SIAM J. SCI. COMPUT.
Vol. 42, No. 3, pp. C69–C96

ENCLAVE TASKING FOR DG METHODS ON DYNAMICALLY ADAPTIVE MESHES*

DOMINIC ETIENNE CHARRIER†, BENJAMIN HAZELWOOD†, AND TOBIAS WEINZIERL†

Abstract. High-order discontinuous Galerkin (DG) methods promise to be an excellent discretization paradigm for hyperbolic differential equation solvers running on supercomputers, since they combine high arithmetic intensity with localized data access, since they straightforwardly translate into nonoverlapping domain decomposition, and since they facilitate dynamic adaptivity without the need for conformal meshes. An efficient parallel evaluation of DG weak formulation in an MPI+X setting, however, remains nontrivial as dependency graphs over dynamically adaptive meshes change with each mesh refinement or coarsening, as resolution transitions yield nontrivial data flow dependencies, and as data sent along domain boundaries through message passing (MPI) have to be triggered in the correct order. Domain decomposition with MPI alone starts to become insufficient if the mesh changes very frequently, if mesh changes cannot be predicted, and if limiters and nonlinear per-cell solves yield unpredictable costs per cell. We introduce enclave tasking as a task invocation technique for shared memory and MPI+X: It does not assemble any task graph; instead the mesh traversal spawns ready tasks directly. A marker-and-cell approach ensures that tasks feeding into MPI or triggering mesh modifications as well as latency-sensitive or bandwidth-demanding tasks are processed with high priority. The remaining cell tasks form enclaves, i.e., groups of tasks that can be processed in the background. Enclave tasking introduces high concurrency which is homogeneously distributed over the mesh traversal, it mixes memory-intensive volumetric DG calculations with compute-bound Riemann solves, and it helps to overlap communication with computations. Our work focuses on ADER-DG and patch-based finite volumes. Yet, we discuss how the paradigm can be generalized to the whole DG family and finite volume stand-alone solvers.

Key words. MPI+X, communication-hiding, ADER-DG, discontinuous Galerkin, patch-based finite volumes, dynamical AMR

AMS subject classifications. 68W40, 65Y20, 68W10

DOI. 10.1137/19M1276194

1. Introduction.

Higher-order discontinuous Galerkin (DG) techniques contribute towards the success story of many solvers for hyperbolic partial differential equations (PDEs) on supercomputers. DG methods are considered to be guarantors for computational efficiency. While they fit to dynamically adaptive (block-structured) grids [12]—no conformity constraints are imposed conceptually—DG methods' high-performance computing (HPC) selling point is that they combine high arithmetic intensity with localized data access. Its computations per mesh cell are arithmetically intense, which is a property they share with many higher-order methods [25]. At the same time, DG's data access pattern however is very localized [11]—this helps to reduce the memory access stress [8, 17, 20, 24, 27]—and its exchange between cells along their connecting faces is conceptually simple. A combination of these two properties—high intensity to exploit vector units and dynamic adaptive mesh refinement (AMR) to invest where it pays off most—is a fit to predictions of what exascale software will have to look like [10].

*Submitted to the journal's Software and High-Performance Computing section September 4, 2019; accepted for publication (in revised form) February 25, 2020; published electronically May 5, 2020.

https://doi.org/10.1137/19M1276194

Funding: This work was supported by the European Unions Horizon 2020 research and innovation programme under grant 671698 (ExaHyPE).

†Department of Computer Science, Durham University, Durham DH1 3LE, UK (DominicEtienne.Charrier@amd.com, ben.hazelwood@featurespace.co.uk, tobias.weinzierl@durham.ac.uk).

DG's localized data exchange is a fit to distributed memory, message-based (MPI) parallelization as it is predominant in supercomputing. Delivering scaling algorithm implementations per node as well as upscaling MPI+X yet remain far from trivial. DG traverses the grid to evaluate its algorithmic steps. Such traversals can be read as task graph traversals: The computational grid spans the graph, whereas the particular DG scheme defines the task type per mesh entity. DG in its basic form distinguishes two tasks: tasks working on cells and tasks working on faces (Riemann solves) [9, 24]. Mapping the task types onto separate mesh traversals makes steps piping data through the cores (Riemann) take turns with the computationally demanding volumetric evaluations. While the latter scale, the other steps tend to be bandwidth- or latency-bound. Furthermore, they couple cells and thus the demand for data exchange. They are thus typically incapable of exploiting all cores, they hinge on interconnect capabilities, and their scalability potential is limited. High polynomial degrees and static adaptivity with a volumetric coupling of cells allow the cost of the cell operations to marginalize the cost of data exchange, Riemann solves, and so forth [9, 19]. Overlapping domain decomposition with volumetric coupling is not studied here. Nevertheless, cheap task phases continue to introduce a low concurrency workload fraction in an Amdahl sense and thus constrain the scalability.

Most codes that achieve high performance focus on particular DG subcategories and master the challenges above by exploiting the subcategories' particular characteristics: If we study classic finite volume (FV) schemes, the Riemann solve and the volume integral can be run in parallel [17, 24], i.e., we can hide the former behind the expensive volumetric computation. If we study linear PDEs, the cost per cell is known a priori, as all cell and face operators are small matrices [19, 24, 43]. This simplifies the decomposition and scheduling of operations [7]. From a task point of view, static on-node scheduling then is sufficient. If we study static adaptive meshes, we know prior to each time step where computationally intense interpolations and restrictions arise that feed into other tasks. We in particular know which face data are to be exchanged via MPI which enables us to prioritize the handling of the underlying computations appropriately such that they are sent out while we continue to work locally [16, 17, 19, 20, 25, 32]. We can even design hardware topology-aware domain splits [40]. If we furthermore stick to conformal meshes plus global time stepping or even regular grids, such operations disappear completely [13, 19, 43]. These success stories show that DG has great upscaling potential. They also show that it is reasonable to reorder and intermix the tasks to obtain high performance.

Any rearrangement or parallelization of the task execution requires care as intergrid transfer operations along mesh resolution boundaries have to be performed in the correct order. We typically interpolate the coarser data representation, then solve Riemann problems, and finally restrict the outcome. Furthermore, tasks sending and receiving MPI messages have to stick to a specific order. The fact that there are "cheap" tasks, i.e., tasks with low arithmetic intensity, further implies that MPI sends have to be issued early to allow the message transfer to hide behind the expensive tasks or many cheap ones. Finally, memory-intense tasks such as mesh refinement or the Riemann solves shall continuously trickle through the system to avoid memory access bursts. Assembling the whole task graph or fractions of it and then deriving a tailored/optimized schedule without assembly penalty is difficult if totally dynamic; unconstrained AMR may change the graph in each and every time step and (almost) any location in the computational domain. The above paragraph gives examples of successful strategies if we constrain the AMR. Some sophisticated DG variants furthermore employ techniques such as (a posteriori) limiting [15], optimistic time stepping

with on-the-fly CFL analysis which occasionally requires roll-backs, or solving nonlinear equation systems locally with a dynamic termination criterion controlling the nonlinear solve. The cost per volume is not known a priori. Local time stepping is beyond the scope of the present work yet can be seen as a technique which amplifies all task balancing and scheduling difficulties.

We propose a grid traversal and task invocation scheme called *enclave tasking*. It works without any task graph assembly, and it makes no assumption about the grid topology. We introduce it by means of dynamically adaptive meshes as they result from octrees as well as generalizations—we call them spacetrees—of those [45, 46], and by means of ADER-DG [13], an explicit time stepping scheme for hyperbolic equation systems. The spacetree is traversed cellwisely. This allows for many efficient storage and traversal schemes [12, 18, 46, 45]. Enclave tasking maps each DG time step onto a pair of mesh traversals. The primary mesh traversal runs over all mesh cells and spawns one task per cell. Work stealing then distributes these computationally intense tasks among idle threads. We realize a producer-consumer pattern. Additionally, the traversal launches the computationally cheap Riemann solves ad hoc, i.e., per face read: it waits for the tasks of the adjacent cells to terminate, and then it immediately runs the bandwidth-demanding computations. Where required, the primary mesh traversal processes a temporarily shifted task graph [9]: It runs the Riemann solves, and then immediately issues the cell tasks of the subsequent time step. Cell tasks update the solution within their cell, which implies a change of the solution representation along the cell faces. In our nonoverlapping domain decomposition, these face data have to be sent out in a deterministic, consistent order to neighboring ranks, as the subsequent primary mesh traversal on the neighboring rank feeds its local data plus the remote counterpart data as input to the Riemann solves. We thus classify the aforementioned cell tasks into high priority and background tasks, and we introduce a secondary (partial) mesh traversal. It takes turns with the primary traversal. Whenever the secondary traversal accesses a cell along a domain boundary, it waits for its local adjacent task to complete and then passes the outcome immediately to MPI. Tasks of cells adjacent to MPI boundaries are issued with high priority. Furthermore, we assume that dynamic adaptivity spreads along existing grid transitions most of the time. It tends to evolve smoothly in space and time. Therefore, we also make cell tasks along refinement transitions have high priority if they feed into a mesh interpolation. Their outcome is processed by the secondary mesh traversal, too. This means that information from interpolation along resolution boundaries becomes available early. Other cells are skipped by the secondary traversal. As our mesh traversals itself are parallelized, too, we end up with three different types of tasks spread over two different types of mesh traversals: Memory-intense tasks tied to the (parallel) mesh traversals, high priority cell tasks, and background cell tasks. The high priority tasks stem from cells adjacent to MPI boundaries and adaptivity. They form *skeletons*. The remainder cells form *enclaves*. They are tasks that are handled in the background of all communication- and bandwidth-critical operations. They deliver the scalability.

The whole paper describes a geometrically inspired multitasking scheme which exploits mesh regularity and the fact that meshes typically do not change dramatically all over the domain within a short time interval. The idea to process "communicating cells" prior to others is, notably in the context of DG and accelerators, not new [2, 20, 25, 40], and many of the present ideas, per se, are well known. To the best of our knowledge, however, there is no work that combines all of the following features into one formalism. Our work (i) derives and updates regularity information—the

skeletons—on-the-fly, and thus imposes no constraints on the dynamic adaptivity; in our code, the adaptivity along subdomain boundaries can change in each and every grid sweep; (ii) does not make any assumptions about the grid structure/topology or restricts itself to particular subgrid regions/enclaves [25, 38, 40]; (iii) works without any assembly of a task graph which becomes expensive if dynamic adaptivity makes this graph change in each and every grid sweep [26, 27, 38] and yet supports very inhomogeneous, unpredictable cost-per-cell distributions; (iv) mixes tasks with different compute characteristics and thus avoids memory access bursts; (v) is MPI-oblivious: As MPI data aligns along the mesh skeleton, MPI data is sent out while prediction tasks still might queue in the background. Therefore, sends are overlapped with computation. Notably, enclave tasking does not have to know the MPI communication pattern a priori, which would be a showstopper for totally free dynamic adaptivity. While our discussion focuses on ADER-DG, a particular flavor of explicit time stepping DG schemes, we sketch that our techniques impact many matrix-free DG methods and notably all FV schemes, too. While our discussion focuses on spacetrees, the extensions to forests [3, 23, 37] is straightforward if we classify all tree, i.e., interforest, boundaries as skeletons.

The remainder of the paper is organized as follows: We sketch ADER-DG and the operators, i.e., tasks of interest (section 2), before we introduce enclave tasking in section 3. Section 4 next describes how we tailor the tasking runtime and use MPI. In section 5, we generalize all patterns to other DG approaches and finite volumes (FVs) and then provide measurements (section 6) that demonstrate the potential of our ideas. A conclusion summarizes the main findings and sketches future work as well as shortcomings.

2. ADER-DG on Cartesian meshes.

We study first-order hyperbolic systems

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) = S(Q) + \sum \delta \qquad \text{with } Q : \mathbb{R}^{d+1} \mapsto \mathbb{R}^m, d \in \{2, 3\}. \tag{1}$$

F is a conservative flux, S is a volumetric source term, and δ denotes the impact of point sources. S and δ usually depend on time and space. Q is the solution over a d-dimensional computational domain. It has m components and changes over time. The system is complemented by well-suited initial and boundary conditions.

Among DG techniques for (1), ADER-DG [14, 15] has grown into a popular approach. ADER-DG relies on particularly expensive volumetric cell operators, as it solves the PDE per mesh cell per time step through a weak space-time formulation. For nonlinear PDEs, this even requires a nonlinear equation system solve. Space-time solves are computationally feasible as the mesh cells are handled independently of each other. The solve, however, is only a space-time predictor which feeds into a follow-up, explicit-in-time Riemann phase. It solves the Riemann problems arising from the discontinuous, predicted solution along the mesh faces. Eventually, both solve outcomes are merged into the next time step's solution. This step is labeled the corrector. ADER-DG exhibits high-order behavior in both space and time, and it is arithmetically intense per cell, although it requires only one data exchange between adjacent cells per time step.

2.1. ADER-DG sketch.

We study (1) over a computational domain Ω_h discretized by a mesh that consists of cuboid cells c. Each cell carries a Q approximation $Q_h(x, t)$ as a linear combination of Lagrangian polynomial shape functions over Gauß–Legendre points. The polynomials are continuous inside the cells, but they induce jumps along the faces between cells. Our derivation of the algorithmic steps

and computational tasks—each step consists of tasks which are atomic work units independent of all other tasks—derives from [13].

ADER-DG starts from a weak formulation of (1) both in space and in time. Let $(T, T+\Delta T)$ span one time step. We obtain a continuous, weak formulation

$$(2)\quad \int_{\Omega\times(T,T+\Delta T)} \left(\frac{\partial Q}{\partial t} + \nabla\cdot F(Q)\right)\hat{v}\,\mathrm{d}(x,t) = \int_{\Omega\times(T,T+\Delta T)} \left(S(Q)+\dots\right)\hat{v}\,\mathrm{d}(x,t),$$

where we replace $Q(\cdot,T)$ by the linear combination of shape functions Q_h, and where we vary a space-time test polynomial $\hat{v}$. To develop Q_h in time, we multiply it with a polynomial in time [14]. The same Lagrangian polynomial order as for the spatial representation is used. With cubes as mesh cells, this is a tensor product approach for the space-time solution. It describes a space-time polynomial $\hat{Q}_h$ approximating the development of the real solution Q over time and space. $\hat{v}$ uses the same ansatz, i.e., we express $Q_h(x,t)$ or $\hat{Q}_h(x,t)$, respectively, from (2) by a weak, discretized space-time Ritz–Galerkin problem with space-time test functions $\hat{v}_h$.

Following [13], we separate the time derivative from the remainder integrals in (2). Partial integration in time for the term comprising $\frac{\partial Q}{\partial t}$, and partial integration in space for all other terms, gives us two computational/implementational advantages. (i) It injects the known solution $Q_h(\cdot,T)$ into the system and yields an explicit expression for $Q_h(\cdot,T+\Delta T)$, as we roll over the time derivative to the test function. (ii) It removes the divergence operator $\nabla\cdot$ from F as it transfers to the test function. F evaluations for applications are straightforward—they typically describe the physics directly—while a derivative computation can be tedious. The two advantages are accompanied by two disadvantages: (i) The overall scheme is high-order in time but describes a globally implicit setup which is usually infeasible to solve. (ii) We inherit jump terms from the partial integration in space. ADER-DG addresses these two disadvantages numerically.

The first step of ADER-DG is the *space-time predictor* (STP). It develops $\hat{Q}_h(x,t)$ by solving the weak, discretized space-time Ritz–Galerkin problem. Yet, it drops all jump terms when it integrates

$$\int_{c\times(T,T+\Delta T)} \left(\frac{\partial Q_h}{\partial t}\right)\hat{v}\,\mathrm{d}(x,t) + \int_{c\times(T,T+\Delta T)} \left(\nabla\cdot F(Q_h)\right)\hat{v}\,\mathrm{d}(x,t) = \dots$$

by parts. With $\bigcup c = \Omega_h$ tessellating the computational domain, the arising STP decouples the individual cells from each other. We ignore the neighbors of any cell c. For a given $Q_h(\cdot,T)$, this yields a space-time $\hat{Q}_h^*$ through Picard iterations.

As $Q_h, \hat{Q}_h, \hat{Q}_h^*$ are all represented by polynomials with compact support that are allowed to be discontinuous along cell faces, no continuity constraints between cells are built into $\hat{Q}_h^*$. Jumps arise if we extrapolate the STP $\hat{Q}_h^*$ from left and right to the faces between neighboring cells. Such projections are labeled $\hat{Q}_h^{*\pm}$ in DG—one projection from the right and one from the left along a coordinate axis. As $\hat{Q}_h^{*\pm}$ is a space-time polynomial, the extrapolation is a space-time expression, too. To make ADER-DG an explicit time stepping scheme, we replace all $\hat{Q}_h$ entries in the "jump" terms arising from partial spatial integration of (2) with our predicted $\hat{Q}_h^*$. These face interface states then are plugged into a Riemann solver. This is the second step of the ADER-DG scheme. We solve the *Riemann* problems.

In the third and final step of ADER-DG, we plug both $\hat{Q}_h^*(\cdot,T+\Delta T)$ and the time integral over the Riemann solution into (2), integrate in time, and solve the

remaining weak formulation. It degenerates to a spatial problem over Ω_h. This step can be read as a correction to the predicted $\hat{Q}_h^*(\cdot, T+\Delta T)$. It is thus called *corrector*.

2.2. A task language. Let C_{STP} denote

$$\hat{Q}_h^* = C_{\text{STP}}\, Q_h(\cdot, T). \tag{3}$$

In (3), we use the symbol C_{STP} as a global operator applied to the solution over the whole computational domain. However, the STP's construction implies that C_{STP} decomposes over the cells. Consequently, we use C_{STP} synonymously for a computational task which advances the solution over one cell: $\hat{Q}_h^*|_c = C_{\text{STP}} \cdot Q(\cdot, T)|_c$. We omit $|_c$ henceforth. As a result, the global C_{STP} evaluation results from the application of a set of cellwise C_{STP} tasks to all $c \in \Omega_h$. Though C_{STP} formally spawns the whole STP, follow-up steps use solely $Q_h^*(\cdot, T+\Delta T) = id|_{T+\Delta T} \cdot \hat{Q}_h^*$ and the two projections $id|_{\partial c} \cdot \hat{Q}_h^*$ of the STP onto each face between any two cells. It is convenient only to store these results [9], i.e., to extrapolate—if we employ Gauss–Legendre points, no sample points coincide directly with the faces—and to integrate over time immediately.

Let F_{R} denote the operator that runs over all faces. It represents the Riemann solves. For global time stepping, where all cells advance in time with the same time step size ΔT, it is convenient to make it comprise the time integral over the result, too. Along the lines of (3), we observe that F_{R} decomposes over the mesh faces. F_{R} consequently describes a set of compute tasks over all faces. They accept input from the C_{STP} tasks.

The *corrector* finally yields a set of cellwise C_{Corr} tasks. We end up with

$$\begin{aligned} Q_h(\cdot, T+\Delta T) &= C_{\text{Corr}} \circ F_{\text{R}} \circ id|_{\partial c} \cdot Q_h^* + id|_{T+\Delta T} \cdot Q_h^* \\ &= (C_{\text{Corr}} \circ F_{\text{R}} \circ id|_{\partial c} + id|_{T+\Delta T}) \circ C_{\text{STP}} \cdot Q_h(\cdot, T) \end{aligned}$$

for one ADER-DG time step. Alternatively, we may distinguish the data stored inside the cell from the data held for the faces by writing them as entries of a vector:

$$\begin{aligned} \begin{pmatrix} Q_h \\ \cdot \end{pmatrix}(T+\Delta T) = & \underbrace{\begin{pmatrix} 1 & C_{\text{Corr}} \\ 0 & 0 \end{pmatrix}}_{\text{corrector (cellwise)}} \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & F_{\text{R}} \end{pmatrix}}_{\text{Riemann (facewise)}} \\ & \underbrace{\begin{pmatrix} id|_{T+\Delta T} \\ id|_{\partial c} \end{pmatrix} C_{\text{STP}} \begin{pmatrix} 1 & 0 \end{pmatrix}}_{\text{STP (cellwise)}} \begin{pmatrix} Q_h \\ \cdot \end{pmatrix}(T). \end{aligned} \tag{4}$$

Face data here is used as temporary data storage and thus does not determine the solution at particular time stamps. Once we are given a mesh, (4) describes the arising ADER-DG task graph. Alternatively, this ADER-DG blueprint maps onto a plain realization employing three loops (Algorithm 1), each issuing independent tasks.

2.3. Dynamic adaptivity. While adaptive mesh refinement (AMR) minimizes computational work, it adds complexity to the task graph. For ADER-DG, we are however able to exploit the discontinuities built inherently into the numerical scheme to bring down AMR implementation complexity from a parallelization point of view. We do not impose any balancing conditions [41]. Yet, we do assume that we have a grid topology which is common to many software packages.

Algorithm 1 Pseudocode of ADER-DG split up into three phases. We highlight what we refer to as C and F tasks. Some technical details (projections and temporary variables) from this algorithm are omitted in the text for clarity. Without local time stepping, time integration and Riemann solve F_{R} can be switched, i.e., we can collapse Riemann solves over time into one spatial Riemann problem.

```
1: function AderDGTimeStep(ΔT)
2:     for all cells c ∈ Ω_h do                                  ▷ Space-time predictor phase
3:         startTask
4:         C_STP(c)          ▷ Run predictor on cell. C_STP is parameterized with ΔT
5:         id_∂c(c)          ▷ Project result to 2d faces of c, keep outcome at T + ΔT,
6:                                  ▷ too, but throw away intermediate time solutions
7:         endTask
8:     end for
9:     for all faces f ∈ Ω_h do                                  ▷ Riemann phase
10:        startTask
11:        Read id|_∂c outcome from f's adjacent cells          ▷ Q_h^{*±} in DG literature
12:        F_R(f)            ▷ Riemann solves over the whole (T, T + ΔT) time span
13:        Integrate Riemann outcome in time
14:        endTask
15:    end for
16:    for all cells c ∈ Ω_h do                                  ▷ Correction phase
17:        startTask
18:        Read F_R outcome of 2d adjacent cells               ▷ Project Riemann solve
19:        C_Corr(c)         ▷ outcomes back to cell, and fuse with predicted data
20:        endTask
21:    end for
22: end function
```

Assumption 1. *Let our grids result from a conformal grid. We assign this grid the level ℓ_{min}. From ℓ_{min} on, we construct a finer, adaptive grid by subdividing each cell that we want to refine a fixed number of times such that the subdivisions along two adjacent cells that are refined match. This new grid has level $\ell_{min}+1$. We continue recursively but independently for each cell. When the recursion has terminated, all cells that are not subdivided further form an adaptive mesh Ω_h. Each cell $c \in \Omega_h$ belongs to a unique level.*

The purest grids of this type are quadtrees and octrees. They start from one square or cube, i.e., a trivial ℓ_{min} mesh, and subdivide this base cell along each coordinate axis once per refinement step. They eventually yield an adaptive Cartesian mesh where all cuboids of one level have exactly the same size. Our grid assumption includes a forest of trees where we start from a conformal mesh and embed octrees into its cells. The extension to more sophisticated subdivision or boundary-fitted meshes is straightforward. The hyperbolic ecosystem around [4] imposes a grid topology suiting our assumption, too. For our experiments, we stick to sole trees. Unlike the traditional bipartitioning, we use three-partitioning [45, 46].

Along each resolution transition of the resulting fine grid Ω_h, we can uniquely identify cells of the coarse and the fine resolutions. AMR now becomes a strict extension not altering any building blocks introduced so far: As the STP and the correction are tied to cells, they are agnostic of AMR. For the Riemann problems, we

first introduce *virtual cells*. Virtual cells are subcells of real cells which have the same size, i.e., face lengths, as their adjacent real cells. No virtual cell overlaps two real cells by construction. To obtain the Riemann preimage for the child face, we interpolate $\hat{Q}_h^*$ from the coarser cell into its virtual cells. The virtual cell then extrapolates its "inherited" STP through $id|_{\partial c}$ onto the face, where it complements the data from the real cell. Interpolation is realized from coarse to fine cell resolutions, prior to the respective Riemann solves. The concept of using virtual cells to remove hanging faces temporarily, i.e., from a solver's point of view, is well established in the (block-) structured world, where it is also known under the term ghost cells [5, 29] or halo layer [37].

The Riemann outcome along the interface of a coarse cell and a fine cell affects the coarse cell's corrector. Here, we switch from a compute-Riemann to an accumulate-Riemann approach. Let a face along a resolution transition be a parent face; it belongs to the coarser fine grid level. Let the term child face refer to a segment of this parent face which coincides with the face of one adjacent finer cell. Where a face is parent to other faces, no Riemann solve is applied to the parent, i.e., the coarse face. Instead, Riemann solves along resolution transitions are always computed along their finest resolution. Every time we determine the Riemann outcome along a child face, we accumulate it back into the parent face. This is a restriction (the transpose of the interpolation). For it, we traverse the adaptive mesh starting from the finest cells.

Adaptive meshes require us to introduce two additional tasks to (4): Interpolation and restriction. The volumetric interpolation of $\hat{Q}_h^*$ can be the subject of optimization—only the projections onto the virtual cells' faces are required. The important observation, however, is that the interpolation is a pure preprocessing step to the Riemann solves that squeeze in-between C_{STP} and F_{R}. Interpolation incorporates resolution logic, but it does not impose additional partial order constraints on either the STP or the Riemann solves. An analogous observation is to be made for the restriction of the Riemann solves' outcome.

Dynamic adaptivity adds further tasks. We focus on feature-based refinement criteria and assume that codes decide cellwisely whether to refine or coarsen. They read the solution $Q_h(\cdot, T + \Delta T)$ and study the solution's character. They thus introduce an epilogue to C_{Corr}. If the criterion triggers a refinement, new cells are created, i.e., volumetric data are interpolated, and the algorithm updates the virtual cells. This has to complete prior to any STP task in the affected part of the computational domain. If the coarsen criterion identifies a cluster of cells which can be coarsened into one bigger cell, we trigger an analogous workflow. Since the analysis of the cell-wise refinement/coarsening criterion is strictly elementwise, all tasks thus can run in parallel. The update of the virtual cells and the merger of small cells into a bigger one induce causal dependencies which however are localized and simple to integrate into the task graph.

2.4. Computational character. As both correction and Riemann tasks in (4) rely on the same PDE terms—through the partial integration on (2) the lion's share of the compute load results from F evaluations in (1)—their abstract arithmetic intensity [47], i.e., their sole computations-per-double ratio, is comparable. This statement holds if the employed Riemann solver—we use Rusanov here—only requires F and a few additional data such as an estimate of the biggest eigenvalue.

C_{STP}, however, integrates over polynomials in space and time. They are typically stored in small continuous array blocks per cell. If the STP is an iterative solve, this solve plus the time integration lead to a high intensity relative to the caches [22]: The

arithmetic intensity given as ratio of operations to loads from the main memory into the registers or a reasonably close cache is expected to be relatively high. If we study a linear variant of (1), we integrate the cell with the Cauchy–Kowalesvki procedure [14]. Here, the STP is significantly cheaper, though it still yields localized data access [8]. The time integration following the STP allows us to reuse the outcome data structure for all intermediate-in-time results.

In contrast, the Riemann solves are cheap and explicit. With the volumetric terms in (1), i.e., sources and point terms, disappearing, a Riemann task loads in the predicted solution and writes back its result to the respective face. The corrector finally is of similar simplicity, as it takes two input data streams and yields $Q_h(T + \Delta T)$. The remaining volumetric integration is simplistic. Also interpolation and restriction tasks are neither sophisticated nor computationally demanding as they are based upon the polynomials.

Finally, all refinement criteria we encounter in our code base are conceptually simple. We study the first or second order derivatives of the solution and make our refinement or coarsening decision from there. With an explicit polynomial representation of Q_h being available, these tasks therefore are of low computational intensity. They however have to read the whole volumetric data, and the subsequent refinement or coarsening might induce further memory accesses and allocations.

ASSUMPTION 2. *We assume that the STP, i.e., the volumetric task, is computationally heavier than all other tasks. In particular, the face task (the Riemann solve) tends to be memory bandwidth- and latency-bound. It brings together data from adjacent cells scattered in memory but does not yield a high number of floating point operations (flops).*

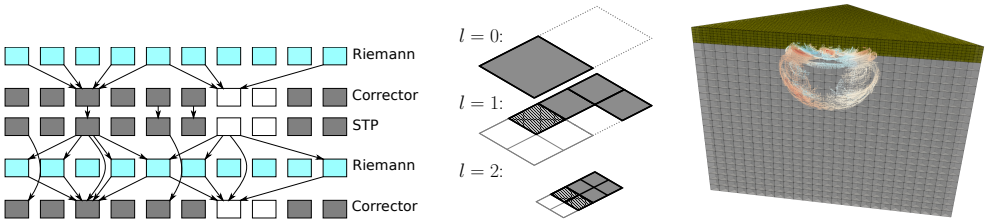

FIG. 1. *Left: Schematic illustration of task graph with some dependencies for two-dimensional setup: Cells within a regular grid region (filled) combine the local DG solution with the result of $2d$ Riemann solves. Cells adjacent to grid refinement (empty) require input from more or fewer Riemann solves if they run the corrector. Middle: Schematic illustration of various spacetree (quadtree) nodes when a tree is used to host an enclave/skeleton mesh. Dark tree nodes are fine grid nodes. They can be refined ($\ell = 0$) or unrefined ($\ell \in \{1, 2\}$). Nodes with hatching are (unrefined) virtual nodes. White nodes are supplemental. They complete the tree, but they do not carry data. Right: Sketch of the LOH.1 benchmark: It is a simplified earthquake setup, where the cubic domain consists of two types of material (layers) and a point source stimulus induces the elastic wave propagation.*

2.5. Task graph structure. ADER-DG's task graphs are conceptually simple (Figure 1): There are cell tasks, face tasks, and AMR tasks. For simplicity, we omit an explicit discussion of tasks tied to dynamic and static adaptivity for the remainder of this section where appropriate. We note the following: (i) The STP decomposes into one task per cell. (ii) The individual C_{STP} tasks are independent of each other. Operator F_{R} decomposes into one task per grid face. (iii) Finally, the individual F_{R} tasks are independent of each other. While each F_{R} task requires input from the C_{STP} tasks from its adjacent cells, each C_{Corr} task requires input from the $2d$ F_{R} tasks of

the adjacent faces.

Our task types translate into two grid traversal types: One over cells, and one over faces. Per type, all task evaluations are independent of each other. Task assembly-free processing thus is possible if we run over the grid three times. A first traversal issues all C_{STP} tasks and eventually waits for them to complete before a second traversal issues all F_{R} tasks. A final sweep corrects the solution and thus yields the subsequent time step's solution. Such an assembly-free approach describes a producer-consumer pattern: One or a few main threads traverse the grid and produce tasks; all other threads consume these.

We assume that our grid changes frequently. A naive realization with grid sweeps exhibits disadvantageous properties:

1. We employ a nonoverlapping domain decomposition and solve the Riemann problems redundantly on both adjacent ranks. An STP adjacent to an MPI domain boundary thus has to send its face data over to neighboring ranks, such that all ranks can run their F_{R} tasks autonomously. Data exchange in MPI has to be deterministic. We may not simply spawn C_{STP} tasks and make them send their outcome.
2. Modern multicore chips are equipped with memory controllers that cannot keep all cores busy. Algorithms have to avoid that all tasks access the main memory controllers concurrently and thus become bandwidth-bound [31, 47]. Fire-and-forget of F_{R} tasks by the sweeps however yields a large set of memory-sensitive tasks in one rush.
3. Cache-efficient codes perform as many operations as possible on data before these are moved out into the main memory again. With one sweep per phase, we have to assume that the outcome of a Riemann solve does not reside inside the cache long enough for the next corrector. A similar consideration holds for the outcomes of the STP.
4. Adaptive grids require us to project the solution along resolution transitions onto the finest grid, to solve the formulation there, and finally to restrict the Riemann solve's outcome again [4]. AMR injects dependencies into the Riemann solve phase.
5. For high flop/s rates, it is important that no phase of the solve exhibits low concurrency, has high bandwidth demands, or synchronizes the other tasks. Mesh cells that dynamically refine run the risk of doing so: If they are processed late throughout the sweep, they allocate memory, initialize data structures, and then invoke the actual computations, while the other threads might already have run out of tasks.

3. Enclave tasking. Our solution to the aforementioned challenges is a technique we label as *enclave tasking.* It relies on our topological assumption on the DG grid plus one assumption on typical refinement patterns.

ASSUMPTION 3. *We assume that mesh refinement criteria typically refine and coarsen the mesh along resolution transitions: A cell belonging to grid level ℓ might be refined if at least one adjacent cell has a level $\hat{\ell} > \ell$, i.e., is finer. A cell belonging to a grid level ℓ might be coarsened if at least one adjacent cell has a level $\hat{\ell} < \ell$, i.e., is coarser. Cells surrounded by cells of the same grid level are typically neither refined nor coarsened.*

Explicit time stepping schemes for hyperbolic equation systems render our assumptions on refinement and coarsening reasonable as the CFL condition ensures

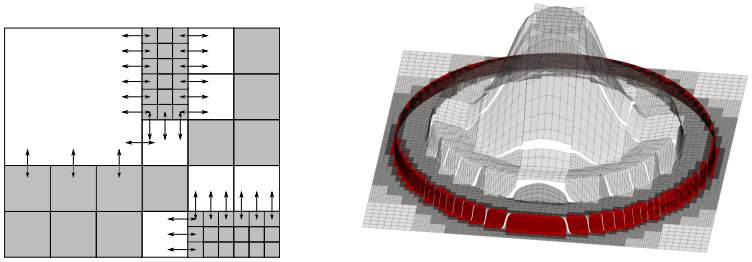

FIG. 2. *Left: An adaptive Cartesian mesh where the Riemann solves along adaptivity boundaries are denoted by arrows. Cells adjacent to cells of a finer resolution describe a skeleton while the filled cells form enclaves. Right: Simulation snapshot of an Euler simulation. The code uses patch-based FV along the shock (colored areas) and ADER-DG with order $p = 7$ everywhere else. The gaps in the visualization are a direct result of the discontinuous shape functions.*

that information does not propagate more than one cell at a time. The assumption does not hold globally for strongly nonlinear equations where areas of interest for a refinement criterion can "pop up" as shocks develop out of smooth solutions. It furthermore breaks down for setups with time-dependent boundary conditions or source terms that stimulate a wave throughout the simulation. Finally, it does not anticipate that wave spreading can yield large regularly refined regions which eventually should make the mesh thin out, i.e., coarsen over a whole subdomain.

Yet, it seems that this happens rarely or locally. In the following, we do not make any semantic modifications to the cell treatment. We only optimize using the assumption. Whenever and where it does not hold, our code does not benefit from the optimizations as proposed.

Let a *skeleton grid* of a given adaptive mesh comprise those mesh cells that are either adjacent to a domain decomposition boundary or are adjacent to at least one cell of a finer level. The remaining cells form *cell enclaves* (Figure 2).

3.1. Algorithmic blueprint. Enclave tasking maps each ADER-DG time step onto two types of mesh traversals. We refer to them as *primary mesh traversal* and *secondary mesh traversal*. They take turns. Furthermore, enclave tasking assigns each cell in the mesh a boolean marker $STP_{complete} \in \{\bot, \top\}$. At construction $STP_{complete}(c) = \top \; \forall c \in \Omega_h$.

Primary mesh traversal. The primary mesh traversal runs through the mesh. It satisfies the following properties:

- P.1 All primary mesh traversals on the parallel computer are deterministically reading the faces along MPI boundaries.
- P.2 A primary mesh traversal reads all $2d$ adjacent faces to any cell before it reads the cell itself.
- P.3 A virtual cell is read before the spatially overlapping real cell is read.

The primary mesh traversal for ADER-DG triggers the following steps:

1. Whenever the traversal reads a face for the first time that is adjacent to an MPI boundary, we receive Riemann solver input data from the neighboring rank. As a result, all data feeding into a Riemann solve is available locally.
2. For each face read, we check that $STP_{complete} = \top$ for all adjacent real cells of the same resolution that are held on the same rank. If one flag is not set, i.e., equals $\bot$, the traversal is suspended. We yield. The runtime thus gets the opportunity to process other tasks. Upon return, we recheck the condition.

3. For each face that is not subdivided further, the traversal computes F_{R}.
4. For each child face, the Riemann result is immediately restricted.
5. For each cell c that we read,
   (a) we run the corrector,
   (b) we evaluate the dynamic adaptivity plus limiter criteria [15],
   (c) we reset the completion flag $STP_{complete}(c) \leftarrow \bot$, and
   (d) we spawn a new STP task C_{STP} if the subsequent time step size is known already [9].

The traversal studies the cell's adjacent faces upon its load. If the cell is adjacent to a resolution transition or adjacent to the MPI domain boundary, the cell is a skeleton cell. Otherwise, it is an enclave cell. For enclaves, the spawned C_{STP} task goes to a standard task queue. We prioritize this queue lower than the actual tree traversal and label it a background queue. Conversely, the task goes into a high priority queue for skeletons.

The primary mesh traversal is a task producer that supplies the task runtime with ready tasks. The traversal itself can run in parallel. Semaphores on the faces—which we did not discuss explicitly—ensure that no race conditions arise from Step 4. If we know all admissible time step sizes and hence can implement Step (5d), enclave tasking logically shifts compute steps, i.e., brings some tasks forward [9]: The primary traversal runs the Riemann solve plus the two subsequent volumetric tasks. The STP among them logically belongs in the next time step. If this is not possible, we have to run through the grid once more after each primary sweep and issue the follow-up STPs. This is an "unproblematic" activity from a performance point of view, as STPs are arithmetically intense. All C_{STP} tasks are straightforward realizations of C_{STP} from (4). Upon a task's termination, it sets its marker $STP_{complete}(c) \leftarrow \top$.

Secondary mesh traversal. The secondary mesh traversal runs through the mesh. It satisfies the following properties:

S.1 All secondary mesh traversals on the parallel computer are deterministically reading the faces along MPI boundaries. If a face separates domain Ω_1 from Ω_2, both ranks r_1 and r_2 owning the respective domains hold a copy.
S.2 A secondary mesh traversal reads a cell before it reads/studies any of its $2d$ adjacent faces.
S.3 A virtual cell is loaded after the spatially overlapping real cell has been read.

The secondary mesh traversal for ADER-DG triggers the following steps:

1. For each cell read that belongs to the skeleton, we check whether $STP_{complete} = \top$. If not, we yield before we check again.
2. For each virtual cell that is loaded, we interpolate from its parent and we project the interpolated data onto the virtual cell's faces.
3. If a face coincides with the MPI domain boundary, we send the Q_h^* projection from the local adjacent cell to the neighboring rank.

The secondary grid sweep is a degenerated grid traversal traversing only the skeleton.

3.2. Relation to trees and forests as well as space-filling curves. Tree discretizations and traversals fit seamlessly to enclave tasking and its traversal. Any coarse to fine traversal [45] allows us to realize it. If the traversal is realized through a (depth-first) push-back automaton, i.e., a recursive function, we embed all routines from the secondary traversal into the recursive function's preamble before we recurse further (pre-order), while we realize the primary traversal's steps in the post-order, i.e., when we backtrack. For breadth-first, enclave tasking's primary traversal runs through the grids from fine to coarse. The secondary traversal starts with the coarsest

resolution.

In a tree world, it is convenient to make the spacetree accommodate both real cells and virtual cells. For this, the nodes of the tree are classified as follows: Inner nodes are refined tree nodes which do not carry an ADER-DG discretization but parent further inner or fine grid nodes. An unrefined fine grid node carries a polynomial from Q_h but is not refined further. This is a node that spans a cell of the ADER-DG mesh. A refined find grid node carries a polynomial, too, but is refined. It parents virtual or supplemental nodes but hosts a cell of the ADER-DG mesh, too, even though it is refined. A virtual node is unrefined and supports our AMR implementation. It does not carry a real solution but temporarily is subject to $\hat{Q}_h^*$ writes. Supplemental nodes can be refined—along their descendant and then are solely supplemental or virtual nodes—or unrefined. Their purpose is to complete the tree language (Figure 1).

In our nonoverlapping domain decomposition, ranks compute the Riemann solutions redundantly. If a face separates domain Ω_1 from domain Ω_2, both rank r_1 and r_2 owning the respective domains have a copy of this face. Our scheme assumes that the secondary traversal sends out data. These data thus become available in the subsequent primary sweep on the destination rank. To avoid resorting boundary data, it is convenient to make both r_1 and r_2 traverse their shared faces in the same order or in reverse order after each sweep [45]. In these cases, queues or stacks can be used for all boundary data exchange.

3.3. Properties. With enclave tasking, the individual ADER-DG steps are not synchronized among different cells: Some might still "wait" for their Riemann solves and correctors

$$(id|_{T+\Delta T} \circ C_{\mathrm{STP}} \cdot Q_h(T), id|_{\partial c} \circ C_{\mathrm{STP}}\, Q_h(T))^T,$$

while others have already issued $C_{\mathrm{STP}}\, Q_h(T + \Delta T)$. The producer-consumer pattern of our traversal ejects ready tasks which can be run immediately once cores become available. The task markers resolve task dependencies. This is the reason we can work completely task graph assembly free.

A high prioritization of the skeleton STPs implies that tasks that yield MPI messages are processed early. The secondary mesh traversal then can issue MPI sends, while many remaining enclave STPs still linger in the ready queue. We thus give MPI the opportunity to overlap computation and communication and reduce the risk of a late sender pattern [30].

Expensive inter-resolution transfer operators (restriction and prolongation) are either explicitly hidden behind enclave STPs, too, or intermixed into the primary mesh traversal where we may assume that many STP spawns prelude the first interpolations. We thus hide their memory-intense operations behind computations. Also bandwidth-demanding Riemann solves and correctors mix with computationally heavy STPs.

Neither the MPI-oblivious behavior nor the orchestration of tasks with different characteristics are constrained by dynamic AMR. Grids can change in each and every primary sweep. All skeleton markers are computed on-the-fly. As mesh refinement is only one substep of the primary traversal, and as the primary traversal both produces new STP tasks and does not wait for all STP tasks from the previous time step to complete before it kicks off, it is fair to assume that expensive memory allocations, which furthermore typically struggle to run in parallel, hide behind computations of further STPs.

The advantageous AMR-agnostic characteristics of enclave tasking require two assumptions to hold: On the one hand, the runtime has to get the prioritization right.

If the secondary mesh traversal waits for STPs too long and too many noncritical (enclave) tasks are processed instead, then we will run into close-to-serial phases in the subsequent primary sweep. On the other hand, the individual STPs have to be expensive relative to other algorithmic steps as well as the task production.

4. Tailoring the task runtime system. Modern task systems are designed to handle millions of small tasks with dependencies. For enclave tasking, the latter feature is not required. On the contrary, constructing a dependency graph on-the-fly—the grid might change every time step—would induce algorithmic overhead/latency that postpones the processing of the first task. The spawned STP tasks are ready by construction. If the grid traversal is deterministic and, besides the AMR grid alterations, always the same, first-in, first-out (FIFO) task processing delivers an optimal task execution order as long as all skeleton tasks are run prior to the enclaves.

Algorithm 2 Blueprint of the consumer task. It accepts the queue q filled with STPs. These are logical tasks not queued into the actual runtime, while the consumer task itself is a real task in a multithreading/-tasking sense.

```
1:  function runConsumerTask(task queue q)
2:      C ← fetch and decrement(#consumer tasks)
3:      if C < size(q)/N_min then
4:          #consumer tasks ← fetch and increment(#consumer tasks)
5:          spawn new consumer task(q)
6:          reenqueue ← ⊤
7:      else if (C ≤ 1) ∨ not empty(q) then
8:          reenqueue ← ⊤
9:      else
10:         reenqueue ← ⊥
11:     end if
12:     process up to N_max tasks(q)
13:     if reenqueue then
14:         #consumer tasks ← fetch and increment(#consumer tasks)
15:         spawn new consumer task(q)   ▷ requeue/respawn of the present task
16:     else
17:         terminateTask                ▷ starve one consumer
18:     end if
19: end function
```

Our realization wraps around Intel's Threading Building Blocks (TBB) [34]. We modify this tasking runtime to accommodate our needs. Three variants are available. Our basic variant maps both enclave and skeleton tasks onto native TBB tasks. The second variant puts the logical tasks into a queue. It then spawns a number of *consumer tasks* which dequeue these (logical) tasks and process them. In our realization, consumer tasks are the real tasks in the TBB sense. Enclave and skeleton tasks are the work items processed by these consumers (Algorithm 2). A third variant switches from a plain FIFO queue to a priority queue as provided by the TBB library. The high priority/low priority concept of skeletons versus enclaves is realized through priorities (integers) attached to the enqueued items.

4.1. Task prioritization and orchestration. The efficiency of the task runtime for enclave's producer-consumer pattern depends on the balancing of task production and task processing. If the traversal fails to produce enough tasks to keep

other cores busy, performance decreases. If the processing of heavy STP tasks constrains the traversal, it runs the risk of decreasing the performance, too, as the system might run out of ready tasks later down the line.

Our code thus is parameterized through an $N_{\min}$. Unless we wait for a termination flag of an STP to be set, the grid traversal issues at most one consumer task. Consumer tasks in turn fork into more consumer tasks if the ready queue is reasonably big, i.e., if each consumer will have at least $N_{\min}$ work items. If the ready queue starts to empty, the consumers starve. Besides ensuring that there are always enough cores for the tree traversal, such an approach also spawns new consumer tasks in a binary tree fashion. The task creation is done by the master thread only for the first consumer. The counterpart parameter $N_{\max}$ ensures that no consumer grabs too many tasks in a row before it re-evaluates the starvation/forking criteria again.

Algorithm 3 The subroutine used to wait for a predictor task to finish.

```
1: function waitForSTP(x)
2:     while x not set do
3:         if MPI_Probe(any message) then                    ▷ MPI progression
4:             MPI_Recv                                ▷ Only one/few messages at a time
5:         end if
6:         process up to N_MAX tasks(task queue q)
7:     end while
8: end function
```

4.2. MPI progression and MPI buffer layout. Overlapping communication and computations is important to ensure scalability. MPI provides nonblocking routines to this end. Our secondary grid sweep can trigger nonblocking sends. The symmetry of the communication—every send out of a face is matched by a receive—implies that the communication scheme is conceptually simple. MPI implementations nevertheless struggle to make the data transfer, the *data progression*, run in the background [21, 39], and instead require the user code to poll the MPI subsystem regularly. This gives MPI a hook in point to manage the actual data transfer. There are two solutions to realize this polling: Either a dedicated progress thread (PT) is deployed, or the user code itself calls MPI routines. The latter has to compromise between frequent calls and call overhead, while most applications do not want to sacrifice a whole thread for MPI progression only. With enclave tasking, we can plug into the $STP_{complete} = \top$ checks to facilitate MPI progression. Our main task acts as progression tasks when it runs into a semaphore (Algorithm 3). Besides the MPI progression, it also processes some tasks, i.e., helps out on the consumer side. It inherently overlaps data exchange and computations.

Many MPI codes aggregate MPI data in dedicated buffers before they send them out. Each send induces some overhead. Message aggregation reduces this overhead. With enclaves, we however benefit from small messages: Enclave partitioning ensures that partition domain boundary data are sent out early compared to work done in the interior. Throughout waits for STP results, we receive Riemann input data chunk by chunk. Exchanging small chunks of nonaggregated per-face data ensures that no single receive of a very large message delays the simulation progress. As we ensure that all MPI data are received in the right order [6, 45], we avoid both resorting overhead and too many unexpected messages.

5. Generalization of enclave tasking. ADER-DG is a peculiar explicit time stepping scheme. We use it as a showcase for enclave tasking. However, the enclave concept applies to a variety of DG approaches for a variety of problems.

5.1. Explicit Runge–Kutta DG schemes. For traditional explicit DG schemes including Runge–Kutta, the weak formulation of (1) yields operations where the volumetric integrals (tasks) do not feed into the Riemann solves—nor does the opposite happen. We however have to bring together the Riemann and volume integrals outcomes to construct a subsequent time step $Q_h(\cdot, T + \Delta T) = (F_{\mathrm{R}} + C_{DG})\, Q_h(\cdot, T)$ or an intermediate step in the Runge–Kutta tableau. We therefore propose making the Riemann solves feed logically into the volumetric kernels: A volumetric kernel computes the weak formulation over the cell, but it also accepts the outcome of the $2d$ adjacent Riemann solves and immediately merges them into the result.

In such a setup, enclave tasking requires two grid sweeps: A primary traversal computes all Riemann problems. It also issues all cell tasks, bringing the ingredients together. As a cell is read after its $2d$ adjacent faces have been read, all cell tasks are by definition ready. The secondary traversal degenerates. It does not compute anything anymore, but projects all updated solutions onto the faces immediately such that they are sent out to adjacent ranks and available there in the next primary sweep.

5.2. Finite volumes. FVs for explicit time stepping schemes choose piecewise constant shape functions for Q_h. They thus can be read as an ADER-DG scheme with a degenerated STP. Solely the Riemann outcome determines a subsequent time step $Q_h(\cdot, T + \Delta T) = (F_{\mathrm{R}} + id)\, Q_h(\cdot, T)$. Enclave tasking relies on reasonably expensive STPs such that grid traversal and Riemann solves disappear behind all the volumetric tasks. Straightforward FVs are a bad fit to enclave tasking.

Yet, all depends on the realization of the Riemann solve. Many advanced FV solvers rely on a sophisticated reconstruction of the solution that they feed into the Riemann solver. If such a construction decomposes into a "left" contribution and a "right" contribution that we can compute independently of each other, these reconstruction steps can be outsourced as volumetric kernels: Each cell task computes the respective reconstructions for $2d$ Riemann solves on its adjacent faces. If such outsourcing is possible, enclave tasking can help: A primary sweep triggers all reconstructions, a secondary sweep ensures that reconstructed data is sent over the network and AMR is handled properly, and the subsequent final primary sweep—which we might combine with the next time step [9]—then issues the actual Riemann solve.

5.3. Block-structured methods. It is this discussion of volumetric cost versus face tasks that implies that block-structured AMR [12] and enclave tasking fit together. In block-structured AMR, blocks or patches—typically regular Cartesian grids—are embedded into the cells. They communicate with their neighbors through halo layers. In such a scheme, the Riemann task F_{R} becomes a halo layer exchange task and we end up with the situation described before where the face tasks feed into the volumetric updates. As long as the halo updates are cheap compared to the patch updates, enclave tasking is of value.

We use block-structured FV in our own ADER-DG code as a limiter [15]: Our code determines the solution update through ADER-DG. If the resulting solution is physically wrong—if it yields negative densities, e.g.—or if the solution exhibits oscillations, we roll back ADER-DG on the respective cell and replace the time step for this particular cell with an FV scheme. To match the ADER-DG time step for

a polynomial ansatz p with the FV time steps, the patches per cell have dimension $(2\,p+1)^d$. Though this solver hybrid validates the claim that (block-structured) FV benefits from enclave tasks, another pro-enclave argument has to be read with care: Enclave tasking for DG ensures that the bandwidth- and latency-sensitive face tasks dribble through the system and that the runtime orchestrates compute-heavy volumetric tasks around them. In an FV world, patch updates tend to be bandwidth-bound. The orchestration argument collapses for a pure FV approach. It continues to hold for ADER-DG where the FV is an a posteriori limiter. Here, we may assume that only a few cells from the domain are limited, i.e., classic ADER-DG cells remain and their volumetric kernels now mix with Riemann solves, traversal tasks, and the patch-based FV updates.

5.4. Implicit schemes and linear equation system solves. Iterative linear equation system solves for DG as they arise for elliptic problems and implicit time stepping schemes typically rely on matrix-vector products over (1). They thus resemble the situation of Runge–Kutta schemes from section 5.1. Enclave tasking thus can be of value if we work matrix-free [44]. Particularly appealing is the combination with multiplicative *hp*-multigrid. Here, the fine grid smoothing, and, hence, residual computation, is the dominant step. If we run multiple smoothing steps in a row which are followed by a final residual computation that feeds into a restriction, enclave tasking unfolds its full potential to hide the Riemann solves behind the volumetric kernels.

For many elliptic (sub)problems, codes start with initial meshes that resolve sources of interesting behavior—typically material transitions or complex boundaries—accurately right from the start. The areas of interest are known, and the code develops the AMR mesh from there. Errors from "problematic" regions decay from there according to their fundamental solution, i.e., the finer grids of an appropriate mesh follow this decay. They spread from the problematic region. Refinement criteria can be throttled to refine at most one additional layer around a given region of interest per iterate. This makes a dynamic refinement pattern fit to our AMR assumption.

5.5. 2:1 balancing and *k*-partitioning. Many codes or numerical implementations require 2:1 balancing [41], while many refinement criteria yield reasonably balanced grids automatically. Our definition of enclaves and skeletons does not rely on a 2:1 balancing property. It is agnostic of resolution balancing.

If grids are balanced plus feature large resolution transitions, we observe a higher skeleton-to-enclave cell ratio than for a mesh with the same resolution difference yet no balancing at all. In balanced meshes, refined regions spread out gradually—they "ripple" through the domain—where other codes would feature massive resolution jumps over $\Delta\ell$ grid levels. Where the latter exhibit one fine–coarse transition manifold, a balanced grid features $\Delta\ell$ of these transitions hosting skeleton cells. Bipartitioning amplifies this skeleton impact. With bipartitioning, whole transition regions can become skeletons. For grids featuring $k \geq 3$-subdivision [46, 45], transition skeletons are always interrupted by enclaves.

We hence may assume that bipartitioning and balancing diminish the performance gain through enclave tasking. Unbalanced grids with $k \geq 3$-subdivision benefit more from it. The analysis and validation of this hypothesis is however out of scope here.

6. Experimental results. We benchmark our algorithm and our code on SuperMUC-NG at the Leibniz Supercomputing Centre (LRZ). Its nodes are two-socket systems, i.e., each node hosts two 24-core Intel Xeon 8174 (Skylake) CPUs. They have been clocked at 2.3 GHz and are connected through Intel Omni-Path. We have

96 GB main memory available on each node.

Enclave tasking is a generic concept within the ADER-DG mindset but applies to FV as a special case of DG schemes, too. We integrated our ideas into the ExaHyPE [1, 33] engine where they support ADER-DG for orders $p \in [3, \dots, 9]$ but also a patch-based FV scheme which is used by ExaHyPE[1] for a posteriori limiting [15]. All results are thus obtained with applications built upon ExaHyPE.

We study the enclave impact for two applications with different character. One application is a seismic wave code that solves the LOH.1 benchmark. The cuboid domain used in this benchmark consists of two material layers. Wave propagation is initiated by a point source that is placed just below the upper layer. Due to the material transition, an interesting wave pattern emerges (Figure 1).

This well-known benchmark is governed by a linear variant of (1). However, we translate it into a nonlinear variant where the material enters the equation as an additional scalar PDE over $\alpha(t)$ following the trivial rule $\partial_t \alpha = 0$. It does not move. Such an immersed boundary approach allows us to handle a material transition which is not grid aligned [42]: Wherever the code encounters a transition, we cover it with an FV patch. Some distance away from the boundary, these FV patches are coupled with ADER-DG cells. In the majority of the domain, we thus use ADER-DG. Though phrased overall as nonlinear PDE—the α term injects this nonlinearity—the code's high-order ADER-DG degenerates to a linear case within the majority of the domain. The STP thus is directly solved through Cauchy–Kowalevski. Only on the FV patches along the material transition, we solve the original nonlinear PDE. The material plus the nonlinear PDE require us to store 13 doubles per degree of freedom.

Our second benchmark is prescribed by the compressible Euler equations [28]. Compared to the seismic setups, this setup has only five unknowns (a scalar material density and a scalar energy which closes the system plus the vector of velocities). However, it is a nonlinear variant of (1), which generally does not degenerate to a linear case and thus requires an expensive nonlinear STP where we do not know the number of internal Picard iterations a priori. This solver switches to FV as an a posteriori limiter if shocks are encountered: the solver uses ADER-DG with high order in the majority of the domain but employs a patch-based FV scheme along discontinuities [15]. Unlike in our immersed boundary setup, the FV regions travel this time. We always refine the DG solution down to the FV level close to the shocks and then glue ADER-DG and FV together volumetrically. This implies that no adaptivity cuts through the FV subdomains, and FV cells by definition thus are enclave cells.

We ran all experiments in 3D. The timings are given per time step, i.e., we freeze particular adaptivity patterns throughout the measurements. Efficient variants of tree modifications including fast balancing—if required—are known [23, 41] and suggest that the total time-to-solution character of a simulation does not change dramatically. Alternatively, any remeshing as well as propagation of the limiter within the mesh can be integrated into the actual time stepping along the lines of [9, 44]. With dedicated remeshing phases, enclave tasking does not impact these phases' runtime, while Assumption 3 holds in a hard sense, i.e., we can omit the phrases *typically*: Throughout the time steps, the skeleton labels never change. We can determine the skeleton versus enclave classification once throughout the mesh adaption and then keep them until the mesh changes again. With a merger of time stepping and AMR, enclave tasking might deteriorate locally, and we have to update the markers on-the-

[1] www.exahype.org [1].

TABLE 1
Serial runtime in seconds per ADER-DG solution degree of freedom per time step per task type for polynomial orders $p \in \{3, 5, 7\}$.

	Seismic		Euler	
	ADER-DG	FV	ADER-DG	FV
$p = 3$, C_{STP}	$1.22 \cdot 10^{-6}$	–	$2.72 \cdot 10^{-7}$	–
$p = 3$, F_{R}	$2.42 \cdot 10^{-8}$	–	$3.59 \cdot 10^{-9}$	–
$p = 3$, C_{Corr}	$6.98 \cdot 10^{-8}$	$6.89 \cdot 10^{-5}$	$2.74 \cdot 10^{-7}$	$2.76 \cdot 10^{-5}$
$p = 5$, C_{STP}	$1.75 \cdot 10^{-6}$	–	$3.42 \cdot 10^{-7}$	–
$p = 5$, F_{R}	$1.76 \cdot 10^{-8}$	–	$2.04 \cdot 10^{-9}$	–
$p = 5$, C_{Corr}	$8.06 \cdot 10^{-8}$	$6.48 \cdot 10^{-5}$	$1.63 \cdot 10^{-7}$	$2.70 \cdot 10^{-5}$
$p = 7$, C_{STP}	$2.79 \cdot 10^{-6}$	–	$6.07 \cdot 10^{-7}$	–
$p = 7$, F_{R}	$1.20 \cdot 10^{-8}$	–	$1.29 \cdot 10^{-9}$	–
$p = 7$, C_{Corr}	$8.80 \cdot 10^{-8}$	$6.23 \cdot 10^{-5}$	$1.70 \cdot 10^{-7}$	$2.65 \cdot 10^{-5}$

fly for each and every time step.

6.1. Computational characteristics. Prior to algorithmic studies, we benchmark how much time we have to invest into one degree of freedom update per time step. This cost is broken down into the ADER-DG or FV cost (Table 1). For our hybrid codes combining ADER-DG and FV, the solver in practice will yield a mixture of the two characteristics as both codes run concurrently. We furthermore emphasize that the total runtime cost of a simulation will comprise grid management and parallelization overhead as well as adaptive meshing cost. The latter comprises the evaluation of refinement and coarsening criteria, interpolation and restriction. Refinement criteria evaluation and inter-resolution transfer operators are fused with the correction steps in our code. The remaining overheads are negligible.

We clearly see that the two ADER-DG solvers have different solver characteristics: The ratio between STP and Riemann solve is comparable, but the dynamic AMR is expensive. As we merge the latter into the STP in our code, Euler's cell tasks are significantly more expensive relative to the Riemann solves than the seismic cell tasks. In contrast, the FV patch updates are by magnitudes more expensive than all the ADER-DG cells. For all setups, the ratio of the STP cost to the remaining tasks shifts towards the STP with rising p. We conclude the data interpretation with a remark that our ADER-DG scheme is aggressively optimized towards Intel architectures, whereas the FV scheme is relatively straightforward. For many applications, it might be possible to reduce its cost per degree of freedom. This fact is beyond our scope here.

With these results, we expect the genuine nonlinear PDE (Euler) to benefit more significantly from enclave tasking than the seismological application which is effectively linear in the majority of the domain. In general, we expect the impact of enclave tasking to become more significant as we increase the polynomial order. It is clear that enclave tasking should notably become very important once we have very expensive enclave tasks. This is the case for our FV cells. It is however not clear how their nonpredictability (we do not know a priori where limiting is required) affects the runtime and performance of the scheme.

Our code employs relatively simple Riemann solvers. More expensive solvers shift the emphasis away from the STPs and thus diminish the impact of enclave tasking, unless the Riemann solve is thinned out: For many solver variants, only the reconstruction step bringing data from the adjacent cells together has to be realized within the face-associated compute kernels, while the actual Riemann solve can be

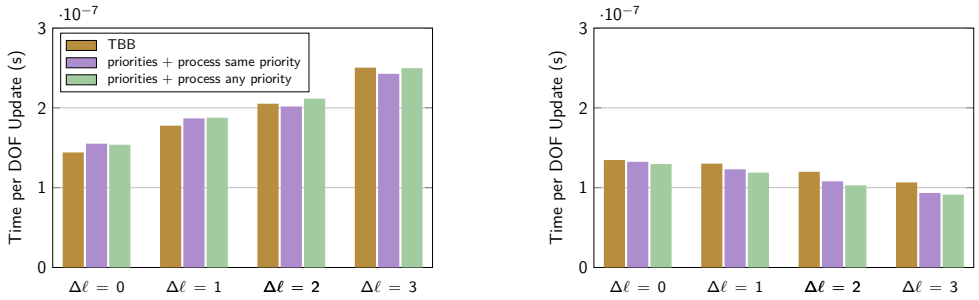

FIG. 3. *Impact of different task processing strategies for $p = 5$ (left) and $p = 9$. All strategies except TBB use $N_{min} = N_{max} = 8$. $\Delta\ell = 0$ denotes a regular grid; $\Delta\ell$ otherwise denotes the maximum number of added AMR mesh levels. One socket is used.*

outsourced to the corrector. Such an implementation breaks the logical face operation up into a face part and a volumetric computation. The latter can be merged into the corrector (see the discussion on lifting [24] in the FV context). It is an open question to which degree the arising increase of compute load—in many cases computations of the volumetric part of the Riemann solve will run redundantly on both adjacent cells of a face—is compensated by an increase of the efficiency of the enclave tasking as well as by the increase of data access locality due to the merger of Riemann operations and correction.

6.2. Impact of the task runtime parameters. A second assessment studies the behavior of our tasking system, i.e., it benchmarks the tasking runtime's tailoring against native TBB. All tests are run for the two-dimensional Euler equations simulating a circular explosion. We employ a 729×729 base grid. If we activate dynamic AMR, this setup yields rather aggressive, time-dependent mesh adoptions.

Our first test setup fixes the quantities $N_{\min} = N_{\max} = 8$, i.e., whenever a thread processes STP tasks, it tries to process eight tasks in a row. The experiments use one socket of the two-socket system to exclude NUMA phenomena. They start from a regular grid and add up to three levels of dynamic AMR. We benchmark native TBB where every STP task is spawned as a real TBB task against implementations that put these STPs into a priority queue and create TBB (consumer) tasks which grab them from there. Skeleton tasks have higher priority than enclave tasks.

Our data (Figure 3) suggest that the queue wrap-around induces a nonnegligible overhead. Plain TBB is thus faster than any modifications if the computational load per STP is sufficiently low. However, aggressive AMR or high relative STP cost imply that an anarchic spawning of native TBB tasks leads to situations where either the main thread becomes idle as it waits for an STP to finish, or the tasking system unfortunately processes the wrong enclave or tasks, or where the main traversals tasks yield, their threads pick up other enclave tasks, and eventually return too late to the actual traversal such that we face delays or work starvation later down the line. The grid traversal threads should process STPs themselves whenever they wait for an STP outcome. If they do so, they naturally prioritize skeleton tasks. If no more skeleton tasks are ready and the STPs are heavy, it is advantageous to switch to enclave tasks. This happens if all skeleton tasks are currently processed. If the STPs are not that heavy, it is however better to make a thread wait for STP outcomes, i.e., to actively poll the completion, whenever no enclave tasks are remaining. Again, actively joining the STP computations for enclaves introduces delays down the line.

Further experiments (not shown) demonstrate that a switch to three dimensions, other polynomial orders, or other applications does not alter our observations qualitatively; it is the presence of dynamic AMR and the relative cost of the STPs that determine which task processing strategy is the fastest. Yet, switching to three dimensions or the activation of a limiter increases this relative cost and thus moves the turnover points. Experiments with various $N_{\max}$ and $N_{\min}$ values support the statements on the overhead. With the best-case processing strategy from above, the two magic parameters make a difference for relatively cheap STPs—that is, low polynomial order p for Euler. Here, it is advantageous to choose reasonably big $N_{\min} \approx 8$. A value of 8 logically fuses tasks and thus reduces overhead. $N_{\max}$ plays no major role for regular meshes. Yet if we tackle a rather adaptive mesh, it is better to have an $N_{\max}$ close to $N_{\min}$ to allow the task processing to re-evaluate the queue often. For high relative STP cost, $N_{\max}$ and $N_{\min}$ seem to play no role.

Our data suggest that fine-granular prioritization is an important feature of (future) task systems and that this prioritization—different from our manual approach—should come along with low overhead. It is obvious that the impact of prioritization depends on the relative cost of tasks: It is this cost that might render nonprioritized, i.e., less sophisticated, scheduling superior. Future task systems will have to investigate a balancing of overhead versus optimality. The other interesting balancing observation is that the best-case scheduling seems to depend on the grid regularity. A homogeneous task spawn pattern asks for a different task processing than a strongly irregular pattern resulting from dynamic AMR. This effect is also worth studying from a task system's point of view. As high-order, three-dimensional setups that utilize AMR are of primary interest to us in this study, we stick to our custom-made tasking with prioritization and process-tasks-if-you-wait strategy from here on. Low-order schemes are typically only used in our setups when we strive for very aggressive AMR. We thus set $N_{\max} = N_{\min} = 8$.

6.3. Shared memory scaling. Enclave tasking can be read as on-the-fly sorting of tasks while they drop in, where the sort heuristic is guided by the grid adaptivity pattern and the parallelization. It brings time-critical tasks forward.

We benchmark our code first on a regular grid in shared memory mode (Figure 4). Our baseline is compiled without TBB. For Euler, we distinguish between smooth and nonsmooth initial conditions, and we benchmark ADER-DG and limiting ADER-DG against patch-based FV. It is only for nonsmooth initial conditions (with shocks) that ADER-DG for Euler uses FV as a limiter. In this case, the scheme becomes a hybrid of both solvers. With smooth initial conditions, no limiter is required. It remains sole ADER-DG. Whenever FV is used, our patch size is chosen as $(2\,p+1)^d$ relative to the corresponding ADER-DG scheme. This guarantees that the CFL condition yields time step sizes of matching magnitude. Different from Euler, the immersed boundary approach uses a limiter always, yet only along the immersed boundary. Its limited region stays in place, while the Euler equations move the limited regions along the shock. For all tests, a regular grid allows us to compare our enclave implementation to a straightforward implementation with parallel fors. The latter realizes ADER-DG as a sequence of three loops triggering STP, Riemann solve, and corrector.

All setups besides the one with the very low relative STP cost scale reasonably if we use parallel for loops (`pfors`). We omit two-core results for TBB's parallel for, as TBB sacrifices one hyperthread for the scheduling of the loop. This kick-off penalty plays no major role for higher core counts. One-to-two overheads do not arise for enclave tasking.

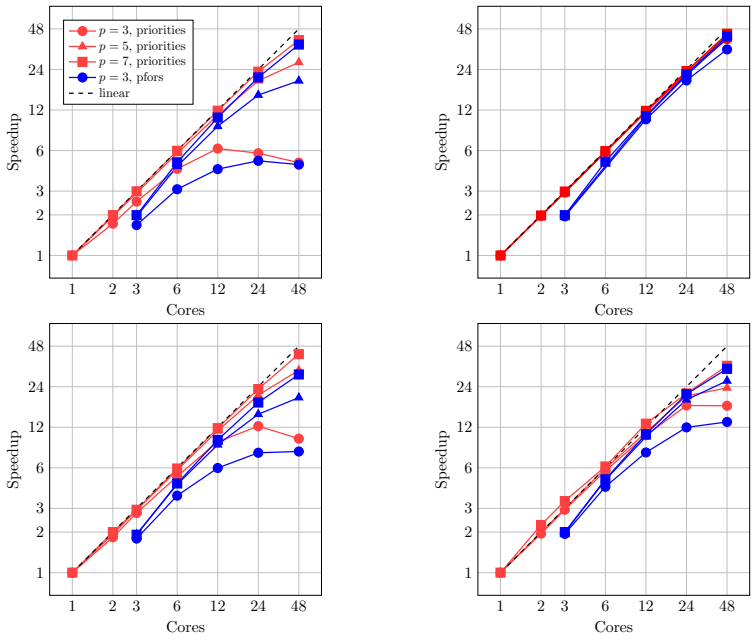

FIG. 4. *Shared memory scaling for regular mesh, i.e., without any adaptivity (strong scaling). Top row: We compare ADER-DG Euler with smooth initial conditions (left) against FV (right). Bottom row: ADER-DG with an FV limiter simulating Euler with shocks (left) against the immersed boundary method of the seismic simulation (right). All plots study three different polynomial orders p (circle, triangle, square) and benchmark classic parallel for-based parallelism against our enclave tasking with priorities (red versus blue). (Color available online.)*

Enclave tasking is robustly faster than loop-based parallelism. This difference is more significant if the limited region changes over time. If we use high orders, which implies that the limited region within the grid changes infrequently relative to the time steps, or if we use expensive cell updates (FV or high orders), both parallel for and enclave tasking play in the same league.

On a regular grid, the loop-based parallelism scales excellently for the STPs. The arithmetic intensity for the two subsequent steps however is not high. It diminishes the overall scalability. If we run high orders, the runtime of the STP becomes so dominant that the impact of these other steps disappears. For smaller orders, it is significant. With enclave tasking, this effect however is hidden behind the STPs; our approach to orchestrating the tasks such that Riemann solves and STPs overlap and the Riemann solves dribble through the system with restricted concurrency pays off.

We continue with adaptive meshes (Figure 5). The adaptivity for the immersed boundary method is static, while the adaptive pattern moves along the waves for Euler. Our adaptivity thus is twofold: there is adaptivity in space and adaptivity in the solver. Since a loop-based parallelization of dynamically adaptive meshes is not trivial, we omit comparisons to parallel loops.

The seismic setup's static adaptivity does not pose any problems to enclave tasking. In line with the regular grid tests, the $p = 3$ tests with only one adaptivity level are the only ones which fail to yield an arithmetic intensity that leads to close-to-linear speedups. Experiments with $p > 3$ plus more than one level of adaptivity are impossible due to memory restrictions, but all remaining data are more or less AMR-

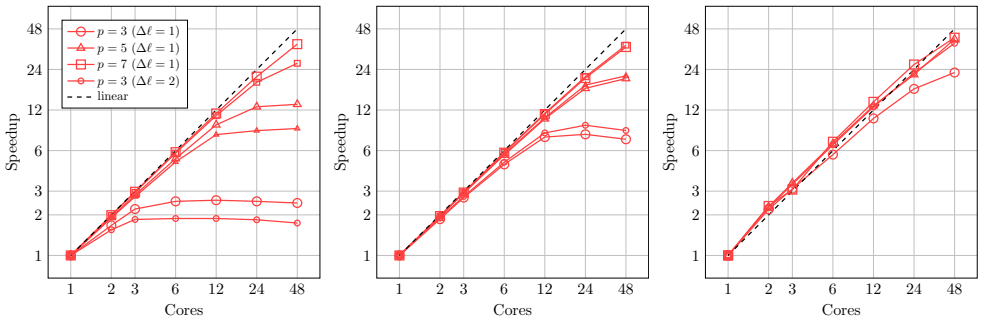

FIG. 5. *Shared memory scaling with dynamically adaptive meshes. We present data for Euler with smooth initial conditions (left) and Euler with shocks and, hence, limiting (middle) as well as immersed boundary data for the seismic setup (right). Different symbols are used for different polynomial orders p. The smaller symbol size represents a "more adaptive" mesh featuring resolution transitions of $\Delta\ell = 2$ levels. All three diagrams follow the same symbol semantics.*

agnostic. For a sole ADER-DG run, i.e., Euler without any limiter, the scalability curve exhibits classic strong scaling behavior. Low core counts yield speedups, but the performance stagnates for bigger counts. The lower the polynomial order, i.e., the cheaper the tasks, the earlier we enter the stagnation regime. Our adaptivity criterion dynamically refines towards the shock or wave gradient. This induces a critical path along the refinement fronts which consists of inter-grid transfer operators and the actual refinement criterion. If the STP tasks are heavy, we succeed in hiding all of this path. Limiting ADER-DG cures this strong order-dependence as the FV cells yield very heavy tasks.

6.4. MPI+X scaling. We close this section with MPI+X scaling tests. To obtain unbiased comparisons, we benchmark against shared-memory parallelization only as long as we stay on one compute node. Shared-memory experiments lack MPI overhead. Furthermore, we disable all dynamic load balancing, i.e., we determine a reasonable domain decomposition pattern prior to the measurements' start and stick to this splitting from thereon. The splitting uses a uniform cost model, i.e., cost per cell, which does not account for imbalances that arise when we solve nonlinear PDEs such as the Euler equations with ADER-DG or when we apply a localized limiter in the immersed boundary tests. It is a sole geometric decomposition following the Peano space-filling curve [6, 45].

The data in Table 1 highlight that a uniform cost model is a particular crude approximation for the limited ADER-DG setup, i.e., the immersed boundary case, since the limiter's FV scheme is by at least one order of magnitude more expensive than sole ADER-DG. Consequently, we may assume that the load decomposition here is unbalanced. Our results however do not suffer qualitatively from this, as we use, per experiment, a fixed number of up to 82 MPI ranks, determine a reasonably balanced static domain decomposition first, and then increase the number of cores per rank to increase the total core count. Furthermore, we only benchmark across a few time steps, i.e., the load distribution does not shift drastically. The limiter however does stress enclave tasking, as almost all runtime is spent on the FV cells. They cover only a small subset of the domain. Enclave tasking thus has only limited freedom to exploit and hide enclave work.

Both the seismic and the Euler (Figure 6) runtimes suffer from the switch from a shared memory experiment to an MPI+X run. Each individual problem size yields

a strong scaling curve, i.e., a curve that starts to stagnate or even deteriorate from a certain core count on, while we cannot really make a statement like "AMR scales less reasonably than its regular counterpart." For all solvers, the switch to a finer mesh improves the throughput. Though higher-order computations yield more science per degree of freedom, the rough cost per degree of freedom update is indistinguishable from a low-order counterpart.

The MPI+X executable suffers from overhead, such as MPI polling cost or global time step synchronization, compared to its single-node cousin. We consequently see a performance drop once we leave the single node. It is difficult to reconstruct where the other nonsmooth effects come from, but it is reasonable to assume that it stems partially from ill-balancing. This comprises not only ill-balancing as discussed above but also certain core/node counts that do not map perfectly to a given mesh. The deterioration for too-high core counts and the otherwise good scaling highlight that enclave tasking automatically hides the communication behind computation. If the amount of work on a node becomes too small, hiding fails. The break-even point is reached quickly: Cells adjacent to the MPI boundary are skeletons, and the enclave size thus tends to shrink drastically relative to the skeleton cardinality once we make the subdomains smaller. Besides standard overhead arguments—we obviously also have to invest compute resources into the mesh management and traversal—this relative growth of skeleton versus enclave size explains why the throughput improves drastically whenever we increase the mesh resolution. If we compare a regular mesh to an AMR mesh, our AMR results are qualitatively similar to the regular mesh data in most cases. This emphasizes that we hide both data exchange and AMR inter-resolution projections successfully behind the enclave work. Overall, we have successfully transferred the advantageous characteristics of enclave tasking from the shared-memory domain into the MPI+X domain.

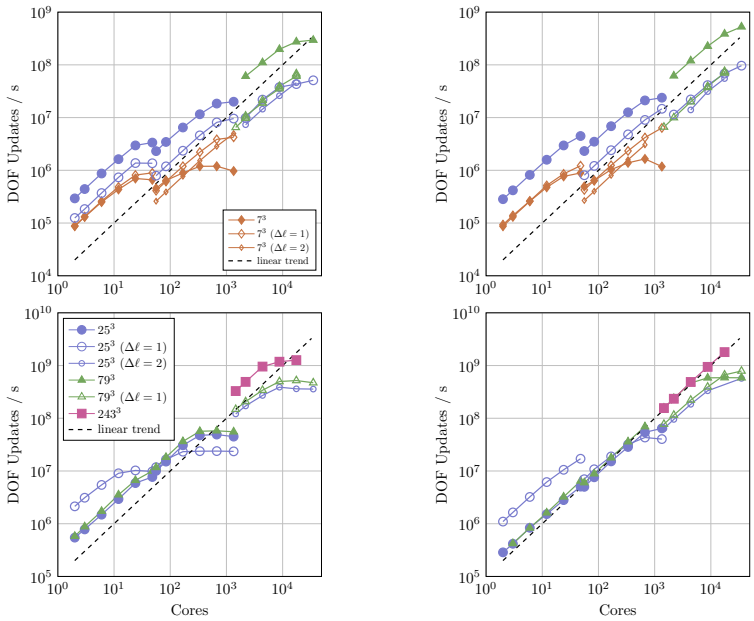

FIG. 6. *MPI+X scaling for the seismic setup (top) and the Euler ADER-DG solver without limiter. We study $p = 5$ (left) and $p = 7$ (right). Each individual connected line represents one strong scaling experiment.*

7. Conclusion. Enclave tasking is a powerful technique to equip DG codes with high concurrency and advantageous communication characteristics. It removes multicore synchronization points, overlaps computation and communication, and implicitly orchestrates a well-blended mix of compute-intense and memory-intense tasks. This makes it a powerful tool in particular in the context of dynamic AMR. Here, it provides an orthogonal technology to domain decomposition that increases the parallelism in an MPI+X environment.

There are conceptual and implementation shortcomings of the present approach. First, it performs best in cases where the adaptivity is localized and the regular subdomains host a sufficient number of cells. For extremely high-order codes which often host only a few cells per node, this might not hold. Second, our data showcases that it relies on a code where cell tasks are significantly more expensive than all other tasks. For some setups, this is only the case for large polynomial orders, for nonlinear solves within the ADER-DG cells, and for simplistic Riemann solves. Third, our "fire-and-forget" strategy for skeleton and enclave STP tasks is appropriate for explicit time stepping where the admissible time step size is well known or can be estimated. An extension to local time stepping, implicit time stepping, or elliptic problems where equation systems are solved is beyond scope here. It certainly requires further work. Fourth, our case studies do not invest in performance engineering or load balancing. Proper application of these techniques certainly will change all outcomes quantitatively.

Enclave tasking's success hinges on the task system. Our realization manually adds priorities to Intel's TBB as we found TBB's native priority scheduling insufficient. It also manually polls MPI—to allow the message exchange to make progress—and it throttles the number of background tasks. The processing of enclaves thus never grabs too many cores from the actual main traversal unless there is an enormous number of STP tasks ready. We expect the next generation of task runtimes to provide appropriate support for priorities. This flavor of our wrapper thus will become obsolete. All other features are, to the best of our knowledge, not yet on any task runtime's roadmap. Moreover, our experience suggests that it might be reasonable to equip tasks with meta-flags indicating whether they are bandwidth or compute intense. A good runtime then can ensure that the bandwidth-intense tasks dribble through the system yet that too many of these bandwidth-demanding tasks are never executed at the same time. We have had good experience with fusing our cheap Riemann solves with the traversal. Memory-expensive inter-grid transfer tasks along resolution boundaries however do not yet benefit from parallelization—we mangle them into the traversal, too—even though we have learned that they tend to align along the critical path.

There are natural follow-up steps and follow-up questions worth further investigation: First, enclave tasking is a promising candidate to be used in connection with accelerators [11, 16, 17, 19, 25, 32, 24]. Our terminology is inspired by the work of Sundar and Ghattas [40], who use enclaves to ensure that accelerators processing an enclave do not have to communicate with other accelerators directly. The enclaves are separated by skeleton cells. Our background tasks are perfect candidates to be deployed to accelerators, too. Their data transfers can be hidden behind computation, and the construction of the skeleton mesh ensures that no accelerator has to exchange DG jumps with another enclave. Second, enclave tasking alters the scaling behavior of the code base and makes it depend on the grid topology. Future work will have to study whether grid refinement criteria should anticipate this scaling behavior. It is reasonable to assume that sophisticated criteria optimize both towards an as

small as possible grid and a scaling grid topology. Their interaction with dynamic load balancing beyond the simple space-filling curve cuts employed here [6] however is not obvious. It is also not clear to which degree generic load balancing strategies can succeed or whether good strategies have to incorporate application specifics and knowledge. Finally, enclave tasking has to be studied in the context of single-sided MPI or distributed shared memory systems where much of the MPI progression pain fades away. It is a particularly fascinating idea to study the deployment of enclave tasks to remote nodes rather than only local cores. As enclave tasking adds an additional dimension of concurrency to classic domain decomposition, this idea adds an orthogonal dimension to classic load balancing. First studies along these lines are promising [35, 36].

Acknowledgments. Our development made use of the facilities of the Hamilton HPC Service of Durham University. Thanks are due to all members of the ExaHyPE consortium. All underlying software is open source [1].

REFERENCES

[1] M. Bader, M. Dumbser, A.-A. Gabriel, H. Igel, L. Rezzolla, and T. Weinzierl, *ExaHyPE—An Exascale Hyperbolic PDE Solver Engine*, 2019, http://www.exahype.org.

[2] A. Baggag, H. Atkins, C. Özturan, and D. Keyes, *Parallelization of an object-oriented unstructured aeroacoustics solver*, in Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1999, pp. 22–24.

[3] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Trans. Math. Softw., 38 (2011), 14.

[4] M. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comput. Phys., 82 (1989), pp. 64–84.

[5] M. J. Berger and R. J. LeVeque, *Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems*, SIAM J. Numer. Anal., 35 (1998), pp. 2298–2316, https://doi.org/10.1137/S0036142997315974.

[6] H.-J. Bungartz, M. Mehl, and T. Weinzierl, *A parallel adaptive Cartesian PDE solver using space–filling curves*, in Euro-Par 2006, W. E. Nagel, W. V. Walter, and W. Lehner, eds., Lecture Notes in Comput. Sci. 4128, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 1064–1074.

[7] C. Burstedde, M. Burtscher, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox, *ALPS: A framework for parallel adaptive PDE solution*, J. Phys. Conf. Ser., 180 (2009), 012009.

[8] D. E. Charrier, B. Hazelwood, E. Tutlyaeva, M. Bader, M. Dumbser, A. Kudryavtsev, A. Moskovsky, and T. Weinzierl, *Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver*, Internat. J. High Perform. Comput. Appl., 33 (2019), pp. 973–986, https://doi.org/10.1177/1094342019842645.

[9] D. E. Charrier and T. Weinzierl, *Stop Talking to Me—A Communication-Avoiding ADER-DG Realisation*, preprint, https://arxiv.org/abs/1801.08682, 2018.

[10] J. Dongarra, J. Hittinger, J. Bell, L. Chacón, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild, *Applied Mathematics Research for Exascale Computing*, DOE ASCR Exascale Mathematics Working Group, 2014, http://www.netlib.org/utk/people/JackDongarra/PAPERS/doe-exascale-math-report.pdf.

[11] S. Dosopoulos, J. D. Gardine, and J. F. Lee, *An MPI/GPU parallelization of an interior penalty discontinuous Galerkin time domain method for Maxwell's equations: MPI/GPU for IP-DGTD*, Radio Sci., 46 (2011), https://doi.org/10.1029/2011RS004689 (accessed 2018-05-21).

[12] A. Dubey, A. S. Almgren, J. B. Bell, M. Berzins, S. R. Brandt, G. Bryan, P. Colella, D. T. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. van Straalen, and K. Weide, *A survey of high level frameworks in block-structured adaptive mesh refinement packages*, J. Parallel Distributed Comput., 74 (2016), pp. 3217–3227.

[13] M. Dumbser, F. Fambri, M. Tavelli, M. Bader, and T. Weinzierl, *Efficient implementation of ADER discontinuous Galerkin schemes for a scalable hyperbolic PDE engine*, Axioms, 7 (2018), 63, https://doi.org/10.3390/axioms7030063.

[14] M. Dumbser and M. Käser, *An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes—II. The three-dimensional isotropic case*, Geophys. J. Internat., 167 (2006), pp. 319–336.

[15] M. Dumbser, O. Zanotti, R. Loubère, and S. Diot, *A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws*, J. Comput. Phys., 278 (2014), pp. 47–75.

[16] C. R. Ferreira and M. Bader, *Load balancing and patch-based parallel adaptive mesh refinement for tsunami simulation on heterogeneous platforms using Xeon Phi coprocessors*, in PASC '17: Proceedings of the Platform for Advanced Scientific Computing, ACM, New York, 2017, 12, https://doi.org/10.1145/3093172.3093237 (accessed 2018-05-21).

[17] N. Gödel, N. Nunn, T. Warburton, and M. Clemens, *Scalability of higher-order discontinuous Galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters*, IEEE Trans. Magnetics, 46 (2010), pp. 3469–3472, https://doi.org/10.1109/TMAG.2010.2046022 (accessed 2018-05-21).

[18] M. Griebel and G. Zumbusch, *Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization*, in Proceedings of Domain Decomposition Methods 10 (DD10) (Boulder, CO, 1997), Contemp. Math. 218, AMS, Providence, RI, 1998, pp. 271–278.

[19] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey, *Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers*, in SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (New Orleans, LA, 2014), IEEE, Washington, DC, 2014, pp. 3–14, https://doi.org/10.1109/SC.2014.6 (accessed 2016-03-01).

[20] F. Hindenlang, G. Gassner, C. Altmann, A. Beck, M. Staudenmaier, and C.-D. Munz, *Explicit discontinuous Galerkin methods for unsteady problems*, Comput. & Fluids, 61 (2012), pp. 86–93.

[21] T. Hoefler and A. Lumsdaine, *Message progression in parallel computing—to thread or not to thread?*, in Proceedings of the 2008 IEEE International Conference on Cluster Computing, IEEE, Washington, DC, 2008, pp. 213–222, https://doi.org/10.1109/CLUSTR.2008.4663774.

[22] A. Ilic, F. Pratas, and L. Sousa, *Cache-aware roofline model: Upgrading the loft*, IEEE Comput. Architecture Lett., 13 (2014), pp. 21–24, https://doi.org/10.1109/L-CA.2013.6.

[23] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*, SIAM J. Sci. Comput., 37 (2015), pp. C497–C531, https://doi.org/10.1137/140970963.

[24] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven, *Nodal discontinuous Galerkin methods on graphics processors*, J. Comput. Phys., 228 (2009), pp. 7863–7882.

[25] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, *High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster*, J. Comput. Phys., 229 (2010), pp. 7692–7714, https://doi.org/10.1016/j.jcp.2010.06.024 (accessed 2018-05-21).

[26] K. Kormann and M. Kronbichler, *Parallel finite element operator application: Graph partitioning and coloring*, in Proceedings of the 2011 IEEE Seventh International Conference on eScience, ESCIENCE '11, IEEE Computer Society, Washington, DC, 2011, pp. 332–339, https://doi.org/10.1109/eScience.2011.53 (accessed 2018-05-27).

[27] M. Kronbichler, K. Kormann, I. Pasichnyk, and M. Allalen, *Fast matrix-free discontinuous Galerkin kernels on modern computer architectures*, in High Performance Computing, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, eds., Lecture Notes in Comput. Sci. 10266, Springer, Cham, 2017, pp. 237–255, https://doi.org/10.1007/978-3-319-58667-0_13 (accessed 2018-05-27).

[28] R. J. LeVeque, *Finite-Volume Methods for Hyperbolic Problems*, Cambridge University Press, Cambridge, UK, 2002.

[29] R. J. LeVeque, D. L. George, and M. J. Berger, *Tsunami modelling with adaptively refined finite volume methods*, Acta Numer., 20 (2011), pp. 211–289.

[30] G. Mao, D. Böhme, M.-A. Hermanns, M. Geimer, D. Lorenz, and F. Wolf, *Catching idlers with ease: A lightweight wait-state profiler for MPI programs*, in Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14 (Kyoto, Japan, 2014), ACM, New York, 2014, pp. 103–108, https://doi.org/10.1145/2642769.2642783 (accessed 2019-07-17).

[31] J. D. McCalpin, *Memory bandwidth and machine balance in current high performance computers*, in IEEE Computer Society Technical Committee on Computer Architecture (TCCA)

Newsletter, IEEE, Washington, DC, 1995, pp. 19–25.
[32] D. Mu, P. Chen, and L. Wang, *Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using multiple GPUs with CUDA and MPI*, Earthquake Sci., 26 (2013), pp. 377–393, https://doi.org/10.1007/s11589-013-0047-7 (accessed 2018-05-21).
[33] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl, *ExaHyPE: An Engine for Parallel Dynamically Adaptive Simulations of Wave Problems*, preprint, https://arxiv.org/abs/1905.07987, 2020 (accessed 2019-05-22).
[34] J. Reinders, *Intel Threading Building Blocks*, O'Reilly & Associates, Sebastopol, CA, 2007.
[35] P. Samfass, T. Weinzierl, D. E. Charrier, and M. Bader, *Tasks unlimited: Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement*, in Concurrency and Computation: Practice and Experience, 2020, to appear; preprint, https://arxiv.org/abs/1909.06096, 2019.
[36] P. Samfass, T. Weinzierl, B. Hazelwood, and M. Bader, *TeaMPI—replication-based resiliency without the (performance) pain*, in Proceedings of the ISC High Performance 2020, Lecture Notes in Comput. Sci., Springer, Berlin, to appear.
[37] A. Sasidharan and M. Snir, *MINIAMR—A Miniapp for Adaptive Meshrefinement*, Tech. report, 2016, https://www.ideals.illinois.edu/handle/2142/91046.
[38] M. Schreiber, T. Weinzierl, and H. J. Bungartz, *Cluster optimization and parallelization of simulations with dynamically adaptive grids*, in Euro-Par 2013 Parallel Processing, F. Wolf, B. Mohr, and D. Mey, eds., Lecture Notes in Comput. Sci. 8097, Springer, Berlin, 2013, pp. 484–496.
[39] M. Sergent, M. Dagrada, P. Carribault, J. Jaeger, M. Pérache, and G. Papauré, *Efficient communication/computation overlap with MPI+Openmp runtimes collaboration*, in Euro-Par 2018: Parallel Processing, M. Aldinucci, L. Padovani, and M. Torquati, eds., Lecture Notes in Comput. Sci. 11014, Springer, Berlin, 2018, pp. 560–572.
[40] H. Sundar and O. Ghattas, *A nested partitioning algorithm for adaptive meshes on heterogeneous clusters*, in Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, ACM, New York, 2015, pp. 319–328, https://doi.org/10.1145/2751205.2751246.
[41] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and* 2:1 *balance refinement of linear octrees in parallel*, SIAM J. Sci. Comput., 30 (2008), pp. 2675–2708, https://doi.org/10.1137/070681727.
[42] M. Tavelli, M. Dumbser, D. E. Charrier, L. Rannabauer, T. Weinzierl, and M. Bader, *A simple diffuse interface approach on adaptive Cartesian grids for the linear elastic wave equations with complex topography*, J. Comput. Phys., 386 (2019), pp. 158–189, https://doi.org/10.1016/j.jcp.2019.02.004.
[43] C. Uphoff, S. Rettenberger, M. Bader, E. H. Madden, T. Ulrich, S. Wollherr, and A. A. Gabriel, *Extreme scale multi-physics simulations of the tsunamigenic* 2004 *Sumatra megathrust earthquake*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, 2017, 21, https://doi.org/10.1145/3126908.3126948.
[44] M. Weinzierl and T. Weinzierl, *Quasi-matrix-free hybrid multigrid on dynamically adaptive Cartesian grids*, ACM Trans. Math. Softw., 44 (2018), 32.
[45] T. Weinzierl, *The Peano software—parallel, automaton-based, dynamically adaptive grid traversals*, ACM Trans. Math. Softw., 45 (2019), 14.
[46] T. Weinzierl and M. Mehl, *Peano—a traversal and storage scheme for octree-like adaptive Cartesian multiscale grids*, SIAM J. Sci. Comput., 33 (2011), pp. 2732–2760, https://doi.org/10.1137/100799071.
[47] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76, https://doi.org/10.1145/1498765.1498785.