel loops, sort,...

- **Expression templates:**

$$\vec{x} = \vec{a} + 2\vec{b} + 3\vec{c}$$

```
x = a + 2 * b + 3 * c;
```

General particle automaton

A **general particle method algorithm** is 7-tuple

$$(P, G, u, f, i, e, \mathring{e})$$

consisting of two data structures

$$P := A_1 \times A_2 \times ... \times A_n \qquad \text{the particle space,}$$
$$G := B_1 \times B_2 \times ... \times B_m \qquad \text{the global variable space}$$

such that $[G \times P^*]$ is the state space of the particle methods and five functions

$$u := [G \times P^*] \times \mathbb{N} \to \mathbb{N}^* \qquad \text{the neighborhood function,}$$
$$f := G \to \{\top, \bot\} \qquad \text{the stopping condition,}$$
$$i := G \times P \times P \to P \times P \qquad \text{the interact function,}$$
$$e := G \times P \to G \times P^* \qquad \text{the evolve function,}$$
$$\mathring{e} := G \to G \qquad \text{the evolve function of global variables}$$

(J. Pahlke & I. Sbalzarini 2023)

What we understand by particle method:

$$\frac{\mathrm{d}\mathbf{x}_i}{\mathrm{d}t} = \sum_{j \in \mathcal{N}_i} \mathbf{f}_{\mathbf{x}}(\mathbf{x}_i, \mathbf{x}_j, \boldsymbol{\omega}_i, \boldsymbol{\omega}_j),$$

$$\frac{\mathrm{d}\boldsymbol{\omega}_i}{\mathrm{d}t} = \sum_{j \in \mathcal{N}_i} \mathbf{f}_{\boldsymbol{\omega}}(\mathbf{x}_i, \mathbf{x}_j, \boldsymbol{\omega}_i, \boldsymbol{\omega}_j)$$

LAYER 2: Particles structure

- **Particles structure:** Structure that handles the points and neighbor search:

```
particles.searchForNeighbors();
```

providing the loops over the particles and neighbors:

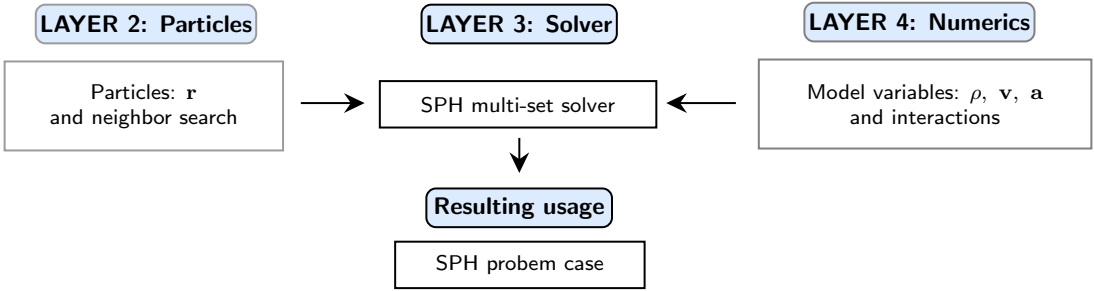
```
// Define operation preformed for every neighbor
auto pairInteraction = [=] __cuda_callable__ ( int i, int j, args.. ) mutable
{
};

// Define operation performed for every particles
auto particleLoop = [=] __cuda_callable__ ( LocalIndexType i ) mutable
{
   ParticlesType::NeighborsLoop::exec( i, r_i, searchToken, pairInteraction, args... );
};
particles.forAll( particleLoop );
```

runs in parallel even for distributed computation!

- **Particles zone:** Provides list of indices of arbitrary selected spatial subdomain.

LAYER 3: Particle solver

- Multi-set solver uses **separate particle sets** for fluid, boundary particles and open boundary patches
- **Possibility to build custom time main loop** enhanced with user defined functions



LAYER 4: Numerical solver

Numerical solvers: Defines **variables** of the scheme/model and **interaction functions** *(fluid-fluid, fluid-boundary, boundary-fluid, pre/post interaction functions)*

```
auto fluidFluid = [=] __cuda_callable__ ( LocalIndexType i, LocalIndexType j, /* args i, j,.. */ ) mutable
{
  // load data of neighboring particle
  const VectorType r_j = r_view[ j ]; const VectorType v_j = v_view[ j ]; const RealType rho_j = rho_view[ j ]
  const RealType p_j = EOS::DensityToPressure( rho_j, eosParams );

  const VectorType r_ij = r_i - r_j;
  const RealType drs = l2Norm( r_ij );
  const VectorType v_ij = v_i - v_j;
  const RealType F = KernelFunction::F( drs, h );
  const VectorType gradW = r_ij * F;

  // continuity equation
  const RealType psi = DiffusiveTerm::Psi( rho_i, rho_j, r_ij, drs, diffusiveTermsParams );
  const RealType diffTerm =  psi * ( r_ij, gradW ) * m / rho_j;
  *drho_i += ( v_ij, gradW ) * m - diffTerm;

  // momentum equation
  const RealType p_term = ( p_i + p_j ) / ( rho_i * rho_j );
  const RealType visco =  ViscousTerm::Pi( rho_i, rho_j, drs, ( r_ij, v_ij ), viscousTermTermsParams );
  *a_i += ( -1.0f ) * ( p_term + visco ) * gradW * m;
};
```

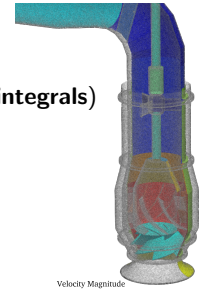
$$\frac{\mathrm{D}\rho_i}{\mathrm{D}t} = \rho_i \sum_{j=1}^{N} (\mathbf{u}_i - \mathbf{u}_j) \cdot \nabla_i W_{ij} V_j + \delta_\psi h c_0 \mathscr{D}$$

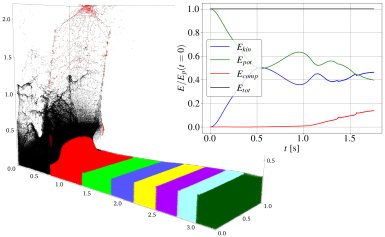
$$\frac{\mathrm{D}\mathbf{u}_i}{\mathrm{D}t} = -\frac{1}{\rho_i} \sum_{j=1}^{N} (p_i + p_j) \nabla_i W_{ij} V_j + \frac{1}{\rho_i} \sum_{j=1}^{N} \Pi_{ij} \nabla_i W_{ij} V_j + \mathbf{f}_i$$

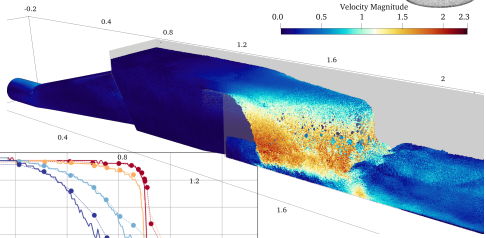
Implemented features

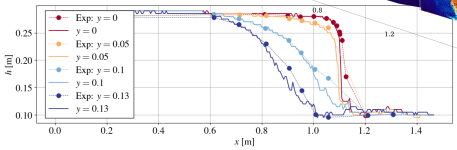
Implementing particular schemes should not be considered as a problem,...

- WCSPH (diffusive terms, various boundary conditions: DBC, MDBC, **boundary integrals**)
- Unconditionally stable SPH scheme *(Cercos-Pita, 2024)*
- RSPH, **SHTC-SPH** - unified framework for solid and fluids
- various integration schemes (Verlet, symplectic Verlet, RK45, implicit midpoint)
- open boundary conditions (inlet, outlets)
- discretization of external geometries

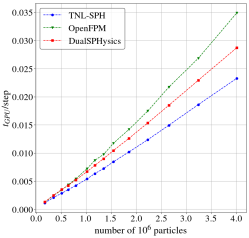




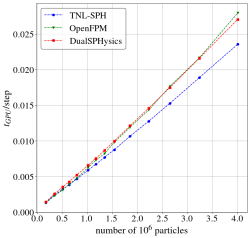




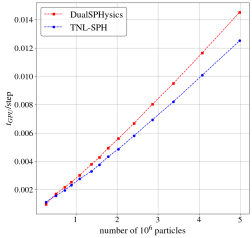
Benchmarks



NVIDIA A40



NVIDIA A100



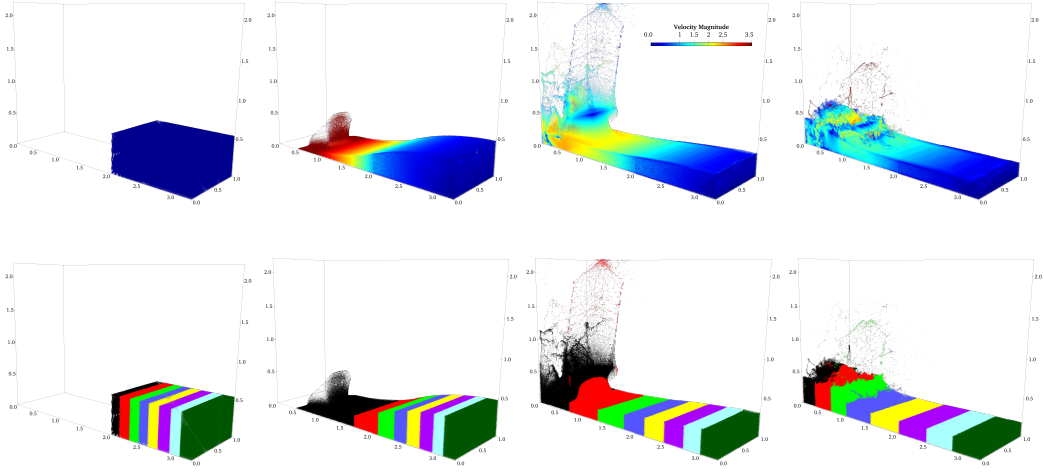
NVIDIA H100

	TNL-SPH		DualSPHysics		OpenFPM	
GPU	t_{step} [s]	MPUPS	t_{step} [s]	MPUPS	t_{step} [s]	MPUPS
A40	0,023	172,3	0,029	139,6	0,035	114,9
A100	0,024	170,2	0,027	148,2	0,028	143,4
H100	0,013	320,5	0,015	276,2		

Test case: SPHERIC testcase 2: 3D dam break, δ-WCSPH scheme with Molteni-DT, artificial viscosity, DBC, Verlet scheme, $\Delta t = const.$, resolutions $\Delta x = [0.02, 0.006]$, **table shows data for** $\Delta x = 0.006$

Benchmark setups for all solvers: `https://github.com/tomashalada/sph-benchmarks/`

Multi-GPU computations



Support of multi-GPUs and distributed computations using MPI, with 1D, 2D or 3D domain decomposition and subdomains load balancing.

Multi-GPU computations - benchmarks

Strong scaling:

N_{GPUs}	GPUPS	t_{step} [s]	*Speedup*	*Eff*
1	0,297	0,346	1,00	1,00
2	0,585	0,176	1,97	0,99
4	1,157	0,089	3,90	0,97
8	2,154	0,048	7,26	0,91

Weak scaling:

N_{GPUs}	$N_{\text{ptcs}}(10^6)$	GPUPS	t_{step} [s]	*Speedup*	*Eff*
1	15,2	0,335	0,045	1,00	1,00
2	28,5	0,622	0,046	1,98	0,99
4	53,8	1,147	0,047	3,87	0,97
8	102,8	2,136	0,048	7,54	0,94

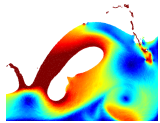
Test case: SPHERIC test case 2: 3D dam break, δ-WCSPH scheme with Molteni-DT, artificial viscosity, DBC, Verlet scheme, **weak scaling** 20×10^6 **particles per GPU, strong scaling** 10^8 **particles**, NVIDIA H100 cards

Summary

- *Iterating by particles*: The limiting factor is the memory access - to many neighborhood particles with to much data to read
- *Iterating by pairs*: The limiting factor is writing to memory - too much data to write

---

TNL-SPH: Open source, modular, header only, and pretty fast SPH solver oriented towards industrial hydrodynamics



TEMPLATE NUMERICAL LIBRARY

This research was supported by Ministry of Education, Youth and Sports of the Czech Republic under the project CZ.02.01.01/00/23 021/0008954 "Hydrodynamic machines for smart energetics"

Co-funded by the European Union

MINISTRY OF EDUCATION, YOUTH AND SPORTS