HighPerMeshes – A Domain-Specific Language for Numerical Algorithms on Unstructured Grids

Samer Alhaddad[1], Jens Förstner[1], Stefan Groth[2], Daniel Grünewald[3], Yevgen Grynko[1], Frank Hannig[2](✉), Tobias Kenter[1], Franz-Josef Pfreundt[3], Christian Plessl[1], Merlind Schotte[4], Thomas Steinke[4], Jürgen Teich[2], Martin Weiser[4], and Florian Wende[4]

[1] Paderborn Center for Parallel Computing and Department of Computer Science and Department of Electrical Engineering, Paderborn University, Paderborn, Germany

[2] Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany
{frank.hannig,stefan.groth}@fau.de

[3] Fraunhofer Institut für Techno- und Wirtschaftsmathematik, Kaiserslautern, Germany

[4] Zuse Institute Berlin, Berlin, Germany

Abstract. Solving partial differential equations on unstructured grids is a cornerstone of engineering and scientific computing. Nowadays, heterogeneous parallel platforms with CPUs, GPUs, and FPGAs enable energy-efficient and computationally demanding simulations. We developed the HighPerMeshes C++-embedded Domain-Specific Language (DSL) for bridging the abstraction gap between the mathematical and algorithmic formulation of mesh-based algorithms for PDE problems on the one hand and an increasing number of heterogeneous platforms with their different parallel programming and runtime models on the other hand. Thus, the HighPerMeshes DSL aims at higher productivity in the code development process for multiple target platforms. We introduce the concepts as well as the basic structure of the HighPerMeshes DSL, and demonstrate its usage with three examples, a Poisson and monodomain problem, respectively, solved by the continuous finite element method, and the discontinuous Galerkin method for Maxwell's equation. The mapping of the abstract algorithmic description onto parallel hardware, including distributed memory compute clusters, is presented. Finally, the achievable performance and scalability are demonstrated for a typical example problem on a multi-core CPU cluster.

Keywords: Domain-specific language · Numerical algorithms · Unstructured grids · Parallel computing

B. Balis et al. (Eds.): Euro-Par 2020 Workshops, LNCS 12480, pp. 185–196, 2021.
https://doi.org/10.1007/978-3-030-71593-9_15

1 Introduction

Simulations of physical systems described by partial differential equations (PDEs) are the cornerstone of computational science and engineering. The ever-growing need for computational performance due to the increasing number and scale of simulations has led to the rise of different and heterogeneous parallel computing platforms, ranging from multi-core CPUs to massively parallel distributed systems and from SIMD vector units to GPUs and FPGAs. Adapting complex simulation algorithms to and implementing them efficiently on these different architectures is a demanding task requiring in-depth computer science knowledge that is usually not directly available to numerical mathematicians and computational engineers. Consequently, many large scale simulation codes address only a narrow and often traditional range of computing environments, missing the performance opportunities offered by new architectures.

In this paper, we present the HighPerMeshes embedded DSL that provides an abstraction layer to C++ application developers to implement efficient mesh-based algorithms for PDE problems on unstructured grids. The focus of the DSL is on finite element (FE) and discontinuous Galerkin (DG) or finite volume (FV) discretizations to address iterative and matrix-free solvers as well as time-stepping schemes. Large parts of PDE simulation problems thus can be covered. HighPerMeshes draws heavily on the C++17 standard and template metaprogramming for genericity and extensibility. Additionally, compile-time information through template parameters can benefit the code generation for specific target architectures.

The following other software projects address PDE computations on unstructured grids: Traditional library approaches such as deal.II [1], DUNE [3], or Kaskade 7 [7] focus on application building blocks and usually provide a rather explicit parallelization based on threads or MPI, providing one or a few selected back ends such as PETSc [2]. High-level DSLs such as FEniCS [16] or FreeFEM [10] on the other hand, allow to specify PDE problems in very abstract notation and use code generation techniques to create efficient simulation programs. The projects closest in scope and intention are the OP2 [17]/PyOP2 [20] and Liszt [6] DSLs, by providing interfaces to execute local compute kernels on unstructured meshes and to access data associated with different mesh entities. These approaches depend on C++, Python, and Scala code transformation and compilation techniques [19]. In contrast, we rely on template metaprogramming methods.

2 The HighPerMeshes Domain-Specific Language

Picking the right abstraction level is central for every DSL or library interface targeting mesh-based algorithms for PDEs. It needs to provide idioms for specifying the algorithmic building blocks on an abstraction level that is high enough to be mapped efficiently to different computing environments. Furthermore, it should be detailed enough to allow implementing a wide range of established or yet to be developed discretizations schemes and numerical algorithms. The HighPerMeshes DSL aims at providing abstractions on a level that is just high enough to allow for an efficient mapping to sequential and multithreaded CPU execution, distributed memory systems, and accelerators. On this level, the core components of mesh-based PDE algorithms include mesh data structures, the association of Degrees of Freedom (DoFs) to mesh entities such as cells and vertices, and the definition of kernel functions that encapsulate local computations with shape functions defined on single mesh cells or faces.

2.1 Mesh Interface

Computational meshes decompose the computational domain $\Omega \subset \mathbb{R}^d$ into simple shapes such as triangles or tetrahedra by which PDE solutions can be represented. Unstructured meshes do so in an irregular pattern that can be adapted to complex geometries or local solution features in a flexible way. Unlike for structured meshes, neighborhood relations between these cells are not implied by the storage arrangement of their constituting vertices, but are usually defined through connectivity lists that specify how they are made up of vertices. Therefore, the storage efficiency of unstructured meshes can be very low if the specifics of the hardware architecture are not taken into account. Similarly, when accessing or iterating over mesh entities (cells, faces, edges, and vertices for $d = 3$), the memory structuring and arrangement of, e.g., geometrically neighboring entities can be critical to performance and present optimization targets on the mesh implementation for different architectures.

The construction of a mesh in the HighPerMeshes DSL starts from a set of vertices $V = \{\boldsymbol{v}_m \in \mathbb{R}^d\}$ and a set $C = \{\boldsymbol{i}_n | \, n = 0, \ldots, \#\text{cells}-1\}$ of connectivity lists $\boldsymbol{i}_n \subset \{0, \ldots, |V| - 1\}$ representing the cells in the mesh.

Users can create meshes by providing V and C directly or by using one of the available import parsers for common mesh data files. Each $\boldsymbol{i} \in C$ references into the vertex set V to encode an entity of the cell dimensionality $d_{\text{cell}} \leq d$. Sub-entities or constituting entities like edges and faces correspond to index sets $\boldsymbol{j} \subset \boldsymbol{i} \in C$ that are deduced according to a particular scheme that is specific to the entity type. All entities are stored in a $(d_{\text{cell}} + 1)$-dimensional set data-structure using their index sets. The mesh manages a lookup table which for each entity holds the IDs of all its constituting entities with one dimension lower, and another with the IDs of all incident super-entities, if present.

Users of the DSL can define their own entity types by implementing the interfaces `EntityTopology` and `EntityGeometry`. The two interfaces define the base functionality that is needed by the DSL, e.g., to navigate through all the different entities in the mesh or provide face normals. Entity-specific extensions can be added easily, which enhances the usability of the DSL.

For the hierarchical definition of entities as an affiliation of sub-entities, `EntityTopology` and `EntityGeometry` must know the actual type of their implementation for explicit instantiations, e.g., requesting or providing information about entities of different dimensionality.

On top of the mesh implementation is the mesh partitioning, which is needed for work distribution in the parallel context. The `PartitionedMesh` type inherits all functionality and state from the `Mesh` type. It selects from C a subset of the entities in the mesh and redirects this subset to the `Mesh` base type.

Iterator ranges over entities of any valid dimension can be created by the mesh and any valid entity through

```
template<int Dimension>
EntityRange<Dimension> GetEntities(){..}.
```

Both `EntityT` and (`Partitioned`)`Mesh` extend this functionality in different ways, thereby enabling the user of the DSL to query topological and geometrical information inside and outside of the kernel functions.

2.2 Buffer Types for Storing Coefficient Vectors

PDE solutions are generally discretized using finite-dimensional ansatz spaces and are represented by coefficient vectors with respect to a certain basis. In FE, FV, and DG methods, the basis functions are associated with mesh entities and have a support contained in the union of the cells incident to their entity. The mapping of coefficients, or Degrees of Freedom (DoFs) to storage locations and access to them depends on the target architecture and may involve nontrivial communication. Therefore, the DSL provides buffer types for coefficient vector storage to relieve the user from these considerations.

Depending on the ansatz space, a particular number of basis functions is associated with mesh entities of different dimensions. Therefore, the number of coefficients $\eta_{\tilde{d}}$ associated to entities of dimension $\tilde{d} \in \{0, \ldots, d_{\text{cell}}\}$ has to be specified when constructing a buffer. Additionally, global values as coefficients of the constant basis function can be stored, e.g.,

```
Runtime hpm{..};
auto dofs = MakeDofs<1,1,1,1,2>(); /* η = {η⃗_d̃, 2} = {1,1,1,1,2} */
auto buffer = hpm.GetBuffer<float>(mesh,dofs);
```

for $d_{\text{cell}} = d = 3$. The buffer holds one value of type `float` for each node, edge, face, and the cell itself. Two additional entries are provided for global values.

DoFs are accessed through a "local-view object" (`lv` in Listing 1, line 7) inside kernel functions. These local views are a tuple of implementation-defined objects that are accessible with the `GetDof` function, which requests DoFs of a certain dimension. This is necessary because access patterns may provide DoFs associated with mesh entities of different dimensions.

Given a data access pattern (Sect. 2.3) and a specific entity—typical program executions loop over all or a subset of the entities in the mesh, one after the other—the corresponding local view makes for a linearly indexable type inside the kernel function, thereby hiding data layout and storage internals.

2.3 Iterating over the Mesh with Local Kernels

In the PDE solver algorithms that we target, a significant part of PDE computation on meshes involves the evaluation of values, derivatives, or integrals on cells or faces, and is therefore local. This allows for various kinds of parallelization, depending on the target architecture. Typically, these local calculations in space are embedded into time-stepping loops or iterative algorithms, which imply dependencies based on the data access patterns of the kernels. With a scheduler that suitably resolves these dependencies, additional parallelism can be exploited by partially overlapping subsequent time steps.

In HighPerMeshes, the application developer specifies the calculations as local kernels at entity granularity and invokes a dispatcher to take care of their parallel execution and scheduling. Line 1 of Listing 1 shows the definition of a distributed dispatcher that uses the command line arguments to set up its environment. The advantage of using this dispatcher model is a complete separation of parallelization techniques and kernel definitions. The interface is technology-agnostic and does not require knowledge about parallel programming.

The dispatcher's `Execute` method takes a number of kernels to be executed as its arguments. If required, those arguments might be supplemented by a range of time steps, as shown in line 3 of Listing 1, in order to iterate the defined sequence of kernels more than once in the specified range. Each kernel must define a range of entities to iterate over. To enable flexible parallelization strategies, the DSL does not guarantee a processing order for these entities. For example, the function call `mesh.GetEntityRange<CellDimension>()` in line 5 specifies that the dispatcher iterates over all cells. `ForEachEntity` in line 4 defines an iteration over all entities in that range. Here, HighPerMeshes provides another option: `ForEachIncidence<D>` iterates over all sub-entities of a certain dimension `D` for the entities in the given range.

The kernel requires a tuple of access definitions, as seen in line 6. Access definitions specify the mode (any of `Read`, `Write`, and `ReadWrite`) and the access pattern for the DoF access. This allows the scheduler to calculate dependencies between kernels, thereby avoiding conflicting DoF accesses in scatter operations despite parallelization. Access patterns determine the DoFs relevant for the calculation by specifying a set of mesh entities incident or adjacent to the local entity. `Cell` in line 6 means that the kernel requires access to the DoFs from the given `buffer` that are associated with the local cell, as frequently used in DG methods. Other common access patterns involve a local cell and all of its incident sub-entities, usually encountered in FE methods, or the two cells incident to a face for flux computations in DG or FV methods. While HighPerMeshes aims at providing all access patterns necessary for common kernel descriptions in FE or DG methods, they can be easily extended by providing the required neighborhood relationship in the mesh interface.

Lastly, the user must define a kernel to be executed (line 7). It must be a callable that takes the specified entities, time steps, and a local-view object `lv` as its arguments. The latter allows access to the requested DoFs.

```
1 DistributedDispatcher dispatcher{argc,argv};
2 dispatcher.Execute(
3   Range{100},
4   ForEachEntity(
5     mesh.GetEntityRange<CellDimension>(),
6     tuple(Write(Cell(buffer))),
7     [](const auto& cell,auto step,auto& lv) { /*kernel body*/ }));
```

Listing 1. Example of a dispatcher definition and kernel execution.

3 Using the DSL

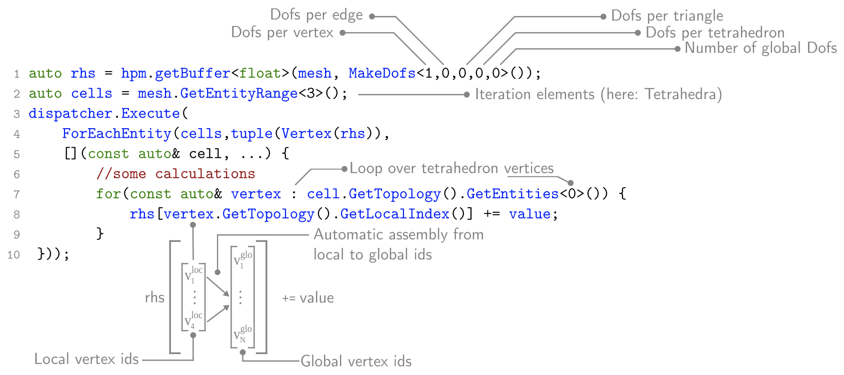
In this section, examples and code segments are presented to illustrate the methods described in Sect. 2 and to explain their use. Further information about the algorithms and examples can be found in the public repositories[1,2].

3.1 Matrix-Free Solver for the Poisson Equation

For illustrating the usage of the DSL, the elliptic Poisson problem

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^3, \qquad u = 0 \quad \text{on } \Gamma \subset \mathbb{R}^3 \tag{1}$$

with homogeneous Dirichlet boundary conditions is solved by a matrix-free conjugate gradient (CG) method [11,21]. By discretizing (1) with linear finite elements on a tetrahedralization of Ω, i.e. with one DoF per vertex, a system $Ax = b$ of linear equations is obtained [5]. Since A is symmetric and positive definite, its solution is the minimizer of the convex minimization problem $F(x) = \frac{1}{2}x^T Ax - b^T x \rightarrow \min$.



```
 1 auto rhs = hpm.getBuffer<float>(mesh, MakeDofs<1,0,0,0,0>());
 2 auto cells = mesh.GetEntityRange<3>();
 3 dispatcher.Execute(
 4     ForEachEntity(cells,tuple(Vertex(rhs)),
 5     [](const auto& cell, ...) {
 6         //some calculations
 7         for(const auto& vertex : cell.GetTopology().GetEntities<0>()) {
 8             rhs[vertex.GetTopology().GetLocalIndex()] += value;
 9         }
10  }));
```

Fig. 1. Code segment for right-hand side computation.

In order to solve this equation system, the right-hand side (*rhs*) b must be assembled. This is done using the buffer datatype and the loop *ForEachEntity*, which iterates over the vertices of each cell (in this case, tetrahedra) and stores the corresponding value in the buffer (Fig. 1 code line 8). The homogeneous Dirichlet boundary conditions can be built into the rhs here as well. To solve the system, a matrix-free CG iteration is used. Its main algorithmic building block is the computation of matrix-vector products Ax. Instead of assembling A and

[1] https://github.com/HighPerMeshes/highpermeshes-dsl.
[2] https://github.com/HighPerMeshes/highpermeshes-drts-gaspi.

performing linear algebra operations, we assemble the product Ax directly by evaluating

$$s_j = \underbrace{\int_C \nabla\phi_i \nabla\phi_j \; dC}_{A(i,j)_{cell}} \cdot x_j \tag{2}$$

per cell and with ϕ_* as shape functions (see line 9 of Listing 2). The same procedure is used for all further matrix-vector products. Finally, the result can be saved into a file and visualized using, for example, *ParaView*.

```
 1 auto AssembleMatrixVecProduct =
 2     ForEachEntity(cells, tuple(Vertex(s)),
 3     [](const auto& cell) {
 4         auto& indices = cell.GetTopology().GetVertexIndices();
 5         for (int col = 0; col < ncols; ++col)  {
 6             for (int row = 0; row < nrows; ++row)  {
 7                 float a_ij = .. //set a_ij using shape functions
 8                 s[indices[col]] += a_ij * x[indices[row]];
 9         }}
10     });
```

Listing 2. Example of a matrix-free computation.

3.2 Discontinuous Galerkin Time Domain (DGTD) Maxwell Solver

Here we sketch an implementation of a Maxwell solver based on the DGTD numerical scheme [9,12]. An initial value problem is solved in the time domain in a free space mesh with perfect electric conductor (PEC) boundary conditions. The user can modify the code accordingly if field sources, materials, or absorbing boundaries are needed. The simulation domain is discretized in a triangular or tetrahedral mesh, which is used as an input. Then, DoFs or calculation points are created within the cells, depending on the ansatz order specified by the user. For example, a three-dimensional simulation with third-order accuracy requires 20 DoFs in each cell to represent the unknown fields. The right-hand sides of Maxwell's equations are evaluated during Runge-Kutta time integration at each time step according to the DGTD method formulation

$$\dot{\mathbf{E}} = \mathbf{D} \times \mathbf{H} + (\mathcal{M})^{-1}\mathcal{F}\left(\Delta\mathbf{E} - \hat{n} \cdot (\hat{n} \cdot \Delta\mathbf{E}) + \hat{n} \times \Delta\mathbf{H}\right) \tag{3}$$

$$\dot{\mathbf{H}} = -\mathbf{D} \times \mathbf{E} + (\mathcal{M})^{-1}\mathcal{F}\left(\Delta\mathbf{H} - \hat{n} \cdot (\hat{n} \cdot \Delta\mathbf{H}) + \hat{n} \times \Delta\mathbf{E}\right) \tag{4}$$

Here $\mathbf{D} \times \mathbf{H}$ and $\mathbf{D} \times \mathbf{E}$ are the curls of the magnetic and electric fields. Correspondingly, $\mathcal{M}$ is the mass matrix, $\mathcal{F}$ the face matrix, $\Delta\mathbf{E}, \Delta\mathbf{H}$ are field differences between the neighboring cells at the interfaces, and $\hat{n}$ the face normal [9]. The first term (the curls) involves only cell-local DoFs and is therefore called "volume kernel" (see Listing 3).

```
auto volumeKernelLoop = ForEachEntity(cells,
    tuple(Read(Cell(H)),Cell(rhsE),..),
    [&](const auto& cell,..,auto& lv){
        Mat3D D = cell.GetGeometry().GetInverseJacobian()*2.0;
        ForEach(numVolumeNodes,[&](const int n){
            const auto& H = GetDofs<3>(get<0>(lv));
            Mat3D dH;
            ForEach(numVolumeNodes,[&](const int m) {
                dH += DyadicProduct(derivative[n][m],H[m]);
            });
            auto& rhsE = GetDofs<3>(get<1>(lv));
            rhsE[n] += Curl(D,dH);
            // code for rhsH: analogue  to rhsE
        });
    });
```

Listing 3. Code segment for the Maxwell volume kernel.

The second term in (3, 4), the "surface kernel," (see Listing 4) stems from a surface integral over the cell's faces, and involves those DoFs from within the two incident cells located on these faces. Calculating the surface kernel requires some operations provided directly by the DSL, e.g., `GetNormal()`. The implementation complexity of DG on unstructured meshes comes from the access or mapping to the neighboring cells DoFs in order to calculate fluxes across faces as described in (3, 4). This access is performed with the data structure `NeighboringNodeMap` (line 12 in Listing 4), which provides the corresponding index for the DoFs in the local view.

```
 1 auto surfaceKernelLoop = ForEachIncidence<2>(cells,
 2     tuple(Read(ContainingMeshElement(H)),
 3         Read(ContainingMeshElement(E)),
 4         Read(NeighboringMeshElementOrSelf(H)),
 5         Read(NeighboringMeshElementOrSelf(E)),
 6         Write(ContainingMeshElement(rhsE))),
 7     [&](const auto& cell, const auto& face,..,auto& lv){
 8         const auto& H = GetDofs<3>(get<0>(lv));
 9         // buffer access to E, nH, n, E, rhsE is analogous
10         auto& NeighboringNodeMap {DgNodeMap.Get(cell,face)};
11         int faceIndex = face.GetTopology().GetLocalIndex();
12         auto faceUnitNormal = face.GetGeometry().GetUnitNormal();
13         auto edg = (face.GetGeometry().GetNormal()*2.0/
14         cell.GetGeometry().GetAbsJacobianDeterminant()).Norm()
               *0.5;
15         ForEach(numSurfaceNodes,[&](const int m){
16             const auto dH = edg*Delta(H,nH,m,NeighboringNodeMap);
17             const auto dE =
18             edg*DirectionalDelta(E,nE,face,m,NeighboringNodeMap);
19             const auto fluxE = (dE-(dE*faceUnitNormal)*
                   faceUnitNormal+CrossProduct(faceUnitNormal,dH));
20             ForEach(numVolumeNodes,[&](const int n){
21                 rhsE[n] += LIFT[face_index][m][n]*fluxE;
22             });
23         });
24     });
```

Listing 4. Code segment for the Maxwell surface kernel.

3.3 Finite Elements for Cardiac Electrophysiology

The excitation of cardiac muscle tissue is described by electrophysiology models such as the monodomain model

$$\begin{aligned} \dot{u} &= \nabla \cdot (\sigma \nabla u) + I_{\text{ion}}(u, w), \\ \dot{w} &= f(u, w), \end{aligned} \tag{5}$$

where σ is the conductivity, I_{ion} is the ion current that forms together with the gating dynamics $f(u, w)$ the membrane model. The simplest FitzHugh-Nagumo membrane model defines $I_{\text{ion}}(u, w) = u(1-u)(u-a)-w$ and $f(u, w) = \epsilon(u-bw)$ with $0 < a, b, \epsilon < 1$ [4,15,23].

The method of lines [22] discretizes the monodomain model (5) first in space and then in time. For the discretization of space, we use linear finite elements again, leading to the system

$$\begin{aligned} M\dot{u} &= \sigma A u + M \cdot I_{\text{ion}}(u, w) \\ \dot{w} &= f(u, w) \end{aligned}$$

with mass matrix M and stiffness matrix A. For time discretization, the forward Euler method

$$u_{t+1} = u_t + \tau \underbrace{\left(M^{-1}\sigma A u_t + I_{\text{ion}}(u_t, w_t)\right)}_{\dot{u} := u_d}, \quad w_{t+1} = w_t + \tau f(u_t, w_t) \tag{6}$$

is widely used in cardiac electrophysiology due to its simplicity and its stability for reasonable step sizes τ [18].

In order to avoid inverting the globally coupled mass matrix, the row-sum mass lumping technique is applied to M [13]. This yields a diagonal approximation M_l of M and allows for efficient, explicit formation of M_l^{-1} to be used in (6) instead of M^{-1}, and matrix-free storage in vector form. The right-hand side u_t including the matrix-vector product Au_t is assembled directly as in (2) without forming A:

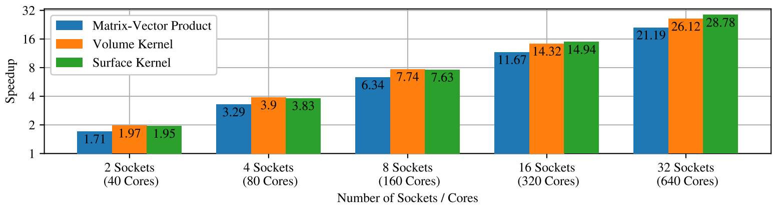
```
auto fwEuler = ForEachEntity(
    mesh.GetEntityRange<0>(),
    tuple(Vertex(u), Vertex(Read(u_d))),
    [&](const auto& vertex, auto step, auto& lv) {
        auto& u = GetDofs<0>(get<0>(lv));
        auto& u_d = GetDofs<0>(get<1>(lv));
        u[0] += tau * u_d[0];
    });
```

Listing 5. Code example of an implementation of a first-order solver (forward Euler).

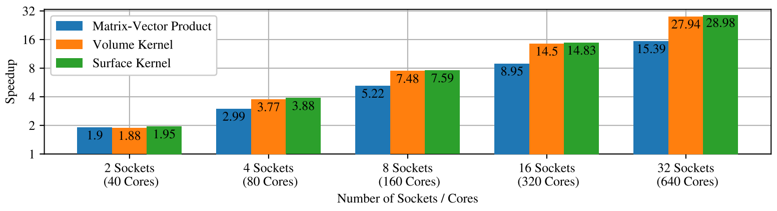
3.4 Distributed Scalability Experiments

In this section, we analyze the distributed scalability of the matrix-vector product (Listing 2), the volume kernel (Listing 3), and the surface kernel (Listing 4).

The experiments were executed on a cluster, where each compute node consists of two sockets. Each socket contains an Intel Xeon Gold 6148 "Skylake" CPU, which has 20 cores and a base frequency of 2.4 GHz. Hyper-threading is deactivated. The nodes are connected on a 100 Gb/s Intel Omni Path network. All experiments were executed with 20 threads per socket, as the scalability of our threading approach on a single compute node has already been shown [8].



(a) Acceleration with ACE's thread pool



(b) Acceleration with OpenMP

Fig. 2. Speedup for iterating over the specified kernels on a mesh with 400,000 tetrahedra and 1000 time steps on an increasing amount of sockets compared to executing the same kernels on one socket. The evaluated back ends use ACE's thread pool (a) or OpenMP (b).

We conducted the experiments for 1000 time steps on a synthetic mesh of 400,000 tetrahedra. Such a setup represents a typical problem size targeted by the distributed dispatcher. For mesh partitioning, we use the Metis library [14].

Figure 2 shows the speedup over a single node for the distributed dispatcher when either scheduling tasks to ACE's thread pool or accelerating tasks with OpenMP for an increasing amount of compute nodes. As a baseline for each experiment, we measure the execution time with both back ends on a single socket, i.e., 20 cores, and use the faster one. For 640 cores, the back end feeding threads to ACE's thread pool achieves better speedups for the matrix-vector product with a speedup of 21.19. The volume and surface kernels achieve a better speedup in the case of OpenMP acceleration, with a speedup of 27.94 and 28.98, respectively. Furthermore, the volume and surface kernels scale better than the matrix-vector multiplication because they are more compute-intensive. They iterate over 20 DoFs instead of just one. To achieve this kind of scalability, the dispatcher requires a sufficient workload.

The results show that HighPerMeshes allows an efficient distribution of matrix-free algorithms. They also show that the provided abstractions are not tailored to a specific back end. Instead, both reference implementations achieve similar speedups, thus showing that the language is portable to different technologies.

Conclusion

HighPerMeshes is an embedded DSL that provides high-level abstractions for iterative, matrix-free algorithms on unstructured grids. It is a powerful tool enabling users to run simulations and implement their own modifications for complex multi-scale problems from a broad range of application domains.

The data structures and procedures provided by HighPerMeshes allow efficient parallelization and distribution as shown by our implementation of a dispatcher that distributes kernels with the help of GASPI, ACE, and OpenMP. This gives the user the opportunity to take advantage of complex parallelization techniques and task scheduling without being an expert on parallelization, saving implementation time and effort on one side, and offering flexibility for different computing platforms without the need for code modification on the other side.

Acknowledgments. This work was partially funded by the German Federal Ministry of Education and Research (BMBF) within the collaborative research project "HighPerMeshes" (01IH16005). The authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing (PC^2).

References

1. Arndt, D., et al.: The `deal.II` library, version 9.1. J. Numer. Math. **27**(4), 203–213 (2019)
2. Balay, S., et al.: PETSc (2019). https://www.mcs.anl.gov/petsc
3. Bastian, P., et al.: A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. Computing **82**(2), 121–138 (2008). https://doi.org/10.1007/s00607-008-0004-9
4. Dauby, P., Desaive, T., Croisier, H., Kolh, P.: Standing waves in the FitzHugh-Nagumo model of cardiac electrical activity. Phys. Rev. E **73**(2), 021908 (2006)
5. Deuflhard, P., Weiser, M.: Adaptive Numerical Solution of PDEs. Walter de Gruyter, Berlin (2012)
6. DeVito, Z., et al.: Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Proceedings of Conference on High Performance Computing Networking, Storage and Analysis (SC 2011), p. paper 9. ACM (2011)
7. Götschel, S., Schiela, A., Weiser, M.: Kaskade 7 - a flexible finite element toolbox. Comp. Math. Appl. **81**, 444–458 (2020)
8. Groth, S., Grünewald, D., Teich, J., Hannig, F.: A runtime system for finite element methods in a partitioned global address space. In: CF 2020: Proceedings of the 17th ACM International Conference on Computing Frontiers. ACM (2020). https://doi.org/10.1145/3387902.3392628

9. Grynko, Y., Förstner, J.: Simulation of second harmonic generation from photonic nanostructures using the discontinuous Galerkin time domain method. In: Agrawal, A., Benson, T., De La Rue, R.M., Wurtz, G.A. (eds.) Recent Trends in Computational Photonics. SSOS, vol. 204, pp. 261–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55438-9_9
10. Hecht, F.: New development in FreeFem++. J. Numer. Math. **20**(3–4), 251–265 (2012). https://freefem.org/
11. Hestenes, M.R., Stiefel, E., et al.: Methods of conjugate gradients for solving linear systems. J. Res. Natl. Bur. Stand. **49**(6), 409–436 (1952)
12. Hesthaven, J.S., Warburton, T.: Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications. Springer, New York (2008). https://doi.org/10.1007/978-0-387-72067-8
13. Hughes, T.J.: The Finite Element Method: Linear Static and Dynamic Finite Element Analysis. Courier Corporation (2012)
14. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. (SISC) **20**(1), 359–392 (1998)
15. Liu, F., Zhuang, P., Turner, I., Anh, V., Burrage, K.: A semi-alternating direction method for a 2-D fractional FitzHugh-Nagumo monodomain model on an approximate irregular domain. J. Comput. Phys. **293**, 252–263 (2015)
16. Logg, A., Mardal, K.A., Wells, G.N., et al.: Automated Solution of Differential Equations by the Finite Element Method. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-23099-8
17. Mudalige, G.R., Giles, M.B., Reguly, I., Bertolli, C., Kelly, P.H.J.: OP2: an active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In: Proceedings of Innovative Parallel Computing (InPar), pp. 1–12 (2012)
18. Puwal, S., Roth, B.J.: Forward Euler stability of the bidomain model of cardiac tissue. IEEE Trans. Biomed. Eng. **54**(5), 951–953 (2007)
19. Rathgeber, F., et al.: Firedrake: automating the finite element method by composing abstractions. ACM Trans. Math. Softw. (TOMS) **43**(3), 24:1–24:27 (2016)
20. Rathgeber, F., et al.: PyOP2: a high-level framework for performance-portable simulations on unstructured meshes. In: Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), pp. 1116–1123. ACM, November 2012
21. Saad, Y.: Iterative Methods for Sparse Linear Systems, vol. 82. SIAM (2003)
22. Schiesser, W.E., Griffiths, G.W.: A Compendium of Partial Differential Equation Models: Method of Lines Analysis with Matlab. Cambridge University Press, Cambridge (2009)
23. Sermesant, M., Coudière, Y., Delingette, H., Ayache, N., Désidéri, J.A.: An electromechanical model of the heart for cardiac image analysis. In: Niessen, W.J., Viergever, M.A. (eds.) MICCAI 2001. LNCS, vol. 2208, pp. 224–231. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45468-3_27