ScaFES: An Open-Source Framework for Explicit Solvers Combining High-Scalability with User-Friendliness

Martin Flehmig
Technische Universität Dresden
Center for Information Services and
High Performance Computing (ZIH)
martin.flehmig@tu-dresden.de

Kim Feldhoff
Technische Universität Dresden
Center for Information Services and
High Performance Computing (ZIH)
kim.feldhoff@tu-dresden.de

Ulf Markwardt
Technische Universität Dresden
Center for Information Services and
High Performance Computing (ZIH)
ulf.markwardt@tu-dresden.de

***Abstract*—We present ScaFES, an open-source HPC framework written in C++11 for solving initial boundary value problems using explicit numerical methods in time on structured grids. It is designed to be highly-scalable and very user-friendly, i.e. to exploit all levels of parallelism and provide easy-to-use interfaces. Besides, the numerical nomenclature is reflected in a nearly one-to-one mapping.**

We describe how the framework works internally by presenting the core components of ScaFES, which modern C++ technologies are used, which parallelization methods are employed, and how the communication can be hidden behind during the update phase of a time step.

Finally, we show how a multidimensional heat equation problem discretized via the finite difference method in space and via the explicit Euler scheme in time can be implemented and solved using ScaFES in about 60 lines. In order to demonstrate the excellent performance of ScaFES, we compare ScaFES to PETSc on the basis of the implemented heat equation example in two dimensions and present scalability results w.r.t. MPI and OpenMP achieved on HPC clusters at the ZIH.

I. Yet Another Framework?

A wide variety of phenomena like heat transport, fluid flow, and electrostatics can be described by initial boundary value problems of the following type: Given a time interval $[t_S; t_E]$ with $0 \leq t_S < t_E$, an open, bounded domain $\Omega \subset \mathbb{R}^d$ with dimension $d \in \mathbb{N}$ and boundary $\partial\Omega$, a source $f : \bar{\Omega} \times (t_S; t_E] \to \mathbb{R}^m$, a boundary condition $g : \partial\Omega \times (t_S; t_E] \to \mathbb{R}^m$, an initial condition $\widetilde{u} : \Omega \to \mathbb{R}^m$, and differential operator F, then the task is to find $u : \bar{\Omega} \times [t_S; t_E] \to \mathbb{R}^m$ such that the following system of equations is fulfilled:

$$\begin{aligned} \partial_t u + F(u, \nabla u, \ldots) &= f && \text{in } \Omega \times (t_S; t_E], \\ u &= g && \text{on } \partial\Omega \times (t_S; t_E], \\ u(\cdot, t_S) &= \widetilde{u} && \text{in } \Omega. \end{aligned}$$

Analytical solutions of such problems exist only for rare cases. Nevertheless, engineers and scientists want to have more and more detailed approximations of these problems, resulting in a significantly increase of the memory requirements as well as the computational time. These computations can only be run in parallel. There are many software packages available which can solve these problems numerically using simple methods like finite difference methods (FDM) or more complex methods like finite elements or spectral methods (AMDiS [1], PETSc [2], and DUNE [3]). So, why should it be necessary to implement yet another framework for solving initial boundary value problems?

The answer is that the software packages like the ones mentioned above, can solve these problems by combining several numerical methods, parallelization approaches and implementation languages. But they have all been designed for more general purposes and therefore provide a large and quite complex infrastructure with a lot of objects, methods and modules which lead to long learning curves. Roughly speaking, they are kind of heavyweight. And indeed, for many initial boundary value problems it is sufficient to use simple numerical methods like the explicit finite difference method on a structured grid (e.g. for solving Maxwell's equations ([4]).

Thus, we designed the framework ScaFES ("Scalable Framework for Explicit Solvers") for explicit methods in time and space on structured grids. Instead of expanding existing software but writing ScaFES from scratch we had the opportunity to clearly focus on our design principles. These are:

- High-scalability, i.e. all levels of parallelism on current multi- and many-core architectures should be efficiently used.
- User-friendliness, i.e. ScaFES should have easy-to-use interfaces such that users can implement their numerical methods as usual. In particular, the numerical nomenclature should be reflected in a nearly one-to-one mapping and knowledge about parallelization aspects should not be required. Besides, it should be easy to build and install on a wide variety of platforms.

As a consequence, ScaFES can be used as a rapid prototyping tool to evaluate and compare different numerical approaches as well as to write high quality production code without loosing scalability and efficiency.

The presented work is organized as follows: In section II, we will discuss the design concepts of the framework. How a multidimensional heat equation problem can be solved using ScaFES will be demonstrated in section III. In section IV, we will present scalability results achieved on an HPC cluster at the ZIH for the implemented problem in the three-dimensional

case. The paper concludes in section V with a summary of the results and an outlook on further work related to ScaFES.

II. Design Concepts of ScaFES

In the following, we describe the design concepts of ScaFES, i.e. the implementation of the design principles. In order to fulfill the two principles high-scalability and user-friendliness we have chosen C++11 [5] as programming language. Since C++11 contains modern programming concepts and features like constructor delegations, class and function templates as well as STL containers, the framework allows well structured development without significant performance losses. The readability and usability of the source code was improved by using additional features of the Boost C++ libraries [6]. The framework is based on the GNU autotools [7] which are available on almost all Linux based systems in order to allow the installation of ScaFES on a high-variety of different systems. This means that the build and installation process is a combination of the usual calls to `configure`, `make`, and `make install`.

ScaFES is highly modularized. It consists of the following core components: The class templates `Grid<DIM>` and `GridSub<DIM>` for the representations of structured grids resp. sub-grids, the class template `GridGlobal<DIM>` for the representation of global grids, the class template `DataField<CT,DIM>` for the representation of physical fields, and the class template `Problem<PRBLM,CT,DIM>` for the representation of initial boundary value problems. In the following subsections, we will present these class templates and their design concepts in detail.

A. Representation of Structured Grids

The considered problems should be discretized using numerical methods which are based on structured grids. In the following, we refer to a structured grid as a uniform decomposition of a given domain $D \subset \mathbb{R}^d$ into d-dimensional hypercuboids. More precise, let $D := (s_0, e_0) \times \ldots (s_{d-1}, e_{d-1})$ the given domain with $s_p < e_p \in \mathbb{R}$ for each direction $p \in \{0, 1, \ldots, d-1\}$ and $2 \leq n_p \in \mathbb{N}$ the number of grid nodes in each direction p. As the domain D is uniformly decomposed, the corresponding grid size $h_p := (e_p - s_p)/(n_p - 1)$ is constant in each direction p. The grid nodes are numbered accordingly to their positions $i = (i_0, i_1, \ldots, i_{d-1}) \in \mathbb{N}_0^d$ in the grid. Then, for a given position $(i_0, i_1, \ldots, i_{d-1})$, the corresponding real-world coordinates $x_{(i_0,i_1,\ldots,i_{d-1})} \in \mathbb{R}^d$ are given by

$$x_{(i_0,i_1,\ldots,i_{d-1})} = (s_0 + i_0 \cdot h_0, \ldots, s_{d-1} + i_{d-1} \cdot h_{d-1}).$$

The set of all coordinates $x_{(i_0,i_1,\ldots,i_{d-1})}$ is denoted by D_h and the set of the corresponding integer tuples $(i_0, i_1, \ldots, i_{d-1})$ by $\mathcal{G}(D_h)$. Furthermore, the sets $\mathcal{G}_I(D_h)$ and $\mathcal{G}_B(D_h)$ are defined as index sets of all interior resp. boundary nodes of D_h such that $\mathcal{G}(D_h) = \mathcal{G}_I(D_h) \cup \mathcal{G}_B(D_h)$, and the total number of grid nodes is denoted by $N := \prod_{p=0}^{d-1} n_p$. All quantities are also explained in Fig. 1a. The positions of the direct neighboring nodes in direction p for a given interior grid node number $i = (i_0, i_1, \ldots, i_{d-1}) \in \mathcal{G}_I(\Omega_h)$ can be accessed using the following connectivity mapping c. Fig. 1b illustrates the mapping in two dimensions.

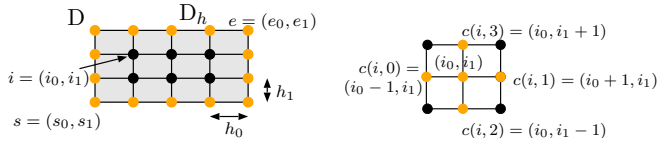

Fig. 1: (a): Two-dimensional grid D_h of a given domain $D = (s_0, e_0) \times (s_1, e_1)$ with 5×4 nodes: All inner grid nodes are colored black, all boundary grid nodes are colored orange. (b): Indices of direct neighboring nodes of a given grid node with index (i_0, i_1), accessed via the connectivity mapping c.

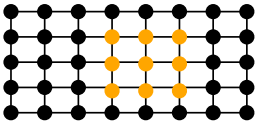

Fig. 2: Two-dimensional (base) grid of 8×5 grid nodes with a sub-grid of 3×3 grid nodes colored orange.

$$c(i; 2 \cdot p) := (i_0, i_1, \ldots, i_{p-1}, i_p - 1, i_{p+1}, \ldots, i_{d-1}),$$
$$c(i; 2 \cdot p + 1) := (i_0, i_1, \ldots, i_{p-1}, i_p + 1, i_{p+1}, \ldots, i_{d-1}).$$

Due to the regular structure of the grids, the set D_h can be completely described by the coordinates $s = (s_0, s_1, \ldots, s_{d-1})$ and $e = (e_0, e_1, \ldots, e_{d-1})$ of the domain D together with the number of grid nodes $n = (n_0, n_1, \ldots, n_{d-1})$ in each direction. In ScaFES, the set D_h is represented by the class template `Grid<DIM>`. The quantities s, e and n are given as member variables (see Listing 1). In particular, there is no need to create huge arrays storing the coordinates and the indices of all grid nodes. The space dimension d is implemented as template parameter `DIM` such that the size of all member variables is known at compile time.

```
// Number of nodes: n=(n_0,...,n_{d-1})
ScaFES::Ntuple<int, DIM> mNnodes;
// Coordinates of first node: s=(s_0,...,s_{d-1})
ScaFES::Ntuple<double, DIM> mCoordNodeFirst;
// Coordinates of last node: e=(e_0,...,e_{d-1})
ScaFES::Ntuple<double, DIM> mCoordNodeLast;
```

Listing 1: Member variables of the class template `ScaFES::Grid<DIM>`.

Subsets of the grid D_h of the following type will be referred to as so-called "sub-grids":

$$\Delta_h := \Big\{ x_{(i_0,i_1,\ldots,i_{d-1})} \in D_h : \; a \leq (i_0, i_1, \ldots, i_{d-1}) \leq b \text{ with } a, b \in \mathbb{N}_0^d \text{ as the indices of the first resp. the last node of the subset.} \Big\}.$$

Fig. 2 shows a grid and a sub-grid in two dimensions. The subset Δ_h can be described by the base grid D_h and the indices a and b of the first and last node of the sub-grid. As Δ_h is related to the (base) grid D_h, the sub-grid is represented by a sub class template named `GridSub<DIM>` of the (base) class template `Grid<DIM>` which itself represents the (base) grid D_h. The indices a and b are given as member variables (see Listing 2), the space dimension d is implemented as template parameter `DIM`.
Currently, all grids and sub-grids will be traversed lexicographically in C style, i.e. node numbers $(i_0, i_1, \ldots, i_{d-1})$ in

```
// Index of first node: a=(a_0,...,a_{d-1})
ScaFES::Ntuple<int, DIM> mIdxNodeFirstSub;
// Index of last node: b=(b_0,...,b_{d-1})
ScaFES::Ntuple<int, DIM> mIdxNodeLastSub;
```

Listing 2: Member variables of the class template `ScaFES::GridSub<DIM>`.

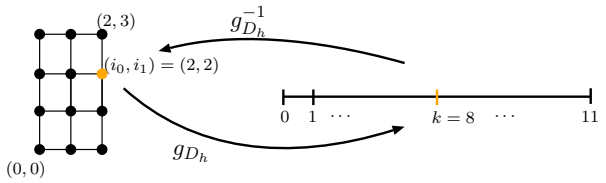

Fig. 3: Correspondents between node numbers in tuple notation (i_0, i_1) and in scalar notation k via the mappings g_{D_h} and $g_{D_h}^{-1}$ of a two-dimensional grid D_h with 3×4 grid nodes. The correspondent is explicitly shown for the node number $(2, 2)$.

tuple notation of sub-grids correspond to scalar node numbers $k \in \mathbb{N}_0$ and vice versa according to the following mappings:

$$g_{D_h}(i_0, i_1, \ldots, i_{d-1}) := \sum_{p=0}^{d-1} (i_p - c_p) \cdot \prod_{q=1}^{p} n_q,$$

$$g_{D_h}^{-1}(k) = \left(c_p + \left\lfloor k / \prod_{p=0}^{p-1} n_q \right\rfloor \bmod n_p \right)_{p=0,\ldots,d-1}$$

with $\lfloor . \rfloor$ as lower Gaussian bracket and $c \in \mathbb{Z}^d$ as index of the first node of the base grid D_h. Fig. 3 shows the correspondents via the mappings g_{D_h} and $g_{D_h}^{-1}$ for a two-dimensional grid.

For traversing through grids and sub-grids, the class templates `Grid<DIM>` and `GridSub<DIM>` each have an internal class named `Iterator`. These internal classes are designed like the iterators of the STL. Thus, users do not have to learn new patterns but can apply the iterators in the same way. Furthermore, the employment of these iterators has the advantage that the numbering of the nodes is hidden from the user and therefore, can be easily changed if necessary (see Listing 3).

```
ScaFES::Grid<DIM> gd;
for (ScaFES::Grid<DIM>::iterator it = gd.begin(),
     et = gd.end(); it < et; ++it) { // [...]
}
```

Listing 3: Application of an iterator of the class template `ScaFES::Grid<DIM>`.

The index $(i_0, i_1, \ldots, i_{d-1})$ of the current grid node can be accessed via `it.idxNode()`, the corresponding scalar $g(i_0, i_1, \ldots, i_{d-1})$ of the current tuple can be accessed via `it.idxScalarNode()`.

B. Representation of the (Discretized) Computational Domain

In order to solve initial boundary value problems using grid-based methods, the computational domain Ω has to be discretized on a given set of grid nodes, first. Usually, the number of grid nodes is very huge ($> 10^7$). This would result in a system of equations which would be too large to be solved in serial. Thus, the discretized computational domain $\Omega_h \subset \mathbb{R}^d$ (called "global grid") has to be decomposed into a given number $q \in \mathbb{N}$ of sub-grids S_k with

$$\bigcup_{k=0}^{q-1} S_k = \Omega_h$$

and the corresponding grid partitions S_k have to be distributed to the appropriate cores of the parallel hardware such that each core will work on an appropriate subset of all grid nodes, only. This so-called "domain decomposition approach" [8] fits to our needs as we have restricted the framework to structured grids. The communication in terms of messages between the cores is enabled by the Message Passing Interface (MPI), Currently, the global grid is partitioned based on the well-known RCB ("Recursive Coordinates Bisection") algorithm [9]. The global grid Ω_h is described by the grid partitions S_k, the number of partitions n_p, and the relations between the grid partitions. Due to the regular structure of the underlying grids, these relations are completely described by the identifiers and the directions of the direct neighboring grid partitions.

The type of all nodes $j \in \mathcal{G}(\Omega_h)$ of the global grid will be stored in a vector $T \in \mathbb{N}^N$ in order to distinguish if a node is a global interior one or a global boundary one:

$$T_j := 1 \quad \text{for all } j \in \mathcal{G}_I(\Omega_h),$$
$$T_j := 2 \quad \text{for all } j \in \mathcal{G}_B(\Omega_h).$$

Additionally, the type of all nodes related to a grid partition S_k will be stored in a vector $R^{(k)} \in \mathbb{N}^{M_k}$ with $M_k \in \mathbb{N}$ as the number of nodes of the grid partition S_k:

$$R_j^{(k)} := 4 \quad \text{for all } j \in \mathcal{G}_I(S_k),$$
$$R_j^{(k)} := 8 \quad \text{for all } j \in \mathcal{G}_B(S_k).$$

The values of T and $R^{(k)}$ are chosen as elements of the dual basis such that the sum can be created and values can be easily extracted via bitwise operators. Fig. 4 illustrates the different grid node types on a two-dimensional computational domain which is discretized and decomposed into four grid partitions.

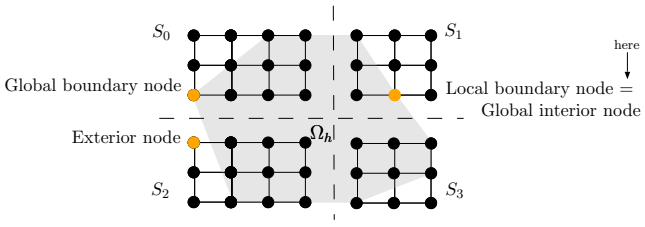

Fig. 4: Decomposition of a two-dimensional global grid Ω_h with 7×6 grid nodes into four grid partitions S_k. Local and global interior resp. boundary nodes are identified.

In ScaFES, global grids are represented by the sub class template `GridGlobal<DIM>` derived from the class template `Grid<DIM>`. The grid partitions S_k, the identifiers of the neighboring partitions and its directions are stored as member variables (see Listing 4). The direction of the

```
// All grid partitions: S_k for k=0,1,...,d-1
std::vector< GridSub<DIM> > mPartition;
// Identifiers of all neighbours of all partitions.
std::vector< std::vector<int> > mvNeighbourId;
// Directions of all neighbours of all partitions.
std::vector< std::vector<int> > mvNeighbourDir;
```

Listing 4: Member variables of the class template `ScaFES::GridGlobal<DIM>`.

neighbors of a given grid partition S_k is described for all $p \in \{0, 1, 2, \ldots, d-1\}$ by the following variable:

$$n(k, 2 \cdot p) := \text{left neighbor of } k \text{ in direction } p,$$
$$n(k, 2 \cdot p + 1) := \text{right neighbor of } k \text{ in direction } p.$$

C. Representation of Physical Fields

Let $v : \Omega \to \mathbb{R}^m$ a given vector-valued physical field and $v_h : \Omega \to \mathbb{R}^m$ the corresponding approximation of v. The approximation v_h to v on the domain Ω can be alternatively described by the matrix $V \in \mathbb{R}^{N,m}$ which contains the function values of the discrete function v_h at all grid nodes $x_j \in \Omega_h$:

$$V_{j,q} := \left[v_h(x_j)\right]_q \qquad \forall j \in \{0, 1, 2, \ldots, N-1\}, \quad \forall q \in \{0, 1, 2, \ldots, m-1\}.$$

The global matrix V is partitioned into sub-matrices $V^{(k)}$ accordingly to the domain decomposition approach, i.e. given the grid partition S_k, the matrix $V^{(k)}$ works on the nodes of this grid partition and is mapped to the corresponding MPI process. An index of the global matrix V is mapped onto the grid partition S_k via

$$h_k := g_{\Omega_h} \circ g_{S_k}^{-1}.$$

Thus, the following equalities hold for all elements $j \in \{0, 1, 2, \ldots, N_k\}$, for all components $l \in \{0, 1, 2, \ldots, m-1\}$, and for all grid partitions $k \in \{0, 1, 2, \ldots, q-1\}$ (see also Fig. 5):

$$V_{j,l}^{(k)} = V_{h_k(j),l} \ , \qquad V_{j,l} = V_{h_k^{1-}(j),l}^{(k)} \ .$$

The sub-matrix $V^{(k)}$ is represented by the class template

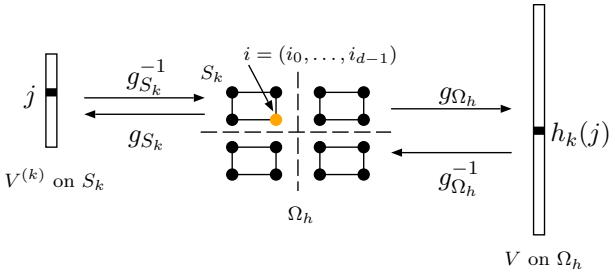

Fig. 5: Mappings of element j of the (local) vector $V^{(k)}$ related to grid partition S_k to element $h_k(j)$ of the (global) vector V related to grid global Ω_h and vice versa.

`DataField<CT,DIM>`. The template parameter `CT` can be replaced by the ScaFES type `ScaFES::Ntuple<CT,MM>` in order to handle the above vector-valued physical fields or by basic data types like `double` in order to handle

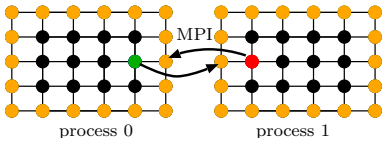

Fig. 6: Synchronization of a function value at the boundary of both grid partitions The corresponding grid nodes at the boundary of each partition are colored red resp. green, ghost grid nodes are colored orange.

real-valued physical fields, too. The function values at all grid nodes are stored continuously in memory. These values can be easily set and accessed just by passing the positions of the grid nodes (see Listing 5). There are two variants for each access method: According to the class template `std::vector<CT>`, the method `operator()` returns the addressed component, directly, whereas the method `at()` performs an additional range check. For enabling access to

```
ScaFES::DataField<double,3> V;
ScaFES::Ntuple<int,3> idxNode(0); V(idxNode) = 0;
int i(0), j(0), k(0);              V(i,j,k) = 0;
```

Listing 5: Possibilities to set the elements of a three-dimensional vector to zero at the first node of a grid.

function values on a different grid partition, the function values at additional grid nodes ("halos") are stored (see Fig. 6). This speeds up the computations as one does not have to fetch the necessary values again and again. Send and receive buffers are provided for the halos in order to exchange the function values between the grid partitions. The communication is implemented asynchronously. As the structures of the send and receive buffers do not change over time, the Boost.MPI skeleton concept [10] is used. Within this concept, the contents are separated from the structures and thus, the contents have to be exchanged only once, resulting in an improved MPI communication [11]. The class template consists of several grids and sub-grids like the grid partition S_k represented by `mIdxSetNormal` or the sub-grids of all ghost grid nodes represented by `mIdxSetGhost` (see Fig.**??**). This has the advantage that the function values at these grid nodes can be easily accessed using the corresponding grid iterators. In particular, the elements of the sub-matrix $V^{(k)}$ can be traversed accordingly to the demands of the communication using the iterators of the member variables `mIdxSetComm` and `mIdxSetGhost`. The data exchange is done if and only if there are at least two grid partitions and the stencil width of the data field is not equal to zero.

D. Representation of Initial Boundary Value Problems

As we consider time-dependent problems, we have to discretize the time interval $[t_S, t_E]$. Let $n_\tau \in \mathbb{N}$ the number of time steps. Then, for a given time step $l \in \{0, 1, 2, \ldots, n_\tau - 1\}$, we get the time $t_l = t_S + l \cdot \tau$ for a time step size $\tau := (t_E - t_S)/(n_\tau - 1)$. We denote the set of all times t_l by τ_h. Let $u_h : \Omega \times [t_S; t_E] \to \mathbb{R}$ be an approximation of u.

```
// Program parameters.
ScaFES::Parameters mParams;
// Global grid.
ScaFES::GridGlobal<DIM> mGG;
// Grid of all nodes.
ScaFES::Grid<DIM> mIdxSetAll;
// sub grid of all normal nodes.
ScaFES::GridSub<DIM> mIdxSetNormal;
// Sub grid of all boundary nodes.
std::vector< ScaFES::GridSub<DIM> > mIdxSetBorder;
// Grid of all ghost nodes.
std::vector< ScaFES::GridSub<DIM> > mIdxSetGhost;
// Grid of all nodes to be communicated.
std::vector< ScaFES::GridSub<DIM> > mIdxSetComm;
// Number of ghost layers.
int mNghostLayers;
// Pointer to the memory of the vector.
CT* mElemData;
// Buffers for sending and receiving.
std::vector<ScaFES::Buffer<CT>> mValuesToExchange;
// Output file for writing vectors.
ScaFES::DataFile<CT, DIM> mOutput;
```

Listing 6: Member variables of the class template `ScaFES::DataField<CT,DIM>`.

Then, defining the matrices

$$U_{j,q}^{(l)} := [u_h(x_j, t_l)]_q, \qquad F_{j,q}^{(l)} := [f(x_j, t_l)]_q,$$
$$G_{j,q}^{(l)} := [g(x_j, t_l)]_q, \qquad \widetilde{U}_{j,q} := [\widetilde{u}(x_j)]_q$$

for all $x_j \in \Omega_h$ and for all $t_l \in \tau_h$, and using a numerical method like finite differences in space leads to the following system of equations for all time steps l:

$$\begin{aligned} U_j^{(l+1)} &= \left(A(U^{(l)})\right)_j - F_j^{(l)} && \forall j \in \mathcal{G}_I(\Omega_h), \\ U_j^{(l+1)} &= G_j^{(l)} && \forall j \in \mathcal{G}_B(\Omega_h), \\ U_j^{(0)} &= \widetilde{U}_j && \forall j \in \mathcal{G}(\Omega_h) \end{aligned}$$

The matrix $A \in \mathbb{R}^{N,N}$ results from the discretization in space. A depends on the current iterate $U^{(l)}$. If the differential operator F in the initial boundary value problem is a linear one, then A depends linear on $U^{(l)}$, too. The discretization of the underlying problems using a numerical method like the finite difference method very often results in a sparse system matrix A, i.e. only the values at direct neighboring nodes are needed for the computation of the new iterate at an interior grid node. In fact, the matrix A will never be set up, but the matrix vector product $AU^{(l)}$ will be computed in each time step. All equations are independent from each other. Thus, the computation of the new iterate $U^{(l+1)}$ can be done completely in parallel. In particular, one can reorder the system of equations such that the communication and the computations can be done concurrently on each grid partition S_k:

- Compute $U_j^{(k;0)} = \widetilde{U}_j^{(k)}$ at all grid nodes $j \in \mathcal{G}$.
- Perform for all time steps $l \in \{0, 1, 2, \ldots, n_\tau - 1\}$:
  - Compute iterate $U_j^{(k;l+1)}$ at all (partitions related) boundary nodes $j \in \mathcal{G}_B(S_k)$,
  - Copy values $U_j^{(k;l+1)}$ at all boundary nodes $\mathcal{G}_B(S_k)$ to the send buffers.
  - Exchange values of the send buffers with all directly neighboring grid partitions using non-blocking sends and receives
  - Compute iterate $U_j^{(k;l+1)}$ at all (partitions related) interior nodes $j \in \mathcal{G}_I(S_k)$,
  - Wait until all communication calls have been finished.
  - Copy values from the receive buffers to halos of current iterate $U^{(k;l+1)}$.
  - Swap old iterate $U^{(k;l)}$ and new iterate $U^{(k+1;l)}$.

Initial boundary value problems are represented by the class template `Problem<PRBLM,CT,DIM>`. The class template makes use of the so called "curiously recurring template pattern" [12]. As a consequence, the user has to implement an own class inherited by this class template. This derived class has to contain the methods given in Listing 8. The template parameter `CT` represents the data type of all involved data fields and `DIM` represents the space dimension d. The old and new iterate at each time step are stored in the two member variables `mVectOld` and `mVectNew`. The type of all nodes (interior, boundary) of the grid partition is stored in a member variable named `mNodeType`. (see Listing 7). Matrices like $G^{(l)}$ can be added to the problem using the method `addDataField()`. This method requires the name of the physical field, its stencil width, and a flag if the field is an unknown one or not. The stencil width directly corresponds to the number of ghost layers at the boundary of a grid partition (see Fig. 6). Amongst other,

```
// Program parameters.
ScaFES::Parameters mParams;
// Global grid.
ScaFES::GridGlobal<DIM> mGG;
// Type of grid nodes.
ScaFES::DataField<short int, DIM> mNodeType;
// Old iterate U^{(k,l)} at grid partition S_k.
std::vector< ScaFES::DataField<CT, DIM> > mVectOld;
// New iterate U^{(k+1,l)} at grid partition S_k.
std::vector< ScaFES::DataField<CT, DIM> > mVectNew;
```

Listing 7: Member variables of the class template `ScaFES:Problem<PRBLM,CT,DIM>`.

there are access methods named `gridsize()` and `tau()` for the grid sizes h_p and the time step size τ. Known data fields can be accessed using the method `knownDf()`. The above algorithm over all time steps is implemented in the method `iterate()`, and the mapping c is implemented in the method `connect()`. In order to control program runs, the most important parameters can be read in from the command line. This has the advantage that one does not have to compile a program again if the program should be executed with a different parameter set. Furthermore, shell scripts can be easily created for parametrized test runs (like weak or strong scalability tests). The class `Parameter` represents a set of command line parameters.

E. Summarizing Used Parallelization Techniques

In the end of this section, we summarize the three used parallelization techniques.
On top, we adopt a domain decomposition approach by dividing the global grid into several grid partitions which are

```
template<typename T>
void updateInner(
   std::vector<ScaFES::DataField<T,DIM>>& v1,
   std::vector<ScaFES::DataField<T,DIM>> const& v0,
   ScaFES::Ntuple<int,DIM> const& idxNode,
   int const& timestep
);
template<typename T>
void updateBorder(
   std::vector<ScaFES::DataField<T,DIM>>& v1,
   std::vector<ScaFES::DataField<T,DIM>> const& v0,
   ScaFES::Ntuple<int,DIM> const& idxNode,
   int const& timestep
);
```

Listing 8: Methods which must be implemented by user in the derived problem class.

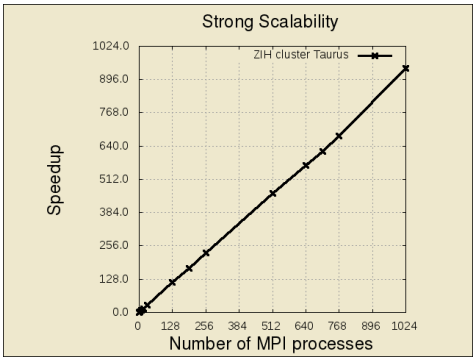

Fig. 7: Strong scaling w.r.t. MPI of the considered three-dimensional heat equation problem on ZIH cluster Taurus.

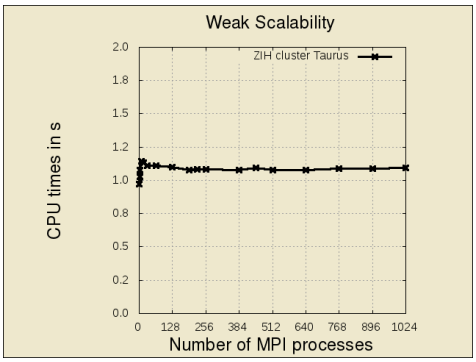

Fig. 8: Weak scaling w.r.t. MPI of the considered three-dimensional heat equation problem on ZIH cluster Taurus.

mapped via a one-to-one relation onto a given number of MPI processes such that each MPI process computes a portion of the global problem (cp. subsection II-B). This technique addresses distributed as well as shared memory systems. On the node level, we use OpenMP work sharing constructs in order to parallelize the traversing and computation of values of physical fields on one grid partition. On the core level, we vectorize small loops using the compiler SIMD vectorization. Nowadays, modern compilers like the GCC or ICC can SIMD-vectorize many loops automatically if these loops are written in an appropriate way. Thus, we decided to support this automatical compiler vectorization and prepared the loops in the framework. i.e. we used for- instead of do-while-loops, removed dependencies between involved elements within the loops e.g.
The user can choose between a pure MPI, a pure OpenMP and a mixed mode parallelization to adopt and benefit from the underlying hardware architecture like a cluster system with shared memory nodes and distributed memory across nodes or a full shared memory system.

III. Solving A d-dimensional Heat Equation Problem

In order to show how an initial boundary value problem can be solved using ScaFES, we consider the d-dimensional heat equation on the d-dimensional unit hypercube for arbitrary $d \in \mathbb{N}$. Given the time interval $[0;1]$, the domain $\Omega := (0,1)^d$, the source $f : \bar{\Omega} \times (0;1] \to \mathbb{R}$, $f(x,t) := 0$, the boundary condition $g : \partial\Omega \times (0;1] \to \mathbb{R}$, $g(x,t) := 0$, and the initial condition $\widetilde{u} : \Omega \to \mathbb{R}$, $\widetilde{u}(x) := \prod_{i=0}^{d-1} x_i \cdot (x_i - 0.5)^2 \cdot (1 - x_i)$, then the task is to find $u : \bar{\Omega} \times [0;1] \to \mathbb{R}$ such that the following system of equations is fulfilled:

$$\begin{aligned} \partial_t u - \Delta u &= f && \text{in } \Omega \times (0;1], \\ u &= g && \text{on } \partial\Omega \times (0;1], \\ u(\cdot, t_S) &= \widetilde{u} && \text{in } \Omega. \end{aligned}$$

We discretized this system of equations in space using the finite difference method with the standard centered stencil (7-point stencil in 3D, e.g.) and in time using the explicit Euler scheme. Therefore, we defined

$$\begin{aligned} U_j^{(l)} &:= u(x_j, t_l), & F_j^{(l)} &:= f(x_j, t_l), \\ G_j^{(l)} &:= g(x_j, t_l), & \widetilde{U}_j &:= \widetilde{u}(x_j). \end{aligned}$$

for all $x_j \in \Omega_h$, and for all $t_l \in \tau_h$. The resulting system of equations reads for all time steps $l \in \{0, 1, \ldots, n_\tau\}$:

$$\begin{aligned} U_j^{(l+1)} &= \tau \cdot \sum_{p=0}^{d-1} \big(-2 \cdot U_j^{(l)} + U_{c(j;2\cdot p))}^{(l)} + U_{c(j;2\cdot p+1))}^{(l)} \big) / h_p^2 \\ &\quad - \tau \cdot F_j^{(l)} + U_j^{(l)} \quad \forall j \in \mathcal{G}_I(\Omega_h), \\ U_j^{(l+1)} &= G_j^{(l)} \quad \forall j \in \mathcal{G}_B(\Omega_h), \\ U_j^{(0)} &= \widetilde{U}_j \quad \forall j \in \mathcal{G}(\Omega_h). \end{aligned}$$

The implementation of this problem is given in Listing 9. We emphasize that this implementation is not bound to a certain storage data type and a certain space dimension due to the employment of the class template parameters `CT` and `DIM`.

IV. Performance Results

In the last section, we demonstrated that ScaFES is very user-friendly. Users do not need any knowledge about parallelization techniques but can concentrate on the implementation of the numerical algorithms. But what is the price to pay for this rapid and simple prototyping? Does a ScaFES application really scale and can it compete against applications which are using one of the more general and heavyweight software packages mentioned in section I? In order to figure it out, we performed different scaling tests w.r.t. MPI and OpenMP and compared ScaFES to PETSc, representing a state-of-the-art software package.
To show the scalability of ScaFES, we used the implementation

```
#include "ScaFES.hpp"
% template<typename CT, std::size_t DIM> // Source f.
inline void funcF(CT& fx, ScaFES::Ntuple<CT,DIM> const& x, CT const& t) {
  fx = 0.0;
}
template<typename CT, std::size_t DIM> // Boundary condition g.
inline void funcG(CT& fx, ScaFES::Ntuple<CT,DIM> const& x, CT const& t) {
  fx = 0.0;
}
template<typename CT, std::size_t DIM> // Initial condition \tilde{u}.
inline void funcUt(CT& fx, ScaFES::Ntuple<CT,DIM> const& x, CT const& t) {
  fx = 1.0;
  for (std::size_t pp = 0; pp < DIM; ++pp) {
    fx *= (x[pp] * (x[pp] - 0.5) * (x[pp] - 0.5) * (1.0 - x[pp]));
}
template<typename CT, std::size_t DIM> // Own problem class.
class HeatEqnFDM : public ScaFES::Problem<HeatEqnFDM<CT,DIM>, CT, DIM> {
  public:
    HeatEqnFDM(ScaFES::Parameters const& cl,
               ScaFES::GridGlobal<DIM> const& gg)
      : ScaFES::Problem<HeatEqnFDM<CT,DIM>, CT, DIM>(cl, gg) {
        this->addDataField("F", 0, funcF<CT,DIM>, true);
        this->addDataField("G", 0, funcG<CT,DIM>, true);
        this->addDataField("U", 1, funcUt<CT,DIM>, false);
      }
    template<typename TT> // Method must be implemented!
    void updateInner(std::vector<ScaFES::DataField<TT,DIM>>& vNew,
                     std::vector<ScaFES::DataField<TT,DIM>> const& vOld,
                     ScaFES::Ntuple<int,DIM> const& idxNode,
                     int const& timestep) {
      vNew[0](idxNode) = vOld[0](idxNode)
                         - this->tau() * this->knownDf(0, idxNode);
      for (std::size_t pp = 0; pp < DIM; ++pp) {
        vNew[0](idxNode) += this->tau() * (
                 -2.0 * vOld[0](idxNode)
                     + vOld[0](this->connect(idxNode, 2*pp))
                     + vOld[0](this->connect(idxNode, 2*pp+1)) )
                     / (this->gridsize(pp) * this->gridsize(pp));
      }
    }
    template<typename TT> // Method must be implemented!
    void updateBorder(std::vector<ScaFES::DataField<TT,DIM>>& vNew,
                      std::vector<ScaFES::DataField<TT,DIM>>const& vOld,
                      ScaFES::Ntuple<int,DIM> const& idxNode,
                      int const& timestep) {
      vNew[0](idxNode) = this->knownDf(1, idxNode);
    }
};
int main(int argc, char *argv[]) {     // Main program.
  ScaFES::Parameters pp(argc, argv);   // Read in command line options.
  ScaFES::GridGlobal<3> gg(pp);        // Create grid partitions.
  HeatEqnFDM<double,3> prblm(pp, gg);  // Create 3D heat eqn. problem.
  prblm.iterate();                     // Iterate over all time steps.
  return 0;
}
```

Listing 9: Source code in ScaFES for solving the three-dimensional heat equation on the three-dimensional unitcube.

of the three-dimensional heat equation problem as shown in section III as test case with purely MPI parallelization. The strong and weak scaling tests w.r.t. MPI were performed on island 1 of the HPC system Taurus at ZIH which is based on Intel® Sandy Bridge multi-core chips with 16 cores per shared-memory node and a total of 4320 cores [13]. The application was compiled with GCC 4.8.0 and highest optimization level. All measurements refer to computing 20 time steps, the computational domains were partitioned in the third dimension, the times for initialization and output of simulation results are not considered. We discretized the computational domain using $128 \times 128 \times 8192$ nodes as fixed workload for the strong scaling test and used a fixed grid of $128 \times 128 \times 8$ nodes per process for the weak scaling test. The results in Fig. 7 and in Fig. 8 show that the application indeed scales weakly and strongly.

The strong scaling tests w.r.t. OpenMP were performed on the HPC cluster Atlas at ZIH as this cluster possesses 64 cores per shared-memory node [14]. We used the implementation of the corresponding two-dimensional heat equation problem as test case. The application was compiled with GCC 4.7.1 and highest optimization level. Again, all measurements refer

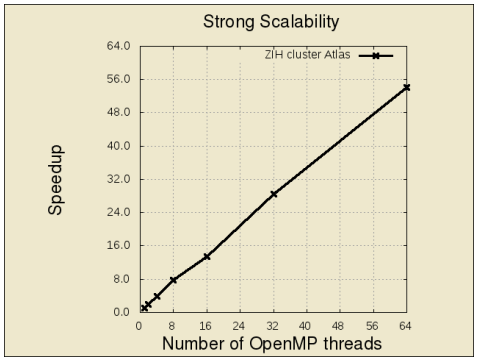

Fig. 9: Strong scaling w.r.t. OpenMP of a two-dimensional heat equation problem on ZIH cluster Atlas.

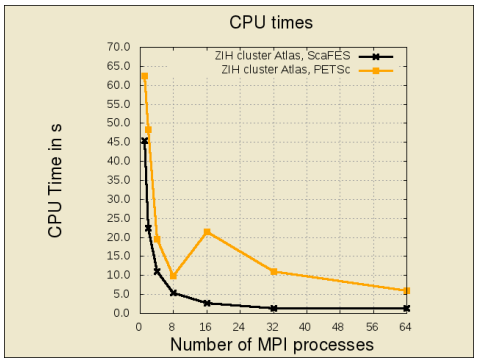

Fig. 10: Comparison of ScaFES with PETSc for a two-dimensional heat equation problem on ZIH cluster Atlas.

to computing 20 time steps, the computational domains were partitioned in the second dimension. The times for initialization and output of simulation results are not considered. We discretized the computational domain using 4096×4096 nodes as fixed workload. The results in Fig. 9 show that the application scales strongly w.r.t. OpenMP, too.

We have seen that ScaFES indeed is a high-scaling framework. But will it be as fast as one of the other existing frameworks? Therefore, we implemented the above heat equation problem in two dimensions in PETSc 3.4.3, too. The performance test was again run on the ZIH cluster Atlas. Fig. 10 shows that implementation in ScaFES is not only comparable to PETSc, but outperfoms it for the considered example, for 64 processes by a factor of approximately 7.

All source codes, scripts and results of the performance tests are provided at `tu-dresden.de/zih/scafes` such that the presented results can be reproduced by interested people.

V. Conclusions and Outlook

We described the principal design aspects of the HPC framework for explicit solvers on structured grids named ScaFES and showed its good scalability. By presenting an implementation example, we illustrated the user-friendly interfaces. The development resp. design of ScaFES was driven by the ambitions to create a high quality and scalable software tool, which is easy to use and is portable to various platforms and architectures. The underlying parallelization is encapsulated and hidden from the user. The user has to implement serial code, only. The parallelization and communication is managed by ScaFES. Because of its user-friendliness, ScaFES can be used as a rapid prototyping tool to evaluate and compare numerical methods as well as to write high quality production code without loosing scalability and efficiency.

Load balancing is normally a key aspect of adaptive mesh methods. If it should be necessary, one could achieve load balancing by introducing a cost function to the domain decomposition algorithm.

Acknowledgments

ScaFES is funded by the Federal Ministry of Education and Research (BMBF) within the project HPC-FliS under the support code 01 IH 11 009.

References

[1] S. Vey and A. Voigt, "AMDiS: adaptive multidimensional simulations," *Computing and Visualization in Science*, vol. 10, no. 1, pp. 57–67, 2007.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework." *Computing*, vol. 82, no. 2-3, pp. 103–119, 2008.

[4] K. Yee, "Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, May 1966.

[5] ISO, *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, last checked on 2013-11-14 (07:30 CET). [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[6] B. Dawes, D. Abrahams, and R. Rivera, "Boost C++ Libraries Homepage," last checked on 2013-11-02 (07:12 CET). [Online]. Available: http://www.boost.org

[7] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor, "The Goat Book," 2000, last checked on 2013-10-18 (16:12 CET). [Online]. Available: http://sources.redhat.com/autobook/

[8] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2010.

[9] M. J. Berger and S. H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors." *IEEE Trans. Computers*, vol. 36, no. 5, pp. 570–580, 1987, last checked on 2013-11-08 (12:30 CET). [Online]. Available: http://dblp.uni-trier.de/db/journals/tc/tc36.html#BergerB87

[10] M. Gauckler and D. Egloff, "The Meat and Bones of Message Passing," Sep. 2006, last checked 2013-10-28 (15:44 CET). [Online]. Available: http://daveabrahams.com/files/2010/09/meat_and_bones_of_mpi.pdf

[11] M. Flehmig, "Framework zur effizienten parallelen Berechnung expliziter orts- und zeitdiskreter Verfahren," Diploma Thesis, Center For Information Services And High Performance Computing At TU Dresden, 12 2011.

[12] J. O. Coplien, "Curiously Recurring Template Patterns," *C++ Report*, 1995.

[13] Center For Information Services And High Performance Computing At TU Dresden, "HPC Web-Compendium: Cluster Taurus," last checked 2013-11-14 (13:22 CET). [Online]. Available: https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus

[14] ——, "HPC Web-Compendium: Cluster Atlas," last checked 2013-10-28 (15:30 CET). [Online]. Available: https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareAtlas

[15] R. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A Limited Memory Algorithm for Bound Constrained Optimization," *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, 1995.