Acta Polytechnica **61**(SI):122–134, 2021
https://doi.org/10.14311/AP.2021.61.0122
© 2021 The Author(s). Licensed under a CC-BY 4.0 licence
Published by the Czech Technical University in Prague

TNL: NUMERICAL LIBRARY FOR MODERN PARALLEL ARCHITECTURES

Tomáš Oberhuber*, Jakub Klinkovský, Radek Fučík

Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, Department of Mathematics, Trojanova 13, 120 00 Praha , Czech Republic

* corresponding author: tomas.oberhuber@fjfi.cvut.cz

Abstract. We present Template Numerical Library (TNL, www.tnl-project.org) with native support of modern parallel architectures like multi–core CPUs and GPUs. The library offers an abstract layer for accessing these architectures via unified interface tailored for easy and fast development of high-performance algorithms and numerical solvers. The library is written in C++ and it benefits from template meta–programming techniques. In this paper, we present the most important data structures and algorithms in TNL together with scalability on multi–core CPUs and speed–up on GPUs supporting CUDA.

Keywords: Parallel computing, GPU, explicit schemes, semi–implicit schemes, C++ templates.

1. Introduction

TNL aims to be an efficient and user friendly library for numerical simulations. To fulfill this goal, it must support modern parallel architectures like GPUs (graphical processing units) and multi-core CPUs on one hand and offer simple and flexible interface for implementation of complex numerical solvers on the other hand. If high computational efficiency is required, we cannot follow the typical rules of the object–oriented programming that usually lead to inefficient organization of data in the computer memory and, for example, use of virtual methods may lower the performance of the final code. The design of TNL profits from advantages of the C++ templates. Especially templates specializations is a natural tool for generating specialized architecture dependent code with no run-time overhead.

There is no doubt that modern numerical libraries must support accelerators like GPUs. They provide high memory bandwidth as well as great computational power which is obtained not by high clock frequencies but by massively parallel design, which is more power efficient. Programming of GPUs is easier with the CUDA framework. Nevertheless, deep knowledge of the GPU hardware is still necessary to produce efficient code which makes the GPUs almost unavailable for many experts in numerical mathematics.

Adding support for GPUs to existing numerical libraries is nearly impossible. The GPU architecture is so different from the CPU that most of the numerical methods and algorithms must be completely rewritten. In CUDA, we deal with two different address spaces: one associated with the CPU and the other with the GPU. Communication between them is remarkably slow and it must be fully managed by the programmer. To get the maximum performance from the GPU, the programmer must take care of the organization of data in the memory, minimize the divergence of CUDA threads, deal with limited shared memory, and many other details of the GPU design [1].

In some cases, one may use the GPU for numerical computing relatively easily. An implicit time discretization of linear problems allows to construct the linear system once on the CPU, transfer it to the GPU and solve it repeatedly there by some iterative solver like CG (conjugate gradients). Such solvers require only common linear algebraic operations implemented in libraries like CUBLAS or CUSPARSE which are part of the CUDA toolkit. Difficulties arise with non-linear problems, where the linear system matrix must be updated in each time step. Transfer of the matrix from the CPU to the GPU makes any speed-up impossible. Therefore the matrix must be assembled on the GPU. This process requires a manipulation with an underlying format for sparse matrices and also efficient access to numerical mesh. Neither is trivial on the GPU.

Table 1 presents a comparison of TNL with several other HPC libraries like Cublas [2], Cusparse [3], Thrust [4], Kokkos [5], ViennaCL [6], PETSc [7] and Eigen [8]. We chose primarily libraries with support of *GPU*. The table shows which of them have modern interface in *C++* and which of them support distributed computation using *MPI*. *Parallel for*, *parallel reduction* and *scan* are emerging programming patterns in HPC which allow to write one code for different parallel architectures. *Sorting* is another common operation defined on arrays or vectors. Blas operations are well known in the HPC community. Blas is a standard library which, however, does not profit from the modern features of C++. Especially Blas level 1 operations can be better expressed with *expression templates*. *Sparse matrices* belong with no doubt to one of the most important data structures in HPC. GPUs are very sensitive to sparse matrix pattern. Re-

	TNL	Cublas	Cusparse	Thrust	Kokkos	ViennaCL	PETSc	Eigen
GPU	Yes	Yes	Yes	Yes	Yes	Yes	Weak	No
MPI	Weak	No	No	No	No	No	Yes	No
C++	Yes	No	No	Yes	Yes	Yes	No	Yes
Parallel for	Yes	No	No	Yes	Yes	No	No	No
Parallel reduction,scan	Yes	No	No	Yes	Yes	No	No	No
Sorting	No	Yes	No	Yes	Yes	No	Yes	No
Blas 1	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Blas 2	Weak	Yes	Yes	No	Yes	Yes	Yes	Yes
Blas 3	No	Yes	Yes	No	No	Yes	Yes	Yes
Expression templates	Yes	No	No	No	No	Yes	No	Yes
Sparse matrices	Yes	No	Yes	No	Yes	Yes	Yes	Yes
Linear systems solvers	Yes	No	Weak	No	No	Yes	Yes	Yes
Preconditioners	Weak	No	No	No	No	Yes	Yes	Weak
Nonlinear sys.solvers	No	No	No	No	No	Yes	Yes	No
Eigenvalues comp.	No	No	No	No	No	Yes	No	No
ODE systems solvers	Yes	No	No	No	No	No	Yes	No
Stencil computations	Yes	No	No	No	No	No	Yes	No
Unstructurted meshes	Yes	No	No	No	No	No	Yes	No
Python interface	Weak	No	No	No	No	Yes	Yes	Yes

TABLE 1. Comparison of TNL with other numerical libraries.

gardless of the great advance in the design of sparse matrix formats for GPUs, it seems that there is no such format which would dominate the others in a large class of sparse matrices. It is profitable for a numerical library to offer several different sparse matrix formats. Having implemented sparse-matrix-vector multiplication, it is relatively simple to incorporate the sparse matrices into iterative *linear systems solvers*. Efficient *preconditioning* for GPUs is still quite rare though very important. Good preconditioning on the CPU can easily outperform high memory bandwidth and computational performance of GPUs. *Nonlinear system solvers*, *eigenvalues computations* or *ODE system solvers* are slightly less common compared to linear systems solvers, nevertheless these algorithms belong to the core of HPC as well. *Stencil computations* cover a large class of numerical algorithms performed on structured rectangular grids. *Unstructured* meshes are necessary for finite volume or for finite element methods. Some libraries also cooperate with Python to offer a more efficient user interface.

Similar to PETSc or ViennaCL, we believe that the main advantage of TNL is that it offers a wider class of data structures and algorithms with a unified interface. For decades, numerical libraries were developed in a very modular way. The users at the end must combine different libraries to solve the problem at hand. This is becoming more difficult on modern architectures like GPUs. Modern features of C++ can make it easier. Our aim is to develop a library with a STL-like consistent templated user interface and native support of GPUs which would make development of HPC algorithms more efficient.

The rest of the paper is organized as follows. First, we briefly explain the basics of GPU programming (Section 2). Then, we describe some basic data structures and solvers already implemented in TNL (Section 3). Finally, we demonstrate the performance of TNL by showing the scalability on multi-core CPUs and speed–up of GPUs (Section 4).

2. PROGRAMMING GPUS

The GPU is an accelerator connected to the CPU via the PCI Express interface. It is equipped with its own memory referred to as *the global memory*. Though its memory bandwidth is several times faster compared to common DDR4 connected to the CPU, communication between the GPU and the CPU is substantially slow. This limits the design of algorithms for GPUs because frequent communication between the GPU and the CPU may negatively affect the overall performance. In case of iterative numerical solvers for PDEs, it is often necessary to copy all necessary data to the GPU before the start of iterations. The result is then copied back to the CPU for post–processing.

Data in the global memory of the GPU need to be accessed in large continuous blocks (this is referred to as *coalesced memory access*), random access is by an order of magnitude slower. Therefore, the data must be often organized in a completely different way than on the CPU. This is also the reason why porting an older code to the GPU is so difficult.

The GPU consists of several independent *multiprocessors* which cannot communicate with each other. They can access the global memory and each multiprocessor has its own *shared memory* which is remarkably smaller (tens of kilobytes only) than the global memory, but much faster in comparison. The shared memory can work as a cache or it can be managed by the programmer. Each multiprocessor may process

32 CUDA threads simultaneously which are referred to as a *warp*. Threads in the same warp behave like the SIMD architecture, i.e., they should follow the same instruction at the same time. If not, we call it *divergent threads*. In this case, efficiency is diminished due to serialization. For more details about CUDA we refer to [1].

3. Data structures and algorithms in TNL

In this section, we describe basic data structures and algorithms implemented in TNL. In the following text, we refer to the GPU as *device* and to the CPU as *host*. Methods, declared as `__cuda_callable__`, can be executed on both, the device and host. If the CUDA framework is not installed, this attribute has no effect.

TNL uses template parameters. Few of them are present in the majority of the code:

- `Device` - can be either `TNL::Devices::Host` or `TNL::Device::Cuda`. It defines where the data are going to be stored and where the related algorithms will be executed.
- `Real` - defines the precision of the floating point arithmetic (`float` or `double`).
- `Index` - defines integer type used for indexing within data structure/algorithm (`int` or `long int`).
- `Allocator` - controls allocation of memory. It can be `TNL::Alocators::Host` for allocation of memory on the host system, `TNL::Allocators::Cuda` for allocation of the memory on the CUDA device `TNL::Allocators::CudaHost` for allocation of a page-locked memory (part of the memory that cannot be swapped off) using CUDA and `TNL::Allocators::CudaManaged` for allocation of CUDA Unified memory (shared memory space between CPU and GPU).

3.1. Arrays and vectors

Arrays are basic structures for the memory management in TNL. An array is a template `TNL::Containers::Array< Value, Device, Index, Allocator >` with template parameters `Value` (an array element type), `Device` (a device where the array elements are stored), `Index` (the indexing type) and `Allocator` (controlling memory allocation). `Array` provides common methods such as allocation (`setSize`), comparison and assignment operators or I/O methods (binary `load` and `save`). Array elements may be manipulated by `setElement`, `getElement` and `__cuda_callable__ operator[]`. The first two methods can be called from the host even for arrays allocated on the device but they cannot be called from the CUDA kernels. The last one, on the other hand, is defined as `__cuda_callable__` and it can be called only from the CUDA kernels, if the array is allocated on the device, and from the host system only if the array is allocated on the host. A method `bind` takes a pointer to an array of given size or an instance of another array. An array adopted in this way is not being deallocated in the `Array` destructor. Such mechanism serves for data sharing between more arrays or for wrapping of data allocated outside of TNL. It also serves for the partitioning of large arrays into a set of smaller ones while keeping them allocated as one continuous block in the memory.

Vectors (`TNL::Containers::Vector< Real, Device, Index, Allocator >`), in TNL, extend arrays with vector operations from the linear algebra like vector addition, scalar products, norms, but also prefix-sum (scan) [9]. The Blas Level 1 operations are handled by expression templates which are more general and user friendly with no loss of performance.

3.2. Matrices

TNL supports dense matrices as well as several formats for the sparse ones (in namespace `TNL::Matrices`). All formats are available for both, the host system and the CUDA device. Optimized sparse matrix formats are crucial especially for good efficiency of the GPU solvers. The user may choose between tridiagonal and multi-diagonal matrices, Ellpack format [10], Sliced Ellpack format [11, 12][1], Chunked Ellpack format [13], BiEll format [14] and CSR format (for the GPU, it is currently in experimental form).

The sparse matrix formats are usually optimized for the matrix-vector multiplication. However, the matrix construction time is also important, especially for non-linear problems where the linear system must be recomputed in each time step. In general, the insertion of a matrix element can be very expansive for the majority of the sparse matrix formats. Global changes of the data structure or even reallocation can be evoked. Fortunately, in many applications the sparse matrix pattern does not change during the computation. In TNL, the matrix assembling process proceeds in two steps:

(1.) *Matrix format meta–data initiation* is based on information about the number of non–zero elements in each row (we refer to it as *compressed row lengths* or *matrix row capacities*). The user calls a method `setCompressedRowLengths` which accepts a vector having the same size as the number of matrix rows. i-th element of the vector says the number of non-zero elements in i-th row. The matrix format is initialized based on this information. Most importantly, it means that the necessary memory for each matrix row is allocated. If the matrix pattern does not change, this method can be called only once.

(2.) *Matrix elements insertion* consists of the non–zero matrix elements insertion. Since the necessary

[1] In [11], we referred the Sliced Ellpack format as Row-grouped CSR. It is, however, almost identical as Sliced Ellpack from [12] and since it uses padding zeros, the name Ellpack seems to be more convenient.

memory for each row is already allocated, this can be done in parallel. Each row can only have as many non-zero elements as its capacity allows.

3.3. Numerical meshes

Currently, TNL supports regular orthogonal structured grids and unstructured conforming homogeneous meshes. In this paper we deal only with the regular grids. Computational results obtained with multidimensional mixed-hybrid finite element method on both structured and unstructured meshes were presented in [15].

The regular orthogonal grids are one, two and three dimensional – `TNL::Meshes::Grid< Dimension, Real, Device, Index >`. The template parameter `Dimension` corresponds to the grid dimension. The others are described at the beginning of Section 3. The grid consists of *mesh entities*. Mesh entity with topological dimension 0 is a *vertex*, mesh entity with the same topological dimension as the mesh itself is a *cell*. Each cell has a boundary made of *faces*, i.e. mesh entities having topological dimension `Dimension-1` . In 3D, each face has a boundary consisting of 1-dimensional *edges*.

Within all mesh entities having the same dimension, each mesh entity has a unique index and coordinates as depicted on the Figure 1.

In 2D, we have two kinds of faces – those parallel to the x-axis (marked with light gray) and those parallel to the y-axis (marked with dark gray). They can be distinguished by its index – the faces parallel to the x-axis are indexed first (indexes $0 - 11$), followed by faces parallel to the y-axis (indexes $12 - 23$) – and their orientation defined by the unit normal. This similarly holds for edges in 3D, whose orientation is given by the axis they are parallel to.

A mesh entity may be obtained using the method `getEntity` defined in `TNL::Meshes::Grid`: All information necessary for a numerical scheme implementation is accessible via the `MeshEntity`. It is especially the mesh entity center, proportions, measure and its neighbor entities indexes. There are several template specializations for each type of the mesh entity depending on what information is precomputed and stored for repetitive use in the numerical scheme. The following code snippet shows how to get their indexes in case of cells:

3.4. Solvers

TNL provides solvers for ODEs and systems of ODEs arising from the method of lines. Currently, the user may choose between the first order accurate Euler solver and the fourth order Runge-Kutta-Merson solver [16] with an adaptive time stepping. Both may run on the GPU. Systems of linear equations can be solved by several Krylov subspace methods like CG, BiCGStab , GMRES and TFQMR (may be executed on the GPU as well [17]) or the stationary SOR method (does not run on the GPU yet).

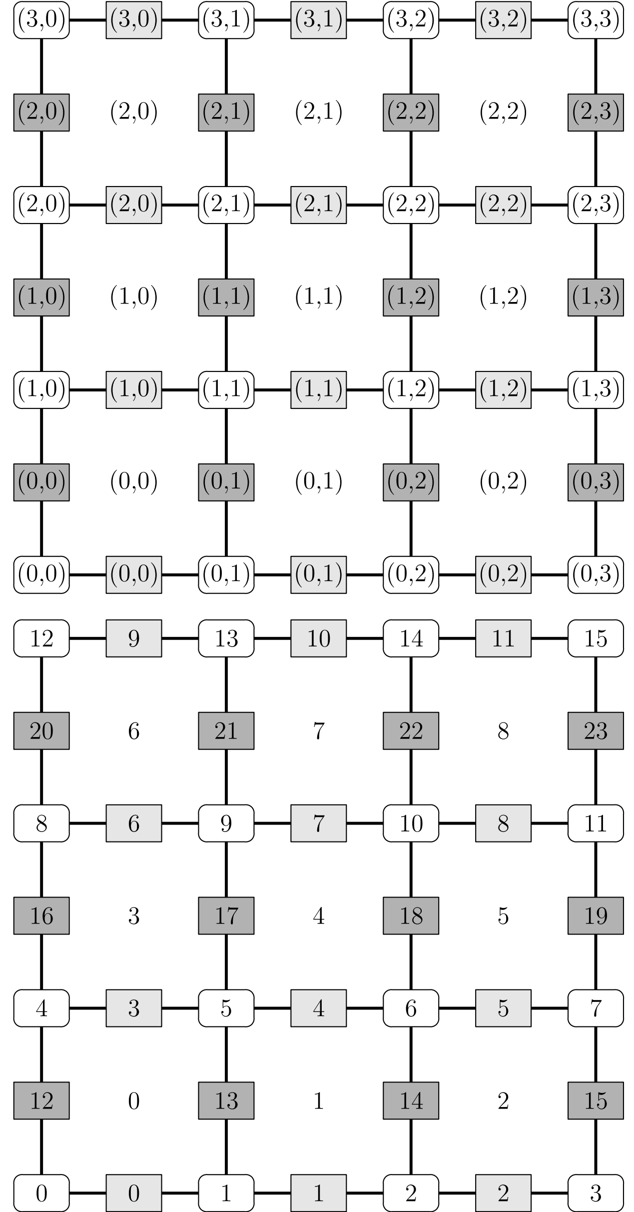

Figure 1. Figure shows 2D grid consisting of 3×3 cells, faces with gray box labels (vertical with dark gray, horizontal with light gray) and vertexes with white rounded box labels. Coordinates are depicted on the top, indexing on the bottom.

4. Solution of parabolic PDEs

Performance of the presented data structures and algorithms is demonstrated on a solution of the heat-equation. The reason for this simple problem is that, especially in explicit time discretization and with zero right-hand side $f(\mathbf{x}, t) = 0$, we get low computational intensity which is the worst case for the GPU and so the lower bound of the speed-up.

We consider a domain $\Omega \equiv [0, 1]^2$, the initial condition $u_{ini} \in C^2(\Omega)$ and the Dirichlet boundary conditions $g \in C^2(\Omega \times (0, T])$ defined as:

$$\frac{\partial u(\mathbf{x}, t)}{\partial t} - \Delta u(\mathbf{x}, t) = f(\mathbf{x}, t) \text{ in } \Omega \times (0, T], \quad (1)$$

$$u(\mathbf{x},0) = u_{ini}(\mathbf{x}) \text{ in } \Omega, \quad (2)$$
$$u(\mathbf{x},t) = g(\mathbf{x},t) \text{ on } \partial\Omega \times (0,T]. \quad (3)$$

The solution u is approximated on vertexes of the 2D grid. Let h be the space step and N the number of cells along x and y axis such that $h = 1/N$. Let

$$\omega_h = \{(ih, jh) \mid 1 \leq i,j \leq N-1\}$$

denote the set of interior grid vertexes and let

$$\overline{\omega}_h = \{(ih, jh) \mid 0 \leq i,j \leq N\} \quad (4)$$

stand for the set of all grid vertexes. Then, by $\partial\omega_h = \overline{\omega_h} \setminus \omega_h$, we denote the boundary vertexes. For some function $u : \Omega \to \mathbb{R}^2$, we define the projection on $\overline{\omega}_h$ as $u_{ij} = u(ih, jh)$. For interior vertexes from ω_h, we define the approximation of the Laplace operator Δ_h as follows:

$$\Delta_h u\left((ih, jh), \cdot\right) \approx \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}}{h^2} = \Delta_h u_{ij}.$$

The explicit time discretization is done using the method of lines which leads to the following system of ODEs

$$\frac{\mathrm{d}}{\mathrm{d}t} u_{ij}(t) = \Delta_h u_{ij}(t) + f_{ij}(t), \quad (5)$$

at the interior vertexes and $u_{ij}(t) = g_{ij}(t)$ at the boundary vertexes. This system can be solved by the Runge-Kutta method. For the semi-implicit time discretization[2], we introduce a time step τ and we denote $u^k_{ij} := u(ih, jh, \tau k)$. With this notation, the semi-implicit scheme is given by a system of linear equations

$$\frac{u^{k+1}_{ij} - u^k_{ij}}{\tau} - \frac{u^{k+1}_{i+1,j} + u^{k+1}_{i-1,j} + u^{k+1}_{i,j+1} + u^{k+1}_{i,j-1} - 4u^{k+1}_{ij}}{h^2} = f^{k+1}_{ij}, \quad (6)$$

at the interior vertexes and

$$u^{k+1}_{ij} = g^{k+1}_{ij} \quad (7)$$

at the boundary vertexes. Both (6) and (7) can be written in the form of a linear system

$$A\mathbf{x}^{k+1} = \mathbf{b}^k, \quad (8)$$

where $x^{k+1}_{iN+j} \equiv u^{k+1}_{ij}$ and for $(i,j) \in \omega_h$

$$\begin{aligned} A_{iN+j,i(N-1)+j} &= -\lambda \\ A_{iN+j,iN+j-1} &= -\lambda \\ A_{iN+j,iN+j+1} &= -\lambda \\ A_{iN+j,i(N+1)+j} &= -\lambda, \\ A_{iN+j,iN+j} &= 1 - 4\lambda, \\ b^k_{iN+j} &= \tau f^{k+1}_{ij} + u^k_{ij} \end{aligned}$$

where we set $\lambda = \tau/h^2$. The Dirichlet boundary conditions (3) are approximated on $(i,j) \in \partial\omega_h$ as $A_{iN+j} = 1$ and $b_{iN+j} = g^{k+1}_{ij}$. Discretization of 1D and 3D problem is done in the same way.

Algorithm 1 Algorithm for the time dependent PDE solver

1: Initialize numerical mesh (4)
2: Allocate degrees of freedom for u^0_{ij}
3: Setup the initial condition (u_{ini})
4: Setup the numerical solver (Runge-Kutta or linear system solver)
5: $t := 0, k := 0, \tau :=$ initial time step
6: **while** $t < T$ **do**
7: $\quad \tau := \min\{\tau, T - t\}$
8: $\quad$ **if** have explicit time discretization **then**
9: $\quad\quad$ Evaluate the right hand side of (5)
10: $\quad\quad$ Update u^k_{ij} by the Runge-Kutta solver to get u^{k+1}_{ij}
11: $\quad$ **else** /* we have semi-implicit time discretization */
12: $\quad\quad$ Assembly the linear system (8)
13: $\quad\quad$ Resolve (8) by the linear system solver to get u^{k+1}_{ij}
14: $\quad$ **end if**
15: $\quad$ t := t + τ, k := k + 1
16: $\quad$ **if** t is snapshot time **then**
17: $\quad\quad$ Make snapshot of the solution u^k_{ij}
18: $\quad$ **end if**
19: **end while**

A numerical solver for such parabolic problem may be written as an Algorithm 1. Each of the steps in this algorithm may be non-trivial for more complex problems or parallel computations. TNL comes with a framework based on what we call a *problem-solver model*. On one hand, there is a *problem* defined by the *user* in a form of a (templated) class. On the other hand, there is a *solver* provided by TNL. The solver interacts with the problem via predefined methods and C++ types definitions.

4.1. PDE solver design

The design of the solver is depicted in Figure 2. The problem to be solved is defined and managed by three template parameters of the solver `TNL::Solvers::Solver` i.e. `ProblemConfig`, `ProblemSetter` and `ProblemBuild`. They serve for definition of the problem configuration parameters, resolution of the problem template parameters and control of the problem build process (complete build may take a lot of time) respectively.

The static method `TNL::Solvers::Solver::run` takes the command line arguments `argc` and `argv`. The TNL solver firstly calls a static method `ProblemConfig::configSetup` to get a definition of the configuration keywords. In the next step, the command line arguments are parsed and based on

[2] The scheme given by (6)–(7) is implicit. In general, implicit schemes for non-linear parabolic PDEs involves the Newton method, which is not implemented in TNL yet. Such problems must be discretized by semi-implicit schemes.

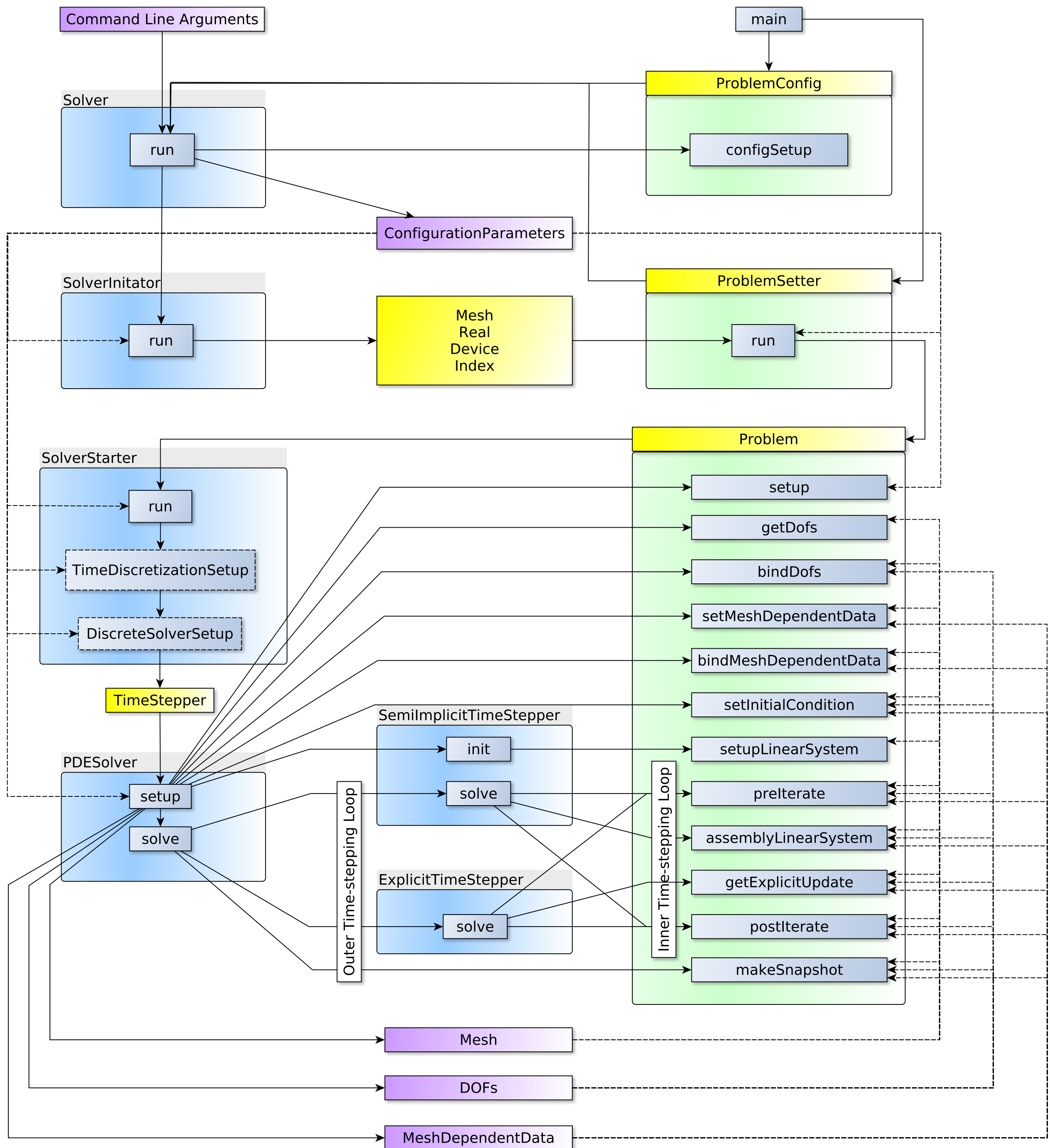

Figure 2. Structure of the PDE framework.The green boxes on the right are part of a *problem* written by the user. The blue ones on the left are the *solver* part implemented in TNL. The yellow boxes represent the template parameters and the violet ones stand for data.

the definition of the configuration keywords, the configuration parameters are resolved and stored in a container `TNL::Config::ParameterContainer`. The control is then passed to the `SolverInitiator` that reads the file with the numerical mesh given by the value of the configuration keyword `-mesh`. It is a binary file created by the TNL tool `tnl-grid-setup` and it contains a binary image of templated class `Grid`. In TNL, most objects may be saved in the binary format using the method `save`. A header of such file contains the object type written in the C++ style, e.g., `TNL::Meshes::Grid< 2, double, TNL::Devices::Host, int >`. The solver initiator parses template arguments of this object type and so it can resolve the mesh type completely. The values of its template parameters `Real` and `Index` are used as default values for floating point arithmetics and indexing type in the *problem*. This can be changed by the configuration keywords `-real-type` and `-index-type`. Together with the argument `-device-type`, the solver initiator can resolve the *primary template arguments* (`RealType`, `DeviceType`, and `IndexType`) and `MeshType` for the *problem*. The solver may, however, depend on some other template types like numerical scheme or boundary conditions. We refer to them as *secondary template arguments*. To resolve them, the control is passed to the `ProblemSetter`.

The solver starter (based on the configuration parameters) sets up the templated types for the time discretization (`TimeStepper`) and related solvers – the Runge-Kutta solver for the explicit time discretization or linear systems solver for semi-implicit scheme. At the end, the solver starter creates an instance of the class `Problem`, passes it to the `PDESolver` and starts the solver. `PDESolver` loads the numerical mesh from the file and, subsequently, it calls methods of the class `Problem` which we describe in the following section.

4.2. PDE problem structure

The class `Problem` representing the heat-equation or similar parabolic problem could be parametrized by four template parameters – `Mesh`, `BoundaryConditions`, `RightHandSide` and `DifferentialOperator` defining the mesh type, the boundary conditions (3), the right-hand side of equation (1) and the differential operator in the same equation (1) respectively. It is inherited from a templated class `TNL::Problems::PDEProblem` which defines the following methods (we list only the most important ones):

- `setup` - this method serves for set-up of the problem configuration parameters which were parsed from the command-line arguments.
- `getDofs` - based on the numerical mesh, problem type (single PDE or system of PDEs) and type of the mesh entities linked with DOFs (cells, faces or vertexes), this method returns a number of DOFs for the unknown mesh function (it is u_{ij}^k (5) or $u_{ij}(t)$ (6) in our example). DOFs are then allocated by the *solver*.
- `bindDofs` - this method serves for binding of DOFs into mesh functions.
- `setInitialCondition` - the initial condition u_{ini} (2) may be set here.
- `getExplicitUpdate` - this method evaluates the right-hand side (5) of the explicit numerical scheme. It is called for the explicit time discretization only.
- `setupLinearSystem` - in the case of semi-implicit time discretization, this method serves for setup of the (sparse) matrix format storing the linear system (8) as described in the Section 3.2. It is called only for the semi-implicit time discretization.
- `assemblyLinearSystem` - this method is responsible for the construction of the linear system (8). It is called for the semi-implicit time discretization only.
- `makeSnapshot` - this method stores the state of the time dependent problem into a file.

5. TNL tools

TNL offers several helper tools (see Figure 3). `tnl-grid-setup` is a simple utility for creating a file with the numerical grid definition. `tnl-init` serves for easier set-up of initial conditions. It produces file with binary image of a mesh function. After the problem solver finishes the computation and saves the solution in a form of a sequence of binary files, they may be post-processed by `tnl-view` to convert the binary data to VTK (or Gnuplot) format or by `tnl-diff` to evaluate differences between different mesh functions or experimental order of convergence (EOC). `tnl-image-converter` can convert images to binary TNL files and vice versa. Currently, TNL supports PGM (natively), PNG (via libpng), JPEG (via libjpeg) and Dicom (via dcmtk).

6. Performance tests

Computational benchmarks were performed on Intel Xeon E5-2640 running at 2.4GHz. This CPU is equipped with 8 computational cores and 25 MB L3 cache and it was connected to DDR4 memory modules. The Turbo-boost technology was turned off only for measuring the weak scalability on CPU. For all other computations it was turned on. The GPU solvers were tested on Nvidia Tesla V100 with 5120 CUDA cores running at 1455 MHz and equipped with 16 GB of global memory. The heat equation (1–3) was solved in a domain $(-1, 1)^n$ where $n = 1, 2, 3$ denotes the dimensions of the domain. The initial condition was $u_{ini}(\mathbf{x}) = \exp\left(-4\left\|\mathbf{x}\right\|^2\right)$. The final time T was set to 0.1.

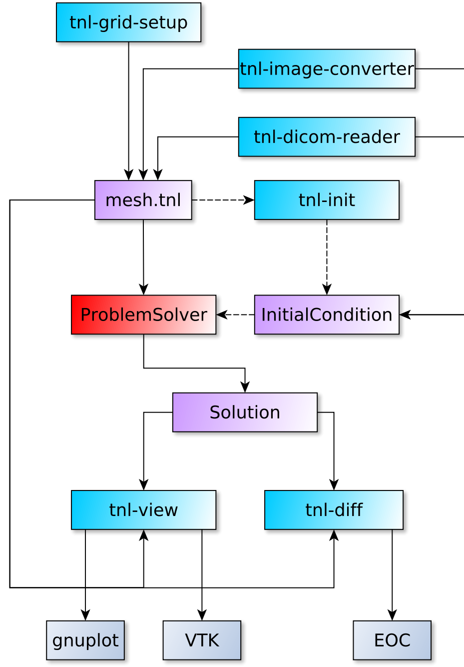

FIGURE 3. TNL tools for a computation data pre-processing and post-processing.

6.1. EXPLICIT NUMERICAL SCHEMES

Firstly, we test the explicit numerical scheme. The integration in time is done by the Runge-Kutta-Merson solver with the adaptive choice of the time step. Results are presented in Tables 2–4. The number of DOFs is shown in the first column. The following columns show times and efficiency of the CPU solver on one, two, four and eight cores, together with the parallel efficiency. The time values in bold face show the best CPU computation time in each row. The last two columns belong to the GPU solver. They show the time and speed-up compared to the best time obtained on CPU. The simulations in 1D (Table 2) have too small number of DOFs and thus we do not obtain good parallel scalability and speed-up on the GPU. In 2D (Table 3), the CPU solver scales relatively well up to four cores on larger numerical meshes. The best time is obtained on eight cores but the parallel efficiency is lower. The GPU speed-up is more than nine on large problems. In 3D, similar results can be seen in Table 4 reporting the GPU speed-up more than eight. Note that Turbo-boost was active on the CPU which can affect the parallel efficiency.

The heat equation (1) with $f(\mathbf{x},t)=0$ exhibits low computational intensity. Therefore the Tables 2–4 demonstrates the lower bound of the GPU speed-up that can be obtained for explicit solvers in TNL. Results with arithmetically more intensive computations are presented in Tables 5–7 where we set

$$f(\mathbf{x},t)=\cos(t)\left(\frac{-2a}{\sigma^2}e^{\frac{-x^2}{\sigma^2}}+\frac{4ax^2}{\sigma^4}e^{\frac{-x^2}{\sigma^2}}\right), \quad (9)$$

$$f(\mathbf{x},t)=\cos(t)\left(\frac{-2a}{\sigma^2}e^{\frac{-x^2-y^2}{\sigma^2}}+\frac{4ax^2}{\sigma^4}e^{\frac{-x^2-y^2}{\sigma^2}}+\right.$$
$$\left.\frac{-2a}{\sigma^2}e^{\frac{-x^2-y^2}{\sigma^2}}+\frac{4ay^2}{\sigma^4}e^{\frac{-x^2-y^2}{\sigma^2}}\right), \quad (10)$$

and

$$f(\mathbf{x},t)=\cos(t)\cdot$$
$$\left(\frac{-2a}{\sigma^2}e^{\frac{-x^2-y^2-z^2}{\sigma^2}}+\frac{4ax^2}{\sigma^4}e^{\frac{-x^2-y^2-z^2}{\sigma^2}}+\right.$$
$$\frac{-2a}{\sigma^2}e^{\frac{-x^2-y^2-z^2}{\sigma^2}}+\frac{4ay^2}{\sigma^4}e^{\frac{-x^2-y^2-z^2}{\sigma^2}}+$$
$$\left.\frac{-2a}{\sigma^2}e^{\frac{-x^2-y^2-z^2}{\sigma^2}}+\frac{4az^2}{\sigma^4}e^{\frac{-x^2-y^2-z^2}{\sigma^2}}\right), \quad (11)$$

for 1D, 2D and 3D respectively. The setup is the same for Tables 2–4, just for 2D and 3D simulations (Tables 6 and 7) we set the final time T was set to 0.01. In this situation we get certain speed-up even for 1D problem. In 2D and 3D the speed-up reaches almost one hundred compared to eight cores CPU time. This is the estimate of the upper bound of speed-up one can obtain with the explicit solver.

6.2. SEMI-IMPLICIT NUMERICAL SCHEMES

Tables 8–10 show results obtained by the semi-implicit numerical scheme (6)–(7). In this case, majority of the time is spend by solving the linear system (8). The sparse matrix A in (8) is stored in CSR format on CPU and SlicedEllpack format on GPU. The linear system was solved by the GMRES method on CPU and CWYGMRES [18] on GPU. The use of GPU makes no sense in 1D where the speed-up is smaller than one as well as in the case of the explicit solver. In 2D and 3D, the speed-up is more than 12 and 10 respectively.

The Figure 4 shows graphs of efficiency of the CPU solvers and so it demonstrates the strong parallel scalability of different solvers on different problems sizes. The left column shows scalability in 1D where the problem size is usually too small for efficient parallelization. In 2D and 3D (the second and the third column), especially for larger grid size, the efficiency grows. The middle row shows the result of arithmetically more intensive explicit solver with f given by (9)–(11). The efficiency is significantly higher. It indicates that the CPU solvers are probably limited by the memory bandwidth.

The weak scalability is reported in Table 11 and Figure 5. To study the weak scalability we increase the problem size linearly with the number of threads by prolongating the domain Ω along the x axis. The initial condition is scaled in the same way. We set the domain Ω as $\Omega\equiv(0,p)$, $\Omega\equiv(0,p)\times(0,1)$ and $\Omega\equiv(0,p)\times(0,1)\times(0,1)$ in 1D, 2D and 3D respectively, where p denotes the number of threads. The discrete numerical mesh ω_h resolution is set as $10000p$, $200p\times 100$ and $100p\times 50\times 50$ in 1D, 2D and 3D respectively.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16	0.003	**0.002**	0.75	0.003	0.25	0.003	0.12	0.04	**0.05**
32	0.003	**0.003**	0.5	0.003	0.25	0.003	0.12	0.04	**0.07**
64	0.005	**0.005**	0.5	0.005	0.25	0.005	0.12	0.05	**0.1**
128	0.015	0.015	0.5	0.015	0.25	**0.01**	0.18	0.12	**0.08**
256	0.06	**0.06**	0.5	0.06	0.25	0.06	0.12	0.43	**0.13**
512	0.28	**0.29**	0.48	0.33	0.21	0.34	0.10	1.36	**0.20**
1024	1.45	**1.8**	0.40	1.85	0.19	2.2	0.08	5.32	**0.27**
2048	8.57	9.111	0.47	**8.5**	0.25	9.38	0.11	20.95	**0.40**
4096	56.37	50.94	0.55	**40.9**	0.34	41.9	0.16	84.66	**0.48**

Table 2. Performance of the **explicit** numerical solver of (5) for the heat equation in **1D**. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16^2	**0.003**	0.003	0.5	0.003	0.25	0.008	0.04	0.003	**0.03**
32^2	**0.005**	0.006	0.41	0.006	0.20	0.006	0.10	0.07	**0.07**
64^2	0.03	0.027	0.55	**0.022**	0.34	0.028	0.13	0.09	**0.33**
128^2	0.41	0.27	0.75	0.17	0.60	**0.16**	0.32	0.22	**0.72**
256^2	6.17	3.67	0.84	2.12	0.72	**1.46**	0.52	0.79	**1.84**
512^2	96	55.47	0.86	30.84	0.77	**18.7**	0.64	5.73	**3.26**
1024^2	1743	990.7	0.87	556.9	0.78	**381.6**	0.57	64.82	**5.88**
2048^2	31226	17627.7	0.88	10403.4	0.75	**8297.8**	0.47	911.11	**9.10**

Table 3. Performance of the **explicit** numerical solver of (5) for the heat equation in **2D**. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16^3	0.005	0.005	0.5	0.004	0.31	**0.004**	0.15	0.08	**0.05**
32^3	0.07	0.049	0.71	0.031	0.56	**0.02**	0.43	0.08	**0.25**
64^3	2.46	1.47	0.83	0.8	0.76	**0.49**	0.62	0.23	**2.13**
128^3	94.32	53.08	0.88	30.66	0.76	**23.75**	0.49	3.48	**6.82**
256^3	3050.47	1720.04	0.88	1005.89	0.75	**827.16**	0.46	103.46	**7.99**

Table 4. Performance of the **explicit** numerical solver of (5) for the heat equation in **3D**. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16	0.003	0.003	0.5	0.003	0.25	**0.003**	0.12	0.02	**0.15**
32	0.003	0.003	0.5	0.003	0.25	**0.003**	0.12	0.05	**0.05**
64	0.007	0.007	0.5	0.007	0.25	**0.007**	0.12	0.07	**0.09**
128	0.032	0.032	0.5	0.03	0.26	**0.03**	0.13	0.11	**0.27**
256	0.19	0.2	0.47	0.2	0.23	**0.2**	0.11	0.36	**0.52**
512	1.35	1.37	0.49	1.5	0.22	**1.5**	0.11	1.37	**0.98**
1024	10.58	6.08	0.87	4.21	0.62	**3.64**	0.36	5.25	**0.69**
2048	78.26	43.08	0.90	26.86	0.72	**20.13**	0.48	20.74	**0.97**
4096	609.17	321.73	0.94	190.37	0.79	**128.69**	0.59	83.55	**1.54**

Table 5. Performance of the **explicit** numerical solver of (5) for the heat equation in **1D** with f given by (9) and $T = 0.1$. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16^2	0.004	0.004	0.5	**0.004**	0.25	0.006	0.08	0.04	**0.1**
32^2	0.01	0.007	0.71	**0.006**	0.41	0.007	0.17	0.05	**0.12**
64^2	0.03	0.02	0.75	0.01	0.75	**0.01**	0.37	0.08	**0.12**
128^2	0.69	0.34	1.01	0.19	0.90	**0.14**	0.61	0.06	**2.33**
256^2	11.03	5.65	0.97	3.13	0.88	**1.83**	0.75	0.12	**15.2**
512^2	181.6	93.57	0.97	53.65	0.84	**29.29**	0.77	0.63	**46.4**
1024^2	2996.2	1557.51	0.96	865.7	0.86	**481.47**	0.77	6.41	**75.1**
2048^2	48965	25065.9	0.97	13716.7	0.89	**8002.77**	0.76	87.6	**91.3**

Table 6. Performance of the **explicit** numerical solver of (5) for the heat equation in **2D** with f given by (10) and $T = 0.01$. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16^3	0.03	0.02	0.75	0.014	0.53	**0.011**	0.34	0.01	**1.1**
32^3	0.33	0.16	1.03	0.1	0.82	**0.09**	0.45	0.01	**9**
64^3	4.5	2.33	0.96	1.31	0.85	**0.81**	0.69	0.03	**27**
128^3	174.2	88.33	0.98	50.11	0.86	**28.65**	0.76	0.37	**77**
256^3	6047.66	3071.05	0.98	1696	0.89	**982.2**	0.76	9.87	**99**

Table 7. Performance of the **explicit** numerical solver of (5) for the heat equation in **3D** with f given by (11) and $T = 0.01$. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16	**0.003**	0.003	0.5	0.003	0.25	0.003	0.12	0.03	**0.1**
32	**0.003**	0.003	0.5	0.004	0.18	0.004	0.09	0.13	**0.02**
64	**0.006**	0.006	0.5	0.007	0.21	0.011	0.06	0.26	**0.02**
128	**0.016**	0.018	0.44	0.019	0.21	0.025	0.08	0.9	**0.02**
256	**0.081**	0.1	0.40	0.09	0.22	0.11	0.09	1.8	**0.04**
512	**0.48**	0.51	0.47	0.54	0.22	0.55	0.1	6.05	**0.08**
1024	3.42	2.98	0.57	**2.83**	0.30	3.57	0.11	21.6	**0.17**
2048	24.88	17.29	0.71	**14.05**	0.44	15.5	0.20	85.8	**0.16**
4096	189.7	114.12	0.83	79.53	0.59	**76.11**	0.31	342.8	**0.22**

Table 8. Performance of the **implicit** numerical solver (6)–(7) for the heat equation in **1D**. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16^2	**0.004**	0.004	0.5	0.01	0.1	0.03	0.02	0.04	**0.1**
32^2	0.015	0.012	0.62	0.011	0.34	**0.01**	0.18	0.12	**0.08**
64^2	0.12	0.07	0.85	**0.047**	0.63	0.05	0.3	0.3	**0.15**
128^2	1.2	0.65	0.92	0.37	0.81	**0.26**	0.57	0.83	**0.31**
256^2	17.51	9.17	0.95	4.79	0.91	**3.03**	0.72	2.86	**1.05**
512^2	230.9	120.83	0.95	66.49	0.86	**39.62**	0.72	12.07	**3.28**
1024^2	3634.24	1928.66	0.94	1081.04	0.84	**752.73**	0.60	86.9	**8.66**
2048^2	59185	32048	0.92	18080.1	0.81	**13167.5**	0.56	1057	**12.4**

Table 9. Performance of the **implicit** numerical solver (6)–(7) for the heat equation in **2D**. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

DOFs	CPU							GPU	
	1 core	2 cores		4 cores		8 cores			
	Time (s)	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Eff.	Time (s)	Speed-up
16^3	0.021	0.013	0.80	0.01	0.52	**0.011**	0.23	0.07	**0.14**
32^3	0.46	0.25	0.92	0.14	0.82	**0.11**	0.52	0.18	**0.61**
64^3	10.04	5.21	0.96	3.02	0.83	**1.83**	0.68	0.6	**3.05**
128^3	240.9	125.11	0.96	81.24	0.74	**51.46**	0.58	6.54	**7.86**
256^3	6429.6	3815.9	0.84	2313.79	0.69	**1536.4**	0.52	140.1	**10.96**

Table 10. Performance of the **implicit** numerical solver (6)–(7) for the heat equation in **3D**. Bold face stresses the best time on CPU (with Turbo-boost turned on) based on which we compute the GPU speed-up written in bold face as well.

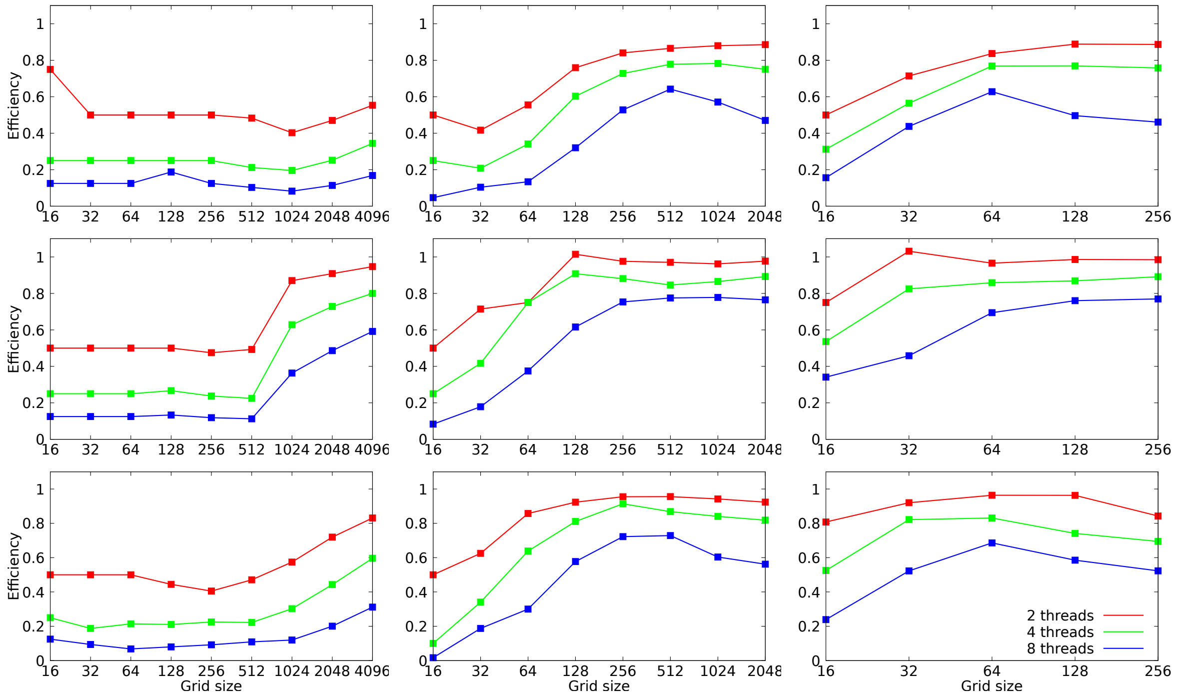

FIGURE 4. Graphs of the CPU efficiency – the first row represents the explicit solver from Tables 2–4, the second row is the explicit solver with f given by (9)–(11) from Tables 5–7 and the last row is the implicit solver from Tables 8–10. The first column contains results of computations in 1D, the second column in 2D and the third one in 3D. The red curve is efficiency of simulation with two threads, the green one with four threads and the blue one with eight threads. The horizontal axes represent grid size of the simulation and the vertical axes show the efficiency.

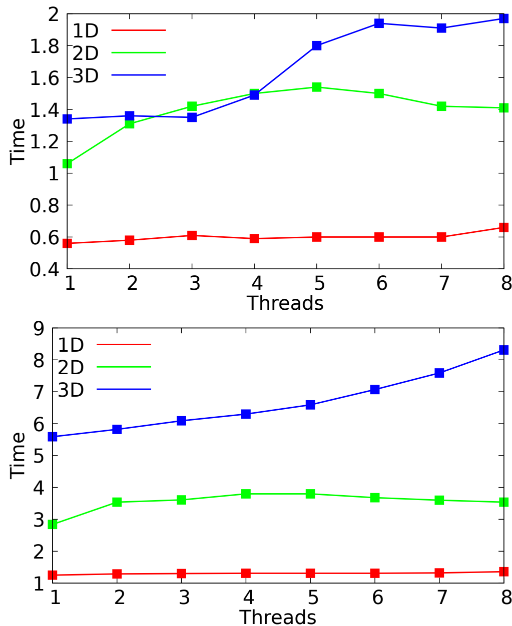

FIGURE 5. Graphs of weak parallel scalability on the CPU – the figure on the top shows the results of the explicit solver, the one on the bottom shows the implicit solver. The red curve represents computations in 1D, the green one in 2D and the blue one in 3D. The problem size grows linearly with the number of threads. Therefore the more straight the curve is the better parallel scalability the solver exhibits.

	Time(s)					
	Explicit			Implicit		
Threads	1D	2D	3D	1D	2D	3D
1	0.56	1.06	1.34	1.25	2.84	5.59
2	0.58	1.31	1.36	1.29	3.54	5.82
3	0.61	1.42	1.35	1.30	3.61	6.09
4	0.59	1.51	1.49	1.31	3.80	6.30
5	0.60	1.54	1.80	1.31	3.80	6.59
6	0.61	1.51	1.94	1.31	3.68	7.07
7	0.60	1.42	1.91	1.32	3.60	7.59
8	0.66	1.41	1.97	1.36	3.54	8.31

TABLE 11. The weak scalability on the CPU – the columns represent the CPU time of simulation on a domain growing linearly with the number of threads (the first column). The less the times grow with the number of threads the better the weak scalabiltiy is.

7. FUTURE WORK

In the future we would like to improve the support of MPI to make computations on clusters easier, implement support of GPUs by AMD company via ROCm toolkit [19] and create interface of TNL into Julia [20]. As we mentioned before, efficient preconditioners for linear systems solvers are extremely important. Currently we are working on a more flexible implementation of sparse matrices, on adaptive numerical grids and distributed unstructured numerical meshes.

8. Conclusion

We have presented Template Numerical Library, TNL, for easy development of numerical solvers on modern parallel architectures. We have described details of the TNL design and we have presented a number of numerical experiments to demonstrate the scalability of TNL on multi-core CPUs together with the speed-up on GPUs. The explicit solver achieves speed–up of 8 to 99 depending on the arithmetic intensity. The semi-implicit solver gives a speed–up of almost 11. Both results were obtained by the comparison of 8 core CPU Intel Xeon with the GPU Nvidia Tesla V100.

Acknowledgements

The work was supported by the Ministry of Education, Youth and Sports OPVVV project no. CZ.02.1.01/0.0/0.0/16_019/0000765: Research Center for Informatics, the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center – LM2015070 and Project No. 18-09539S of the Czech Science Foundation.

References

[1] J. Cheng, M. Grossman, T. McKercher. *Professional CUDA C Programming.* Wrox, 2014.

[2] Cublas. `https://developer.nvidia.com/cublas`.

[3] Cusparse. `https://developer.nvidia.com/cusparse`.

[4] Thrust. `https://developer.nvidia.com/thrust`.

[5] Kokkos. `https://github.com/kokkos/kokkos`.

[6] Viennacl. `http://viennacl.sourceforge.net`.

[7] Petsc. `https://www.mcs.anl.gov/petsc`.

[8] Eigen. `http://eigen.tuxfamily.org/index.php`.

[9] M. Harris, S. Sengupta, J. D. Owens. *GPU gems 3*, chap. Parallel prefix sum (scan) with CUDA, pp. 851–876. 2007.

[10] N. Bell, M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Tech. Rep. Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[11] T. Oberhuber, A. Suzuki, J. Vacata. New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. *Acta Technica* **56**:447–466, 2011.

[12] A. Monakov, A. Lokhmotov, A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *HiPEAC 2010*, pp. 111–125. Springer-Verlag Berlin Heidelberg, 2010.

[13] M. Heller, T. Oberhuber. Improved Row-grouped CSR Format for Storing of Sparse Matrices on GPU. In A. H. nad Z. Minarechová, D. Ševčovič (eds.), *Proceedings of Algoritmy 2012*, pp. 282–290. 2012.

[14] C. Zheng, S. Gu, T.-X. Gu, et al. Biell: A bisection ellpack-based storage format for optimizing spmv on gpus. *Journal of Parallel and Distributed Computing* **74**(7):2639 – 2647, 2014.

[15] R. Fučík, J. Klinkovský, J. Solovský, et al. Multidimensional mixed-hybrid finite element method for compositional two-phase flow in heterogeneous porous media and its parallel implementation on GPU. *Computer Physics Communications* **238**:165–180, 2019.

[16] T. Oberhuber, A. Suzuki, V. Žabka. The CUDA implementation of the method of lines for the curvature dependent flows. *Kybernetika* **47**:251–272, 2011.

[17] T. Oberhuber, A. Suzuki, J. Vacata, V. Žabka. Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods. *Journal of Math-for-Industry* **3**:73–79, 2011.

[18] Y. Yamamoto, Y. Hirota. A parallel algorithm for incremental orthogonalization based on compact WY representation. *JSIAM Letters* **3**(0):89–92, 2011.

[19] Rocm. `https://github.com/RadeonOpenCompute/ROCm`.

[20] Julia. `https://julialang.org`.