SIAM J. SCI. COMPUT.
Vol. 44, No. 2, pp. C156–C183

SCALABLE LOCAL TIMESTEPPING ON OCTREE GRIDS*

MILINDA FERNANDO† AND HARI SUNDAR‡

Abstract. Numerical solutions of hyperbolic partial differential equations (PDEs) are ubiquitous in science and engineering. Method of lines is a popular approach to discretize PDEs defined in spacetime, where space and time are discretized independently. When using explicit timesteppers on adaptive grids, the use of a global timestep size dictated by the finest grid-spacing leads to inefficiencies in the coarser regions. Even though adaptive space discretizations are widely used in computational sciences, temporal adaptivity is less common due to its sophisticated nature. This paper presents highly scalable algorithms to enable local timestepping (LTS) for explicit timestepping schemes on octrees. We demonstrate the accuracy of our methods and the scalability of our framework across 16K cores on TACC's Frontera supercomputer (https://frontera-portal.tacc.utexas.edu/). We also present a speedup estimation model for LTS that predicts the distributed speedup compared to global timestepping with an average of 0.1 relative error.

Key words. spacetime adaptivity, local timestepping, octree grids, PDEs

AMS subject classifications. 68U01, 68W10, 68U05, 65M06

DOI. 10.1137/20M136013X

1. Introduction.

The numerical solution of hyperbolic partial differential equations (PDEs) plays an important role in science in engineering, with a wide range of applications from modeling earthquakes [8] to simulating gravitational waves [26, 24, 35]. These show up commonly as initial value problems that are typically solved using the method of lines by first discretizing in space and then solving the resulting set of ordinary differential equations (ODEs).

The above is commonly done using timestepping schemes, with explicit timestepping methods such as the Runge–Kutta (RK) methods [32] being more common for evolving hyperbolic systems. Additionally, these systems are characterized by the need for high levels of spatial adaptivity [47, 24, 26]. High levels of adaptivity impose severe stability restrictions for the explicit timestepping schemes popular for solving such systems. Therefore, it is common—especially for large-scale distributed memory codes—to use a global (everywhere in space) timestep, that is dictated by the smallest grid resolution in space [22]. The above is highly inefficient for large systems with several orders of magnitude difference between the finest and coarsest grid resolution, as the coarser regions are forced to take extremely small timesteps compared to what would be needed for stability [31, 42]. Local timestepping schemes (LTSs) can greatly speed up such codes by ensuring that the adaptivity in space is matched by a corresponding adaptivity in time. In this work, we develop an efficient, scalable LTS for explicit single-stage and multistage timestepping algorithms on octree-adaptive spatial grids. The developed LTS methodology uses the spatial adaptivity to enable

*Submitted to the journal's Software and High-Performance Computing section August 17, 2020; accepted for publication (in revised form) September 20, 2021; published electronically April 13, 2022.

https://doi.org/10.1137/20M136013X

Funding: This work was supported by the National Science Foundation (NSF) through grants OAC-1808652, PHY-1912930 and by National Aeronautics and Space Administration (NASA) grant 80NSSC20K0528. The presented work used computing resources from the Texas Advanced Computing Center (TACC) allocation PHY20033.

†Oden Institute of Computational Engineering and Sciences, University of Texas at Austin, TX 78712 USA (milinda@oden.utexas.edu).

‡School of Computing, University of Utah, Salt Lake City, UT 84112 USA (hari@cs.utah.edu).

temporal adaptivity, but it is agnostic to how the spatial adaptivity is determined. We demonstrate our scheme's efficacy using linear and nonlinear wave equations on up to 16K processes on the Frontera supercomputer at the Texas Advanced Computing Center (TACC). We also present a model to estimate the expected speedup from using our scheme with an average of 10% relative error in the estimation of speedup. This derived speedup model provides a reliable way to determine in which cases the use of local timestepping can be beneficial.

Our framework allows space adaptivity via the use of octrees (quadtrees in two dimensions) and uses high-order finite difference (FD) methods for space discretization. Octree-based adaptive space discretizations [19, 33, 29, 43, 54] are popular in large-scale simulations because of their quasi-structured nature, allowing for efficient and scalable data structures and algorithms. Our framework targets applications in computational relativity [26] and uses a 3+1 decomposition of spacetime operators to compute a space slice where time is constant and uses standard time integration methods (explicit) to evolve the space slice forward in time. The time integration method of choice has been the RK method because of the larger stability region and the availability of low-storage versions. For the specific problem of simulating binary black hole mergers to estimate gravitational waves, the spatial grid is characterized by 12 to 22 levels of adaptivity and requires 10^6 to 10^9 global timesteps to attain the final solution. This results in simulations that need to run for months on thousands of processes. The use of LTS for these simulations can significantly speed them up, which can greatly reduce the time and cost of obtaining gravitational waveforms.

While the theoretical aspects of LTS have been an area of active research in recent years [40, 41, 34, 46, 5], performing LTS in a distributed computing environment comes with additional challenges. A central bottleneck to scalability with LTS is the variability in computational loads for different regions of space based on their spatial adaptivity, as finely refined regions take exponentially more timesteps than coarser regions. The above requires partitioning approaches that can account for such variable workloads to ensure that the parallel constraints of LTS do not negate the overall speedup. Additionally, since in our target applications the meshes are dynamic, frequent repartitioning is required, requiring fast partitioning algorithms that are able to adapt to the variable computational loads. While the use of graph partitioning approaches [46] is likely to produce superior partitions, the cost of partitioning our meshes in parallel makes the approach infeasible. The key contributions presented in the paper can be summarized as follows.

- **Scalable LTS on octrees**: We present a scalable LTS framework for multistage explicit methods on 2:1 balanced octree grids. To the best of our knowledge, existing octree frameworks [17, 58, 7, 56] are limited to space adaptivity.
- **Load balancing in LTS**: The number of local timesteps needed to reach the coarsest time on the grid depends on the spatial adaptivity. This leads to load balancing issues in LTS. To resolve this, we propose a weighted partitioning scheme based on a space filling curve (SFC). Compared to traditional SFC-based partitioning, we compute the weighted length of the curve to achieve a balanced load for LTS partitions.
- **Low overhead of LTS compared to GTS**: We present both strong and weak scaling results for LTS and GTS approaches on octrees. These results demonstrate that local block synchronization in LTS followed by time interpolations has a lower cost than global block synchronization present in the GTS approach.
- **Accuracy of LTS**: We conduct numerical experiments to demonstrate the correctness of the implemented LTS framework. The presented numerical

results demonstrate the accuracy of the LTS framework for both linear and nonlinear problems.

- **LTS performance model**: We present an analytical performance model to estimate the speedup of LTS over the GTS scheme. The analytical model is extended to compute a theoretical upper bound for the speedup that can be achieved for a given spatial adaptivity structure.

Organization of the paper. The rest of the paper is organized as follows. In section 2, we give a brief motivation on the importance of LTS and a quick overview of the existing state-of-the-art approaches in the field. In section 3, we present the algorithms and methods developed in detail to compare its efficiency to GTS. In section 4, we discuss the experimental setup, demonstrate strong and weak scalability of our approach, and demonstrate the accuracy of our scheme. In section 5, we conclude with directions for future work.

2. Background. In comparison to spatial adaptivity, local time adaptivity is less frequently used by large-scale applications. Local timestepping requires additional corrections in mismatching regions in time using interpolations or extrapolations. Depending on the differential operator properties, these operations can lead to problems in stability [30] of the numerical scheme. Recent LTS methods have been influenced by the split RK methods [45], where two ODE systems are integrated using different step sizes (one called *active* with the smaller timestep and the other called *latent* with the larger timestep size) on the grid. Corrections using interpolations are performed for the interface between the two grid regions. A complete numerical analysis of spacetime adaptive timestepping methods is complicated, but early work by Berger provided the first mathematical analysis for adaptive schemes for the wave equation [12, 13]. Algorithms presented in these papers discuss two main approaches, interpolation-based and coarse mesh approximations. In the interpolation methods, the coarse mesh solution is used to interpolate the values needed for the finer mesh. In the coarse mesh approximation, the coarse mesh is used to take a pseudotimestep used by the finer mesh. In [21], the authors present methods for energy-conserving corrections in time for Maxwell's equations. There is a rich literature of LTS for discontinuous Galerkin methods, focusing on energy-conserving time correction operators [49, 38, 42, 39] which are important for complicated nonlinear spacetime differential operators. As mentioned previously, to enable LTS in a distributed parallel setting requires specialized partitioning methods to ensure load balance. Dynamic load balancing for adaptive mesh refinement (AMR) is an active research area [9, 25, 20, 28]. Sophisticated hypergraph partitioning techniques have been used for LTS for the wave equation [46]. However, the cost of partitioning makes it prohibitively expensive for AMR applications requiring frequent remeshing and, therefore, repartitioning.

AMR in space and spacetime is an active research area. Here we present a brief overview of AMR packages that focus on both space and spacetime. Block-structured or patch-based AMR is widely used in astrophysics and computational fluid dynamics communities. In block-structured AMR, the adaptivity structure is predetermined and evolved during the simulation appropriately. Some codes support LTS with block AMR [16, 24, 6], primarily based on the Berger–Oliger AMR criteria [14]. Berger–Oliger AMR criteria support adaptivity in space and time but requires strong constraints on the structure of the adaptivity, such as all grids at level $l+1$ (child grids) should be entirely contained within the grids at level l (parent grids) while grids at the same level may overlap. There exist other block-AMR codes [2], which only support space adaptivity and no adaptivity in time.

Another commonly used approach for large-scale AMR is octree-based AMR [17, 58, 7, 56, 50, 51]. In octree-based AMR, the adaptive grid is represented using quadtrees and octrees. Unlike block-based AMR, octree-based AMR has relaxed constraints on refinement, providing highly adaptive quasi-structured (point-structured) grids in space. Currently available octree-AMR codes are limited only to space adaptivity with no support to enable adaptivity in time to the best of our knowledge. This paper mainly focuses on developing scalable parallel algorithms and data structures to enable LTS on octree grids. We choose the simple time interpolation methods presented in the papers [40, 41] which form the mathematical basis for the methods presented in this work. Additional details and analysis can be found in [40, 41].

3. Methodology. This section presents parallel algorithms and data structures used to enable LTS on adaptive octree grids. This paper mainly focuses on numerical solutions of hyperbolic PDEs using the method of lines approach, where time discretization is performed using explicit timestepping methods.

3.1. 3+1 decomposition of PDEs. In this paper, we focus on differential operators defined on the traditional Euclidean spacetime ($\mathcal{R}^4$) that is (3 space + 1 time dimension). Let $\mathcal{L}$ be a differential operator, $\mathcal{L} : X \to Y \subset X$ where X, Y are Banach spaces with appropriate smoothness conditions which $\mathcal{L}$ acts upon. For example, when $\mathcal{L} \equiv \partial_t - \partial^2_{xx}$, we get the heat operator, or when $\mathcal{L} \equiv \partial^2_{tt} - \partial^2_{xx}$ one attains the traditional wave operator. Throughout this paper, we focus only on the operators $\mathcal{L}$, which can be transformed into an evolution equation of the form (3.1), which we refer to as $3 + 1$ decomposition of operator $\mathcal{L}$,

$$\partial_t u = F(t, u(t)) \tag{3.1}$$

where, for $T \in \mathcal{R}^+$, $F : [0, T] \times W \to X$, and $W \subset X$. For a given $s \in [0, T)$ and $\phi \in W$, the solution u is the integral curve of F, that satisfies (3.1) on $[s, T]$ with the initial condition $u(s) = \phi$. Analysis of the well-posed nature of these integral curves is out of the scope of this paper, hence we assume these integral curves are well-posed, and can be computed with numerical timestepping, with the appropriate necessary stability constraints.

3.2. Spatial adaptivity. While this paper is centered on adaptivity in time, it is closely related to our realization of adaptivity in space using octrees. We give a brief overview of our spatial adaptivity framework (Dendro-5.01) in this section. The Dendro-5.01 framework is freely available via an MIT license [27] and additional details on our algorithms can be found in [25, 26, 55]. Octrees are widely used [48, 17, 58, 10] in computational sciences for their simplicity, efficient data structures, ease of partitioning, and parallel scalability. We use axis-aligned octrees to represent the underlying spatial discretization (see Figure 1). Only the leaf nodes are stored, and all the nonleaf nodes are discarded to reduce the overall octree memory footprint. The above is mainly because nonleaf nodes are redundant and can be computed with a single bottom-up traversal on the tree structure. For a given spatial domain $\Omega = [-a, a]^3$, where $a \in \mathcal{R}^+$, and let $F(x, y, z)$ be a field (i.e., scalars, vectors, and tensors) defined on Ω. Then the field F can be adaptively discretized using an octree data structure. To enable efficient hierarchical splits using bitwise operations, we map the domain Ω to integer domain $\Omega_I = [0, 2^L]$, where L is the maximum refinement level allowed in the octree. Therefore leaf nodes are stored as a four-integer tuple, (x, y, z, l), where (x, y, z) denotes the leftmost corner of a leaf node octant, and l denotes the leaf

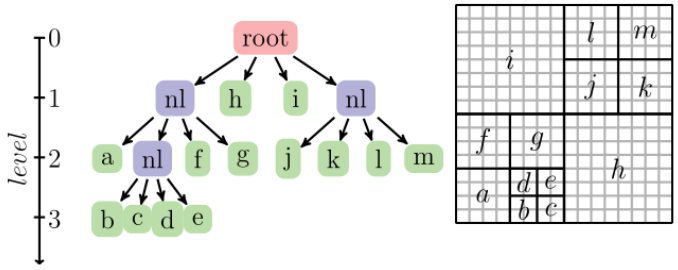


FIG. 1. *An illustration of axis-aligned hierarchical splitting of a two-dimensional (2D) domain with the underlying quadtree. For an adaptive discretization, the leaf nodes (i.e.,* $\{a, b, c, \ldots, l, m\}$*) will be at different levels. Note that when storing the quadtree, nonleaf nodes are discarded since they can be computed using bottom-up traversal on the tree structure. The above generalizes for 3D domains where the underlying data structure becomes an octree.*

node level (i.e., a leaf node at level l has the size 2^{L-l}). The presented LTS approach is agnostic to the adaptivity criteria where it can be application-specific. The adaptivity criteria can be represented as a function,

$$A : \{\text{octree node set}\} \rightarrow \{refine, coarse, no\_change\}, \tag{3.2}$$

where the specific definition of A is application-specific. For example, A can be defined based on gradient jump between neighboring octants, interpolation error reduction from the parent node to the child nodes refinement, and any other criteria specific to the governing PDEs.

3.3. Distributed octrees. The partitioning of the spatial domain is required to achieve parallelism in space. In a sequential octree, all the leaf nodes are stored in a single processor. In contrast, in a distributed octree, the leaf nodes are partitioned across the specified p number of processors. When computing the octree partitions, it is crucial to consider work and communication balanced partitions to achieve efficient parallel scalability. In AMR applications, the adaptivity structure should evolve with the underlying fields (i.e., capturing a propagating wave). The change in the refinement structure can make the previously computed partitions load imbalanced. In AMR applications, frequent repartitions are required to achieve proper load balancing. Therefore these partitioning methods should be fast and efficient yet produce partitions with higher quality (i.e., work and communication balanced). SFCs are commonly used by the HPC community for partitioning data [18, 23, 57] and for resource allocations [11, 52]. By mapping high-dimensional spatial coordinates (i.e., octants) onto a 1D curve, the task of partitioning is made trivial (see Figure 2). In this work, we use the SFC-based partitioning scheme OPTIPART [25] to partition the octree data.

3.3.1. Octree construction. This section presents a brief overview of the distributed octree construction with application-specific refinement criteria A (see (3.2)). The tree construction is a distributed process where all processors start with zero refinement level (i.e., the root node). The specified refinement criteria A is called on the root node, where if A indicates to refine, children of the root node are added. Once the new nodes (i.e., children) are added, partitioning algorithm OPTIPART is executed

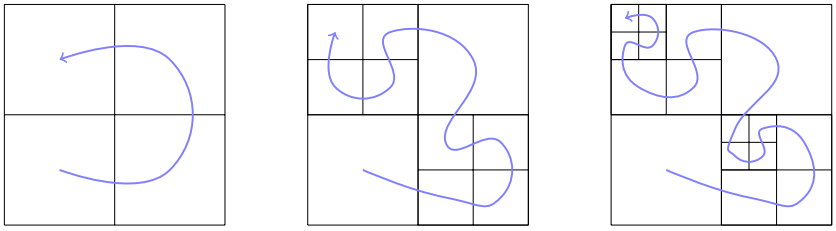
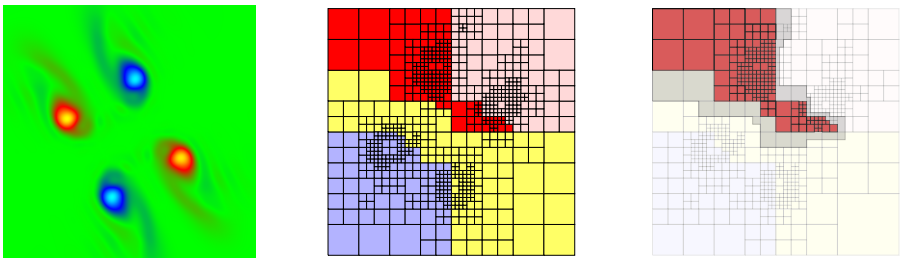


FIG. 2. *A simple 2D example of how SFCs can induce an ordering operator on 2D quadtree nodes. The 2D spatial coordinates can be ordered based on the traversal order induced by the SFC. SFCs can be used to reduce the partitioning problem from high-dimensional space to 1D space.*



(a) $f : \Gamma \to \mathcal{R}$ (b) constructed octree (c) ghost/halo layer

FIG. 3. *The leftmost figure shows a scalar field f defined on $\Gamma \subset \mathcal{R}^2$. The middle figure shows the constructed 2:1 balanced distributed quadtree to capture the field f with respect to the specified refinement criteria A. Note that the quadtree partitions are color coded. The rightmost figure shows a ghost/halo element layer for the selected single partition.*

to repartition the current tree to ensure load balance. The above process is continued until the refinement structure does not change (see Algorithm 3.1). Following the octree construction, we enforce a 2:1 refinement constraint (i.e., 2:1 balancing). The 2:1 balancing condition ensures that for any given leaf node in the octree, its neighboring leaf nodes at most differ by a single level (i.e., an octant can have a neighbor that is either the same level, one level coarser, or one level finer). The key motivation behind enforcing the 2:1 balancing constraint is to ease the subsequent numerical operations on octree grids simpler (see Figure 3). There exist different 2:1 balancing algorithms [36, 53, 15, 56] in the field, and our balancing algorithm is similar to existing approaches for balancing octrees [56] with minor changes in the choice of data structures and process-local balancing.

3.4. Data structures. Up until now, we have discussed adaptive octree-based spatial discretizations. This section discusses distributed data structures required to enable numerical computations on octrees. In this paper, the terms "meshing" and "mesh generation" refer to constructing the data structures required to enable numerical computations on adaptive octree grids. The notion of neighbors and neighborhood connectivity is crucial to enable numerical computations on AMR grids. The neighborhood information that is needed depends on the nature of the numerical computation. For example, the FD and finite volume (FV) methods will require neighborhood information proportional to the used stencils' width. In contrast, finite element methods will require nodal information corresponding to each element. In adaptive discretizations, finding this neighborhood information becomes nontrivial and requires additional search operations on the octree. We use an efficient TREESEARCH [26]

Algorithm 3.1 Initial distributed octree construction (TreeConstruction)

Require: A refinement criteria, L maximum refinement level, p number of ranks
Ensure: T distributed tree
1: $T \leftarrow$ RootNode
2: $C \leftarrow \emptyset$
3: **while** $T \neq C$ **do**
4: $C \leftarrow \emptyset$
5: **for** $node \in T$ **do**
6: **if** $nodel.level < L$ and $A(node) == refine$ **then**
7: $C \leftarrow C \cup Children\_Of(node)$
8: **else**
9: $C \leftarrow C \cup node$
10: $C \leftarrow$ OptiPart(C, p) ▷ Partition C across p ranks
11: $swap(T, C)$
return T

algorithm to build these data structures. To make the constructed neighborhood information process-local, we compute a layer of ghost/halo octants for each partition. For a given distributed octree $T = \{t_1, t_2, \ldots, t_p\}$ where t_k denotes the kth partition out of p ranks (i.e., $k \leq p$), the ghost octant layer consists of octants that are neighboring to t_k but belong to another partition t_i where $i \neq k$ (see Figure 3). Following distributed octree construction, the ghost layer for each partition is computed using parallel sort and search operations [25, 26]. The computed ghost octant layer ensures the constructed subsequent data structures are process-local. To perform LTS on octree grids requires elemental and nodal level neighborhood information. At first we compute an octant to octants (o2o) map where for a given octant $e \in t_k$, o2o$(e) = \{e_1, e_2, \ldots, e_6\}$ where $\{e_1, e_2, \ldots, e_6\}$ are the face neighboring octants to e. The map o2o is computed by generating search keys on each face direction followed by search operations on the octree.

Embedding nodal information. The constructed o2o map is used to embed nodal information on the octree. To enable dth-order spatial interpolations, for each octant $(d+1) \times (d+1) \times (d+1)$ nodes are uniformly spaced. The above is referred to as *octant local nodes* (i.e., the ownership of the nodes is local to the octant). Note that, between the shared octant boundaries, there will be duplicate nodes. The *shared octant nodes* representation is computed to eliminate nodal duplications. The nodal ownership is determined by the globally consistent criteria (i.e., SFC ordering of the octants). In the case of finer and coarser octant boundary, the finer nodal points (also called "hanging nodes") are discarded [26, 37], since they can be computed with interpolation from the coarser grid points assuming that the AMR criterion A determines appropriate adaptivity structure (see Figure 4). The octant to nodal (o2n) map maps a given octant $e \in t_k$ to its *shared octant nodes* indices. The computation of o2n is process-local where it initially starts with *octant local nodes* and removes duplicate and hanging nodes with proper nodal indices updates (see Figure 5).

3.5. Time adaptivity identification. For a given octree $\mathcal{T}$, we compute a compressed octree of $\mathcal{T}$, denoted as $\mathcal{B}$ (blocks) where each leaf node in $\mathcal{B}$ is a node in $\mathcal{T}$ with uniformly refined suboctree (see Figure 6). The above is referred to as octree to block decomposition, where the blocks are computed by performing top-down traversal with additional octant level constraint. The block decomposition transforms the adaptivity on the octree to a sequence of uniform grid patches. The above (1) identifies the time adaptivity (the refinement level on the block determines timestep size) and (2) enables numerical computations such as FD and FV methods on adaptive grids (i.e., to evaluate the right-hand side (RHS) in (3.1)).

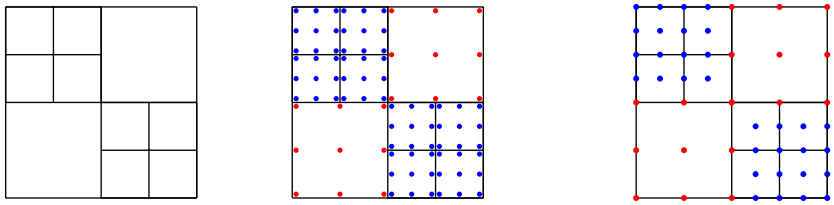


FIG. 4. *A 2D example of octant local nodes (in the center) and shared octant nodes (the rightmost figure) nodal representation (for octant order of 2) of the adaptive quadtree shown in the leftmost figure. Note that in octant local nodes, representation nodes are local to each octant and contain duplicate nodes. By removing all the duplicate and hanging nodes by the rule of nodal ownership, we get the shared octant nodes representation. Note that the nodes are color coded based on the octant level.*

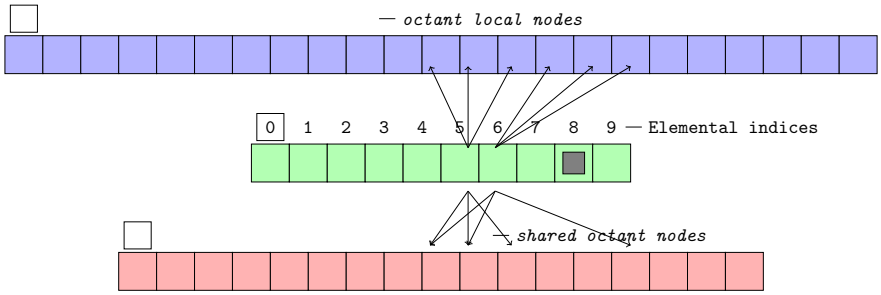


FIG. 5. *An illustration of the underlying representation of the elemental and nodal vectors. The elemental vectors are used to store local time for each octant, while octant local nodes representation is used to store solution vectors that are not synchronized in time. Note that the duplicate nodal information allows two neighboring octants to be at different evolved times. The shared octant nodes representation is used to represent solutions that are synchronized in time. The figure also illustrates how elemental and nodal indices are connected through the constructed element to nodal indices maps.*

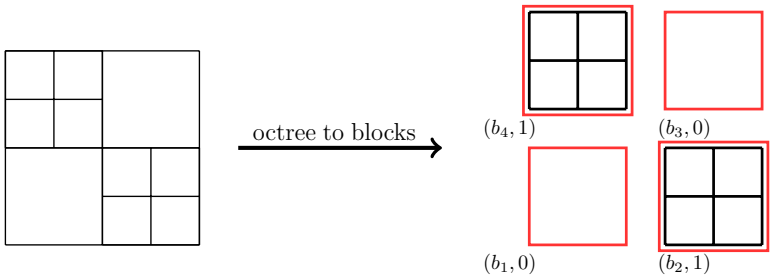


FIG. 6. *A simplistic example of octree ($\mathcal{T}$) to block decomposition. The left figure shows the considering adaptive octree, and its block decomposition is shown on the right. Each block (b_k, l) is associated with uniform grid level parameter l, where l denotes the level of refinement of the subtree rooted at b_k node in $\mathcal{T}$.*

3.6. Numerical computations on octree grids. This section presents a brief overview of how the constructed data structures are used to perform numerical computations on octree grids. To evaluate the RHS during the time integration of (3.1) requires the ability to perform numerical computations such as finite element (FE), FD, and FV type computations on octree grids. Performing FD and FV type compu-

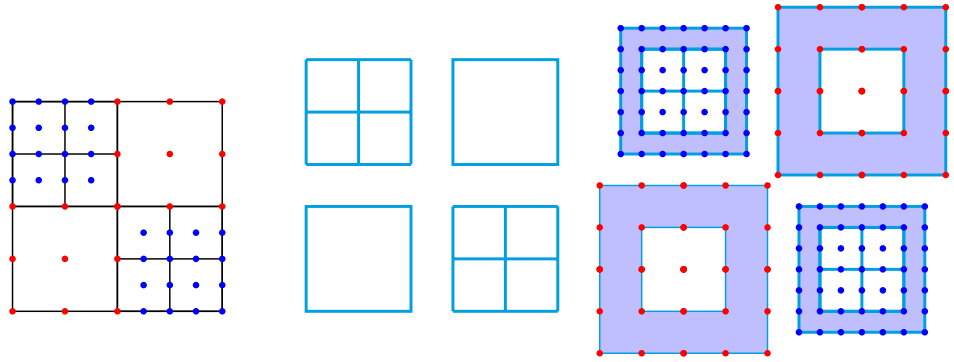


FIG. 7. *A simplistic example of octree to block decomposition and unzip operation. The leftmost figure shows the considering adaptive octree with shared octant nodes, and its block decomposition is shown in the middle. Note that the given octree is decomposed into four regular blocks of different sizes. The rightmost figure shows the decomposed blocks padded with values coming from neighboring octants with interpolation if needed. In order to perform the unzip operation, both* O2O *and* O2N *mappings are used.*

tations requires neighboring nodal information, while FE type computations require element local neighborhood information. The work presented mainly focuses on FD computations to approximate spatial derivatives while the proposed LTS methodology is extensible for FE and FV type operator approximations. To perform FD stencil computations requires spatial grid points that are at the same resolution. The above is achieved by performing octree to block decomposition where for each block, we compute a padding region of a specific width corresponding to the FD stencil width. In order to compute the padded blocks, we use the computed O2O and O2N information with the corresponding space interpolations. For example, between finer and coarser grid blocks, we use coarser to finer interpolation (i.e., parent octant to child octants) to compute the padding region for the finer block while using finer to coarser injections for the padding region of the coarser block. Note that in the paper, the octant shared node representation is also referred to as *zipped* representation (see Figure 4) and block with padding computed is referred to as the *unzipped* representation. All the FD stencils are applied at the *unzipped* representation, while just prior to the communication, we perform a *zip* operation, so the interprocess communication happens in the efficient, more compact form (see Figure 7).

3.7. Explicit timestepping schemes. Explicit timestepping is a class of numerical schemes that compute the solution curve for (3.1). In explicit methods, the solution at time t_{n+1}, $u^{n+1} \equiv u^{n+1}(t_{n+1}, \cdot)$ is computed directly from the solution at the previous timestep u^n and does not require a linear solve. In order to numerically evolve (3.1), we discretize F, i.e., discretization in space first. The resulting set of ODEs are discretized using explicit time integration. Depending on the properties on $\mathcal{L}$, there can be additional constraints on $\Delta t, \Delta x$. For most hyperbolic operators, the Courant–Friedrichs–Lewy (CFL) condition [22] is a necessary condition for stability for numerical time evolution. The CFL condition $\frac{\Delta t}{\Delta x} < C$, where C is a constant that specifies a necessary condition for stability. Intuitively, it imposes the constraint that we cannot propagate spatial information in time, faster than the speed of information propagation defined by operator $\mathcal{L}$. RK [44, pp. 350, 379] schemes are widely used explicit timesteppers (see (3.3)) that will be our main focus for spatially adaptive local timestepping.

$$
\begin{aligned}
k_1 &= F(u^n),\\
k_2 &= F(u^n + a_{21}k_1),\\
&\vdots\\
k_p &= F(u^n + a_{p1}k_1 + \cdots + a_{pp-1}k_{p-1}),\\
u^{n+1} &= u^n + \Delta t \left(\sum_{i=1}^{p} w_i k_i \right).
\end{aligned}
\tag{3.3}
$$

3.8. GTS: Global timestepping. This section presents the global timestepping (GTS) approach on octrees assuming FD approximations for spatial derivatives. The octree to block decomposition is computed to enable FD computations on adaptive grids. In GTS, all the blocks are evolved using the same timestep size (i.e., no temporal adaptivity) where the stability constraints dictate the timestep size. For example, for hyperbolic PDEs, with explicit GTS, the timestep size is determined by the grid's finest spatial resolution. Let $u^n \equiv u(t_n, \cdot)$ be the solution at time t_n. In GTS, first we compute the *unzipped* representation for u^n. The above allows evaluation of the RHS using FD approximations. Once the RHS is evaluated, we perform the *zip* operation followed by intermediate timestepping stage computation. After the intermediate stage computation, a global block synchronization (i.e., get updated values for the block padding region) is performed using the *unzip* operation (see Figure 8). The above process is performed to evaluate the required timestepping stages,

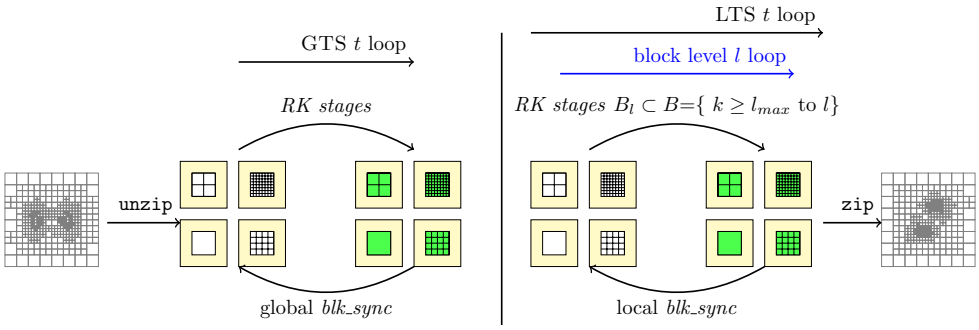


FIG. 8. *This figure illustrates the overview of time evolution using GTS and LTS for multistage explicit timestepping on octree grids. In order to perform FD computations, the adaptive grid is decomposed into uniform block patches with appropriate padding, and spatial derivatives are evaluated on equispaced block representation (unzipped) computed using previous timestep solution u_n. The unzip operation results in a sequence of that which are used to compute the solution on the internal block (■), using the padding values at the block boundary (■). After time evolution, the next timestep u_{n+1} is projected back to sparse grid (*`zip`*) representation. In GTS, for each explicit stage, we evolve all the blocks using Δt_{finest}, followed by a global block synchronization operation. This global synchronization operation consists of the projection of block local solution for zipped representation (shared octant nodes), followed by interprocess communication and project back to unzipped representation. Note that shared octant nodes are used for interprocess communication since it is compact and does not contain duplicate nodal values. For LTS, we have a block level loop that selects a subset of blocks B_l which are eligible to evolve, followed by the explicit stage loop. Once B_l is evolved, we perform block synchronization for B_l. Hence the above is a local block synchronization. For this local block synchronization, for interprocess communication, we use octant local nodes representation (unzipped) without padding region). Unlike GTS we cannot use shared octant nodes representation since blocks evolved are at different times.*

Algorithm 3.2 Global timestepping

Require: U^n previous timestep, Δt, B sequence of blocks
Ensure: $U^{n+1} = U(t^n \Delta t, \cdot)$
1: $u\_unzip \leftarrow unzip(U^n)$
2: **for** $s = 1 : k$ **do**
3: **for** $b \in B$ **do**
4: $k[b, s] \leftarrow compute\_stage(u\_unzip, k[1, \ldots, s-1])$
5: $k[s] \leftarrow blk\_sync(k[s])$
6: $U^{n+1} \leftarrow compute\_step(U^n, k)$

and once all the stages are computed the solution u^n is updated to the next timestep $u^{n+1} \equiv u(t_n + \Delta t, \cdot)$. The GTS approach is presented in Algorithm 3.2.

3.9. LTS: Local timestepping. This section presents the methodology to perform LTS on spatially adaptive octree grids. In LTS, the timestep size is adapted to the spatial resolution on the grid. The above is achieved using the block representation of the octree (i.e., octree to block decomposition), where for each block, the timestep size is determined by the resolution of the block. For example, a block with the finest resolution L uses the timestep size Δt, and a block at the resolution $l < L$ uses a timestep size of $2^{L-l}\Delta t$. Assuming that the use of the same CFL constant c on the grid where $\Delta t = c\Delta x$, and the 2:1 balance property on the octree grid ensures that for any coarser and finer grid boundary, the timestep size ratio between coarser (Δt_c) and finer (Δt_f) is equal to two (i.e., $\Delta t_c = 2\Delta t_f$). In LTS, the blocks are evolved using different timestep sizes. Hence the block padding region synchronization requires a notion of synchronization in time in addition to space synchronization. The above can be achieved mainly in two ways: (1) coarse grid approximation (i.e., full-step-half-step method) or (2) perform interpolations in time.

3.10. LTS for single-stage explicit schemes. For single-stage explicit schemes such as forward Euler, we can enable the correction between coarser (b_c) and finer (b_f) blocks by making b_c take a pseudotimestep for block b_f. The key advantage of the above is that the time synchronization between the coarser and the finer grid boundary does not require time interpolations. Temporal synchronization between coarser and finer refinement boundaries is computed by the coarser grid using the appropriate timestep size for the finer block. The above is referred to as the *full-step-half-step* method (see Figure 9). For single-stage timesteppers, the full-step-half-step approach is ideal since the half-step approximation is only needed by a single layer of coarser blocks surrounding a finer block. The implementation of the full-step half-step approach for multistage schemes is challenging due to complicated block synchronization patterns. Also, in multistage methods, the number of coarser block layers that have to take a half-step is proportional to the number of intermediate stages (N_s) in the scheme. The application of the full-step-half-step method will cause a significant loss in the temporal adaptivity for multistage schemes.

3.11. LTS for multistage explicit schemes. In multistage explicit methods, the time evolution of (3.1) is given in (3.3). For 2:1 balanced grids, the resolution difference between block boundaries is bounded by a factor of two. Therefore, for any given two neighboring blocks in the octree, their refinement level difference can be either zero (i.e., same refinement level) or one (i.e., coarser finer refinement boundary). In LTS, the timestep size used by a block depends on its refinement level. Hence, for multistage methods, computed intermediate stages (k_i) need to be synchronized

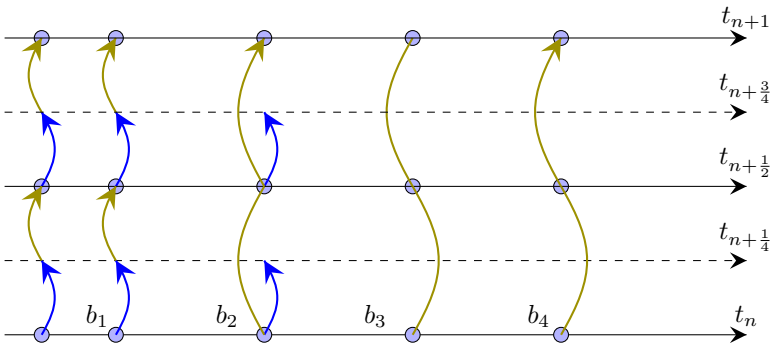


FIG. 9. *Simple illustration of local timestepping for single-stage schemes on* 2:1 *balanced adaptive grids. Here we have a refined block* b_1 *with neighbor* b_2 *that can be at most twice as large. We also consider block* b_3 *that doesn't have any neighbors smaller than itself. Note that* 2:1 *balancing ensures that these are the only cases that can exist for any adaptive mesh. Blocks* b_1 *and* b_3 *take timesteps corresponding to the size of the blocks. Block* b_2*, however, takes* $2\times$ *as many timesteps compared to* b_3*, first a half-step to help its neighbor* b_1 *take its second timestep, and then a full size timestep to reach* t_{n+1}*.*

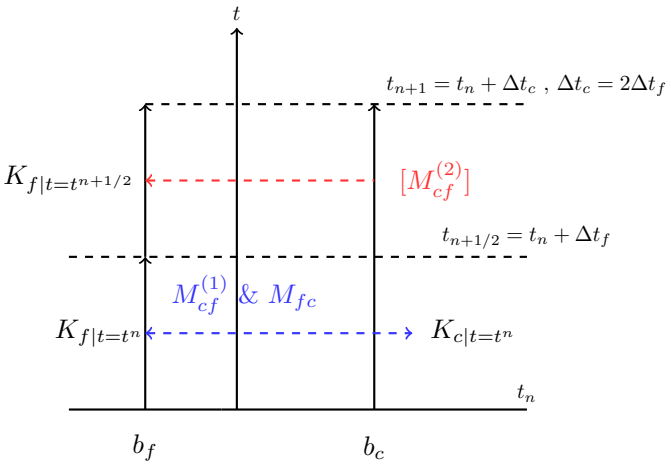


FIG. 10. *A simple illustration of how neighboring coarser* (b_c) *and finer* (b_f) *blocks perform synchronization in LTS for multistage timesteppers. The solution at coarser and finer regions is synchronized at time* t_n*. The* b_c *block reaches the time* $t_{n+1} = t_n + \Delta t_c$ *using a single timestep while block* b_f *reaches the same time using two timesteps with the intermediate step at the time* $t_{n+1/2} = t_n + \Delta t_f$*. During the first timestep block* b_f *computes the coarser grid correction using* M_{fc} *and block* b_c *computes the finer grid correction using* $M_c^{(1)} f$*. At the end of the first timestep, block* b_c *waits till block* b_f *performs the second step. During the second timestep block* b_c *computes the finer grid correction using the* $M_c^{(2)} f$ *operator.*

between refinement boundaries to accommodate spatially adaptive timestep sizes. Let b_f, b_c be two neighboring blocks where b_c is one level coarser than b_f (see Figure 10). Let $\Delta t_f, \Delta t_c = 2\Delta t_f$ denote the timestep sizes for blocks b_f, b_c respectively. Assume at time t_n the solution u^n at b_f, b_c are synchronized, and the solution needs to be evolved to time $t_{n+1} = t_n \Delta t_c$. The b_c solution will reach time t_{n+1} in a single timestep while the b_f solution needs to perform two timesteps to reach t_{n+1}. Let $t_{n+1/2} = t_n + \Delta t_f$ denote the solution at the intermediate timestep taken by the

block b_f. The solution in the coarser block is evolved using Δt_c, and as per formula (3.3) we can write time integration given in (3.4).

$$
\begin{aligned}
k_{c,1} &= F(u_c^n),\\
k_{c,2} &= F(u_c^n + a_{21}k_{c,1}),\\
&\vdots\\
k_{c,p} &= F(u_c^n + a_{p1}k_{c,1} + \cdots + a_{pp-1}k_{c,p-1}),\\
u_c^{n+1} &= u_c^n + \Delta t_c \left(\sum_{i=1}^{p} w_i k_{c,i} \right).
\end{aligned}
\tag{3.4}
$$

For the finer block b_f we need to perform two timesteps to reach time t_{n+1}. The b_f solution initially reaches the intermediate time $t_{n+1/2}$ as specified in (3.5).

$$
\begin{aligned}
k_{f,1}^{(1)} &= F(u_f^n),\\
k_{f,2}^{(1)} &= F(u_f^n + a_{21}k_{f,1}^{(1)}),\\
&\vdots\\
k_{c,p}^{(1)} &= F(u_c^n + a_{p1}k_{c,1} + \cdots + a_{pp-1}k_{f,p-1}^{(1)}),\\
u_f^{n+1/2} &= u_f^n + \Delta t_f \left(\sum_{i=1}^{p} w_i k_{f,i}^{(1)} \right).
\end{aligned}
\tag{3.5}
$$

Similarly, for the second timestep for block b_f we can write time integration equations as specified in (3.6).

$$
\begin{aligned}
k_{f,1}^{(2)} &= F(u_f^{n+1/2}),\\
k_{f,2}^{(2)} &= F(u_f^{n+1/2} + a_{21}k_{f,1}^{(2)}),\\
&\vdots\\
k_{c,p}^{(2)} &= F(u_c^{n+1/2} + a_{p1}k_{c,1} + \cdots + a_{pp-1}k_{f,p-1}^{(2)}),\\
u_f^{n+1} &= u_f^{n+1/2} + \Delta t_f \left(\sum_{i=1}^{p} w_i k_{f,i}^{(2)} \right).
\end{aligned}
\tag{3.6}
$$

The following operations are needed to synchronize the LTS solution between the finer and the coarser grid boundaries.

- **Finer to coarser ($\boldsymbol{M_{fc}}$)**: The stages vector $\{\bar{k}_{f,i}\}_{i=}^{p}$ should be approximated from the available finer block (b_f) stages $\{k_{f,i}^{(1)}\}_{i=}^{p}$ to synchronize timestepping stages in the coarse block b_c to enable timestep from t_n to t_{n+1}.
- **Coarser to finer 1 ($\boldsymbol{M_{cf}^{(1)}}$)**: The stages vector $\{\bar{k}_{c,i}^{(1)}\}_{i=}^{p}$ should be approximated from the available coarser block stages vector $\{k_{c,i}\}_{i=}^{p}$ to synchronize the stages in the finer block b_f to enable the first timestep (i.e., from t_n to $t_{n+1/2}$).
- **Coarser to finer 2 ($\boldsymbol{M_{cf}^{(2)}}$)**: The stages vector $\{\bar{k}_{c,i}^{(2)}\}_{i=}^{p}$ should be approximated from the available coarser block stages vector $\{k_{c,i}\}_{i=}^{p}$ to synchronize the stages in the finer block b_f to enable the second timestep (i.e., from $t_{n+1/2}$ to t_n).

The derivation of the above operators can be motivated by various numerical constraints. In this paper, we derive these operators based on the linear approximation of the $F(u)$ for a given timestep interval Δt. Using (3.3) and the piecewise linear approximation of the RHS $F(u)$, we get

$$K = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_p \end{bmatrix} = C \times P_{\Delta t} \times \begin{bmatrix} \partial_t u \\ \partial_t^2 u \\ \vdots \\ \partial_t^p u \end{bmatrix}_{t=t_n}, \tag{3.7}$$

where C and $P_{\Delta t}$ are given by

$$P_{\Delta t} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & \Delta t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \Delta t^{p-1} \end{bmatrix}, C = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & c_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & c_{p2} & \cdots & c_{pp} \end{bmatrix}. \tag{3.8}$$

The coefficients of C are related to the explicit timestepper coefficients given in (3.9).

$$\begin{aligned} c_{22} &= a_{21}, \\ c_{32} = a_{31} + a_{32},\ & c_{33} = a_{32}a_{21}, \\ c_{42} = a_{41} + a_{42} + a_{43},\ & c_{43} = a_{43}a_{32} + a_{43}a_{31} + a_{42}a_{21}, \\ c_{44} &= a_{43}a_{32}a_{21}, \\ &\vdots \\ c_{p2} = a_{p1} + \cdots + a_{pp-1},\ & c_{p3} = a_{pp-1}a_{p-1p-2} + \cdots + a_{p2}a_{21}, \\ c_{pp} &= a_{pp-1}a_{p-1p-2} \ldots a_{21}. \end{aligned} \tag{3.9}$$

For example, the C coefficient matrix for RK3 and RK4 is given in (3.10). The P matrix is computed at the runtime with the appropriate timestep size.

$$C_{RK3} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1/2 & 1/4 \end{bmatrix}, C_{RK4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1/2 & 0 & 0 \\ 1 & 1/2 & 1/4 & 0 \\ 1 & 1 & 1/2 & 1/4 \end{bmatrix}. \tag{3.10}$$

Equation (3.7) can be used to compute $\{\bar{k}_{f,i}\}_{i=}^{p}$ from $\{k_{f,i}^{(1)}\}_{i=}^{p}$ (i.e., M_{fc} operator) as given in (3.11), where M_{fc} can be written as $M_{fc} = CP_{\Delta t_c}P_{\Delta t_f}^{-1}C^{-1}$.

$$\begin{bmatrix} \bar{k}_{f,1} \\ \bar{k}_{f,2} \\ \vdots \\ \bar{k}_{f,p} \end{bmatrix} = CP_{\Delta t_c}P_{\Delta t_f}^{-1}C^{-1} \begin{bmatrix} k_{f,1}^{(1)} \\ k_{f,2}^{(1)} \\ \vdots \\ k_{f,p}^{(1)} \end{bmatrix} = M_{fc} \begin{bmatrix} k_{f,1}^{(1)} \\ k_{f,2}^{(1)} \\ \vdots \\ k_{f,p}^{(1)} \end{bmatrix}. \tag{3.11}$$

Similarly we can derive the coarser to finer grid correction for the first timestep of the finer block (i.e., $M_{cf}^{(1)}$) as specified in (3.12).

$$\begin{bmatrix} \bar{k}_{c,1}^{(1)} \\ \bar{k}_{c,2}^{(1)} \\ \vdots \\ \bar{k}_{c,p}^{(1)} \end{bmatrix} = CP_{\Delta t_f}P_{\Delta t_c}^{-1}C^{-1} \begin{bmatrix} k_{c,1} \\ k_{c,2} \\ \vdots \\ k_{c,p} \end{bmatrix} = M_{cf}^{(1)} \begin{bmatrix} k_{c,1} \\ k_{c,2} \\ \vdots \\ k_{c,p} \end{bmatrix}. \tag{3.12}$$

Since the correction matrices are lower triangular, a synchronization procedure can be carried out by the blocks b_f, b_c simultaneously for each timestepping stage computation. Also note that the $M_{cf}^{(1)} = {M_{fc}}^{-1}$ as expected. To derive the synchronization operator for the second timestep of b_f, the Taylor expansion is used at the time $t_{n+1/2} = t_n + \Delta t_f$ for the vector $\{\partial_t^i u\}_{i=1}^p$. Specifically we can write the timestepping stages computed from time $t_{n+1/2}$ as an approximation given in (3.13).

$$\begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_p \end{bmatrix}_{t=t_n+\Delta t_f} \approx CP_{\Delta t_f} B_{\Delta t_f} \begin{bmatrix} \partial_t u \\ \partial_t^2 u \\ \vdots \\ \partial_t^p u \end{bmatrix}_{t=t_n} . \tag{3.13}$$

The matrix $B_\Delta t_f$ is an upper triangular matrix form that consists of the Taylor series expansion coefficients. The above can be written as

$$B_{\Delta t_f} = \begin{bmatrix} 1 & b_{12}\Delta t_f & b_{13}\Delta t_f^2 & \cdots & b_{1p}\Delta t_f^{p-1} \\ 0 & 1 & b_{23}\Delta t_f & \cdots & b_{2p}\Delta t_f^{p-2} \\ 0 & 0 & 1 & \cdots & b_{3p}\Delta t_f^{p-3} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{3.14}$$

where $b_{ij} = \frac{1}{(j-i)!}$. Using (3.7) and (3.13) we can compute the timestepping stages vector $\{\bar{k}_{c,i}^{(2)}\}_{i=}^p$ from the known coarser block stages vector $\{k_{c,i}\}_{i=}^p$ that will be used to enable the second timestep for the finer block b_f. Therefore the correction operator $M_{cf}^{(2)}$ can be written as

$$\begin{bmatrix} \bar{k}_{c,1}^{(2)} \\ \bar{k}_{c,2}^{(2)} \\ \vdots \\ \bar{k}_{c,p}^{(2)} \end{bmatrix} = CP_{\Delta t_f} B_{\Delta t_f} P_{\Delta t_c}^{-1} C^{-1} \begin{bmatrix} k_{c,1} \\ k_{c,2} \\ \vdots \\ k_{c,p} \end{bmatrix} = M_{cf}^{(2)} \begin{bmatrix} k_{c,1} \\ k_{c,2} \\ \vdots \\ k_{c,p} \end{bmatrix}. \tag{3.15}$$

For linear problems, the derived synchronization operators have the same order of accuracy as the specified, explicit timestepper. However, for nonlinear problems, the derivation uses the piecewise linear approximation of the $F(u)$ computation and hence have an overall accuracy of $\mathcal{O}(\Delta t^2)$. More details on these operators' error and stability analysis can be found in [41, 40].

Synchronization between blocks. The above coarser and finer grid correction operators are computed once and stored during the LTS. In the GTS approach all the blocks in the octree are evolved at every timestep using a fixed timestep size. In contrast to GTS, the LTS approach has a block level loop that evolves only a subset of blocks. Let B_l be the subset of the total blocks B that are being evolved at block level iteration l. For example, blocks at the coarsest level L will get evolved only once, blocks at the next coarsest level (i.e., $L-1$) get evolved two times, and blocks at level $l < L$ get evolved 2^{L-l} times (see Figure 11). During these block evolutions appropriate grid correction operators (i.e., $M_{fc}, M_{cf}^{(1)}$ and $M_{cf}^{(2)}$) are applied to synchronize the solution between refinement boundaries (see Algorithm 3.3). Notice that during block level loop iterations, the evolved solution u is not synchronized in time. Therefore, for the same spatial grid point, p can have solution u defined at two different times. To facilitate the above LTS uses *octant local nodes* representation

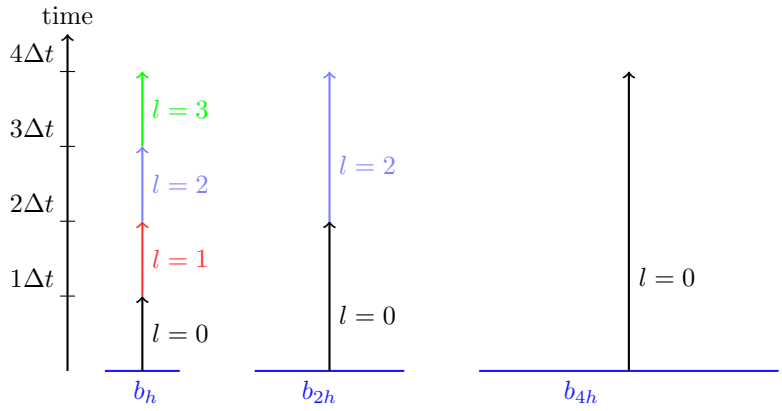


FIG. 11. *A simple illustration of block set $B = \{b_h, b_{2h}, b_{4h}\}$ that is evolving at each level l of the time level loop. Assume in the beginning that all the blocks are synchronized in time. At level $l = 0$, $B_0 = \{b_h, b_{2h}, b_{4h}\}$, perform timestep of its corresponding timestep size $\{\Delta t, 2\Delta t, 4\Delta t\}$. Note now that block b_{4h} has already reached the coarsest time over B. Similarly, for $l = 1$, $B_1 = \{b_h\}$, for $l = 2$, $B_2 = \{b_h, b_{2h}\}$, and for $l = 3$, $B_3 = \{b_h\}$ are evolved with corresponding timestep sizes. After B_3, all the blocks have reached the next coarsest timestep size over B. Note that appropriate time interpolation and corrections are applied across different refinement regions (see Figure 10).*

to store the solution vectors that are asynchronous in time. As mentioned, for block level iteration l, block set $B_l \subset B$ is considered in the evolution. The above can be used in the distributed memory case to perform partial communication between the grid partitions.

Algorithm 3.3 Local timestepping

Require: U^n previous timestep, Δt, B sequence of blocks
Ensure: $U^{n+1} = U(t^n \Delta t, \cdot)$
$V \leftarrow U^n$
$u\_unzip \leftarrow unzip(U^n)$
for $l = l_{max}$ to l_{min} **do**
  $B_l \leftarrow compute\_blk\_subset(B, l)$
  for $b \in B_l$ **do**
    for $s = 1 : k$ **do**
      $k[b, s] \leftarrow compute\_stage(u\_unzip, k[1, \ldots, s-1])$
      $k[s] \leftarrow blk\_sync\_local(k[s], B_l)$
    $V \leftarrow compute\_step\_partial(V, k, B_l)$
$U(t^n + \Delta t_{coarsest}, \cdot) \leftarrow V$

Weighted partitioning. We use the SFC based partitioning scheme to distribute work among the processors. For each local partition τ_k, the amount of work that each block has to perform to reach the global coarsest timestep depends on the block refinement level. For example, a block of level l has to perform $2\times$ timesteps compared to its adjacent coarser block at level $l - 1$. A partitioning scheme with equal weights assigned to each octant will result in load imbalanced partitions for LTS timestepping. To overcome the above, we perform weighted partitioning, where the relative weight of the block increases with the refinement level (see Figure 12). We modified our SFC-based partitioning scheme to account for the specified weights of the blocks. A 3D SFC curve can be considered an injective mapping between the 1D domain and the 3D octree domain. Using SFC ordering, we can sort the octants,

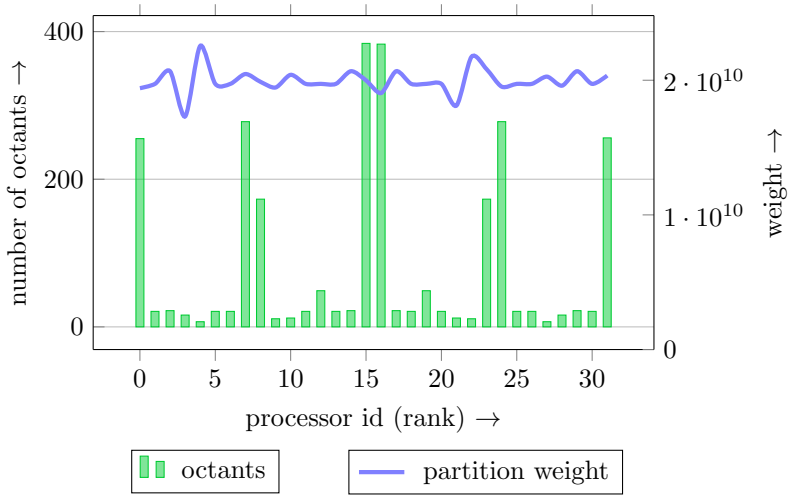


FIG. 12. *A simple illustration of our weighted SFC-based partitioning scheme for an adaptive octree partitioned across* 32 *processors. The bar plot shows the number of octants each partition has, and the line plot represents the total weight of each partition. Note that partitions with low octant counts have highly refined regions; hence they need to perform a larger number of timesteps to reach the coarser level timestep and vice versa, but the total weight of each partition is roughly equal, i.e., all partitions perform roughly the same amount of work.*

resulting in a linear ordering of the octants while ensuring spatial locality. Once ordered according to the SFC, partitioning the domain reduces to partitioning a 1D curve. In weighted SFC-partitioning, we use weighted length of the curve, i.e., we aim for equal $\sum_{e\in\tau_k} w_e$, where w_e denotes the weight of the octant, τ_k a given partition. Note that for GTS, we use $w_e = 1\ \forall e \in \tau_k$.

3.12. GTS versus LTS: Approximating the speedup. This section presents theoretical bounds for the work performed by the GTS and LTS timestepping approaches, on 2:1 balanced octrees. Let $L = (l_{max} - l_{min})$ be the difference between the maximum and minimum refinement levels for a given octree. Let $W = \{\alpha_0, \ldots, \alpha_L\}$ be the corresponding work for a block level loop, where B_l denotes the active blocks for the lth $\in \{0, 1, \ldots, L-1, L\}$ loop iteration. The work performed by the GTS and LTS schemes can be written as (3.17) and (3.16).

$$W_{lts} = \sum_{l=0}^{L} 2^{L-l}\alpha_l, \tag{3.16}$$

$$W_{gts} = 2^L \sum_{l=0}^{L} \alpha_l. \tag{3.17}$$

Assuming constant time to perform a work unit, for speedup S, $1/S$ can be written as (3.18). Since, $\frac{\alpha_0}{|W|} < \frac{1}{S}$, the maximum theoretical speedup that can be achieved for a given block distribution can be written as $S < \frac{|W|}{\alpha_0}$.

$$\frac{W_{lts}}{W_{gts}} = \frac{1}{|W|}\left(\alpha_0 + \frac{\alpha_1}{2} + \cdots + \frac{\alpha_L}{2^L}\right) < 1, \text{ where} \tag{3.18}$$

$$|W| = \sum_{l=0}^{L} \alpha_l. \tag{3.19}$$

3.13. LTS with external library packages. The proposed LTS approach uses the available spatial adaptivity to enable temporal adaptivity and is not limited to octree grids. The LTS approach can be extended in a mesh agnostic way to the underlying meshing library. To enable LTS on a given grid structure, the underlying meshing library needs to provide the following neighborhood mappings (see Figures 4 and 5).

- **Element to element map**: For a given element in the grid, what are its neighboring elements?
- **Element to shared node map**: For a given element in the grid, what are its *element shared* indices?
- **Element to local node map**: For a given element in the grid, what are its *element local* indices? The above will allow storing solution vectors that are not synchronized in time. The mapping between *element local* to *element shared* is provided by appropriate interpolations (i.e., in both space and time).

Provided these additional data structures, the proposed LTS approach can be easily extended to other mesh libraries such as p4est [17] and AMReX [59].

4. Results. This section presents an overall evaluation of the proposed LTS methodology in terms of (1) parallel scalability, (2) accuracy of the method for linear and nonlinear problems, and (3) performance evaluation of the octree grid generation and balancing.

4.1. Experimental setup. The large scalability experiments reported in this paper were performed on TACC's Frontera supercomputer. Frontera is an Intel supercomputer with a total of 8,008 nodes, each consisting of a Xeon Platinum 8280 ("Cascade Lake") processor, with a total of 448,448 cores. Each node has 192GB of memory. The interconnect is based on Mellanox HDR technology with full HDR (200 Gb/s) connectivity between the switches and HDR100 (100 Gb/s) connectivity to the compute nodes.

4.2. Nonlinear and linear wave propagation. In this section, we introduce a simple model to demonstrate LTS, the classical wave equation. We write the classical wave equation in a form with first derivatives in time and second derivatives in space. This allows us to easily apply the specified, explicit schemes.

For a scalar function $\chi(t, x^i)$, the classical wave equation in Cartesian coordinates (t, x, y, z) with a nonlinear source term can be written as

$$\frac{\partial^2 \chi}{\partial t^2} - \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \chi = -c \frac{\sin(2\chi)}{r^2}, \tag{4.1}$$

where $r = \sqrt{x^2 + y^2 + z^2}$. We write the equation as a first-order in time system by introducing the variable ϕ as

$$\frac{\partial \chi}{\partial t} = \phi, \tag{4.2}$$

$$\frac{\partial \phi}{\partial t} = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \chi - c \frac{\sin(2\chi)}{r^2}. \tag{4.3}$$

For nonlinear wave propagation results presented in this paper, we used $c = 1$. We choose outgoing radiative boundary conditions for this system [4]. We assume that

the variables χ and ϕ approach the form of spherical waves as $r \to \infty$, which decay as $1/r^k$. The radiative boundary conditions then have the form

$$\frac{\partial f}{\partial t} = \frac{1}{r}\left(x\frac{\partial f}{\partial x} + y\frac{\partial f}{\partial y} + z\frac{\partial f}{\partial z}\right) - k(f - f_0), \tag{4.4}$$

where f represents the functions χ and ϕ, and f_0 is an asymptotic value. We assume $k = 1$ for χ and $k = 2$ for ϕ.

For the linear wave propagation results presented, we simply zero out the nonlinear source term (i.e., set $c = 0$). The analytical solution for the 1D linear wave operator in (4.1) with zero source term can be written as

$$\chi(t, x) = \frac{f(x - t) + f(x + t)}{2}, \text{ where } \chi(0, x) = f(x). \tag{4.5}$$

4.3. Accuracy. We conduct numerical experiments using linear and nonlinear wave propagation to test the accuracy of our methods and implementation (see Figure 13). We compute the analytical solution for wave propagation in the x direction for linear wave propagation and compare the analytical solution with the computed solution using global and local timestepping. For the above experiments, we used a third-order RK scheme with increasing spacetime resolution. Figure 14 shows the numerical error for LTS and GTS approaches with increasing resolution.

Since the computation of the analytical solution for the 1D wave operator with a nonlinear source term is complicated, we compare the l_∞ norm computed on the numerical difference between global and local evolved timesteps. In Table 1 we present the difference between the solution χ evolved using GTS and LTS for increasing maximum refinement level (MAXDEPTH) 8 and 10. Again, both GTS and LTS are in agreement close to machine precision.

4.4. LTS efficiency and space adaptivity. As mentioned in section 3.12, we can approximate the speedup S between LTS and GTS for a given octant distribution. Since we can end up with meshes where the use of LTS will not provide significant advantages over GTS, we can selectively use LTS based on the expected speedup.

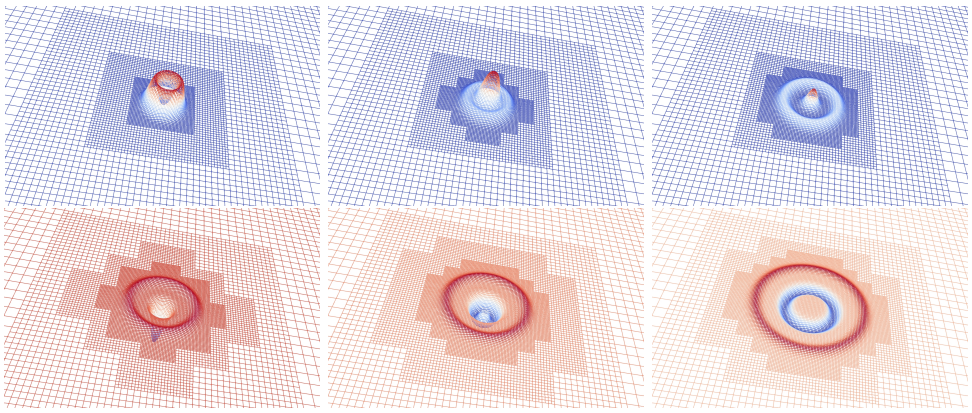


FIG. 13. *Plots for linear wave propagation with LTS with a velocity vector* $(1, 1, 1)$ *using a Gaussian pulse centered at* $(0, 0, 0)$ *as the initial condition. Images shown from left to right and top to bottom in increasing order of simulation time.*

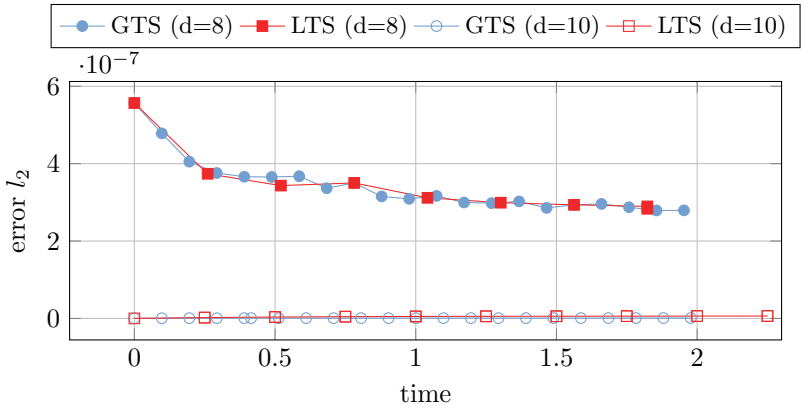


FIG. 14. *Discrete l_2 error compared to the analytical solution for the 1D wave operator in three dimensions for GTS and LTS timestepping. For this experiment we used* MAXDEPTH *of* 8 *and* 10 *with refinement trigger tolerance of* 10^{-5}.

TABLE 1

l_2 difference between GTS and LTS at corresponding timesteps for nonlinear wave propagation. For this experiment we used MAXDEPTH *of* 8 *and* 10 *with refinement trigger tolerance of* 10^{-5}. *Note that for* MAXDEPTH 8, 8 *global timesteps and for* MAXDEPTH 10, 32 *global timesteps were equivalent for a single* LTS *step.*

Time (s)	$\|\|\chi_{lts} - \chi_{gts}\|\|_2$	
	MAXDEPTH=8	MAXDEPTH=10
0.00	0	0
0.42	2.44E-11	5.99E-13
0.83	4.59E-11	1.85E-12
1.25	6.43E-11	3.31E-12
1.67	7.97E-11	4.70E-12
2.08	9.30E-11	5.84E-12
2.50	1.05E-10	6.68E-12
2.92	1.15E-10	7.23E-12
3.33	1.25E-10	7.58E-12
3.75	1.34E-10	7.86E-12
4.17	1.42E-10	8.17E-12
4.58	1.48E-10	8.56E-12
5.00	1.54E-10	9.03E-12

To evaluate our speedup model (3.18), and to assess the overhead of applying the LTS correction operators, we computed the actual speedup reported for the linear wave propagation with increasing MAXDEPTH. The estimated and reported speedup values are presented in Table 2, and weak scalability study on the above problem is presented in Figure 15. As can be seen, the estimated speedup values are sufficiently close to the predicted ones, allowing applications to determine when it is beneficial to use LTS.

Adaptivity in spacetime can be vital for computational applications, especially when spacetime adaptivity is necessary to make these simulations feasible even on leadership architectures. Here we present estimated speedup by using LTS, for the simulation of binary black hole mergers and the computation of the resulting gravitational waves [1, 3, 35, 26]. These simulations' computational cost increases signifi-

TABLE 2
The estimated versus reported speedup for LTS over GTS for linear wave propagation with increasing MAXDEPTH *for adaptive octrees.*

d	l_{min}	l_{max}	GTS (s)	LTS (s)	Est. speedup	Reported speedup
9	2	7	2.91	0.88	3.31	3.30
10	3	8	26.69	9.26	3.31	2.87
11	2	9	114.74	36.76	3.49	3.12
12	2	10	238.05	73.94	3.56	3.21

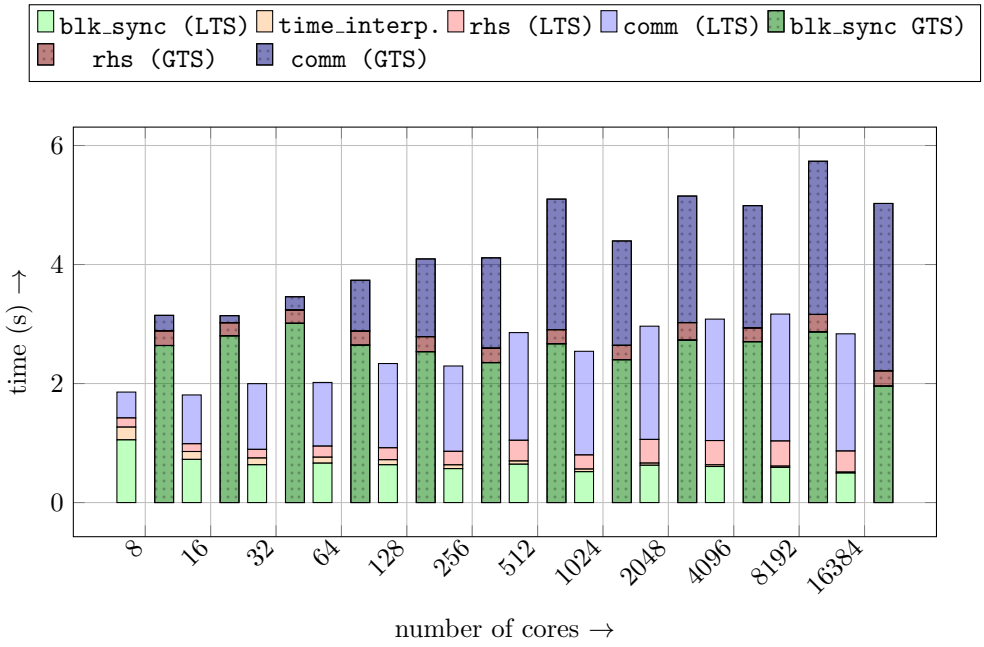


FIG. 15. *Weak scaling results on TACC's Frontera for RK3 timestepping using LTS (left) and GTS (right) approaches. For this experiment, the maximum and minimum refinement levels are* $l_{max} = 8$ *and* $l_{min} = 3$*, hence* $\Delta L = 5$*. Therefore a single LTS step is equivalent to* $2^{\Delta L} = 32$ *global timesteps. For LTS, the plot shows the overall cost breakdown between block synchronization (*`blk_sync`*), applying time interpolations between blocks (*`LTS_interp.`*), computing the RHS (*`rhs`*) and communication costs (*`comm`*). For GTS, we show the cost breakdown between* `blk_sync`*,* `rhs`*, and* `comm`*. Note that for GTS time interpolations are not required. Note the significant difference of* `blk_sync` *cost between LTS and GTS. For GTS* `blk_sync` *is a global operation, while in LTS* `blk_sync` *is a local operation, where synchronization is performed only on the subset of blocks, which are currently being evolved. These weak scaling results were performed using a grain size of* $\sim 100K$ *unknowns per core, with the number of cores ranging from* 8 *to* 16,384 *cores. The largest problem recorded had* 1.6×10^9 *unknowns. The above results are generated for radial wave propagation with a* MAXDEPTH 10 *and a refinement tolerance of* 10^{-5}*.*

cantly when the mass ratio q of the two black holes increases. Assuming we need n grid points in one dimension to capture the larger black hole, to capture the smaller black hole in the presence of the larger black hole, we need qn in one dimension, hence, an increase of q^3 points in three dimensions. The above makes the large mass ratio gravitational wave simulations infeasible at the time. We use these large mass ratio binary black hole grids (see Figure 16) to estimate the speedup that the time adaptivity can enable (see Table 3). For mass ratios of 10, one can expect up to 70× speedup,

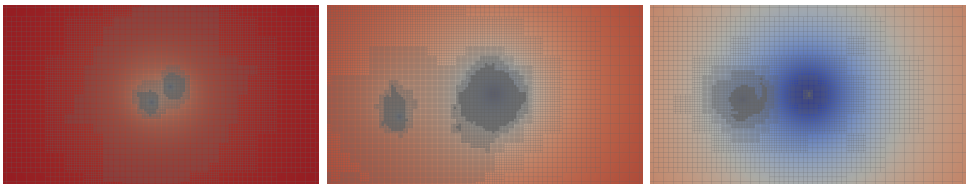


FIG. 16. *Example octree grids generated for black hole binaries with mass ratio* 1,10,100.

TABLE 3
Estimated speedup for binary black hole grid with increasing mass ratios from 1 *to* 10.

Mass ratio	l_{min}	l_{max}	Est. speedup
1	3	15	9.82
2	3	16	18.105
5	3	18	15.035
10	3	22	71.8302

which is a significant reduction in the cost (runtime and energy) of estimating the gravitational waves and can reduce the burden on supercomputing resources.

4.5. Weak and strong scaling. Parallel scalability of timesteppers is essential when dealing with large-scale simulations. This section presents weak and strong scalability results for the linear wave propagation problem on octree meshes using global and local timestepping. For both weak and strong scaling, we used 7^3 grid points per octant. For weak scaling, we set the computational domain to $[-10, 10]^3$ and used MAXDEPTH 10. The MAXDEPTH 10 grid generated $l_{min} = 3$ and $l_{max} = 8$, hence 32 global timesteps is equivalent to a single LTS timestep. Therefore, in order to make the GTS and LTS results comparable, in the following scaling results, we present timing for 32 steps in GTS and 1 step (32 partial steps) in LTS. In weak scaling, we increase the grid size, keeping the degrees of freedom per core roughly constant (100K unknowns per core). Weak scaling results for LTS and GTS are presented in Figure 15. Each bar presents the corresponding evolution time between LTS and GTS. For each scheme, we present the overall cost breakdown between computation of the RHS (`rhs`) of the PDE, and block padding synchronization (`blk_sync`) and interprocess communication (*comm*). The `blk_sync` cost consists of space interpolation, which is common for both LTS and GTS schemes due to space adaptivity. We present an extra fraction of time interpolation cost between blocks only present in the LTS. For GTS the `blk_sync` operation is a global synchronization, i.e., all the blocks are evolved and need to synchronize the padding regions for the next `rhs` computation. In contrast, for the LTS scheme the `blk_sync` operation is a local (partial) synchronization limited to the blocks evolved at the current partial step followed by time interpolation to correct the padding regions between blocks. The weak scaling plot shows that the partial synchronization with appropriate time interpolation is more efficient than the global synchronization in the GTS scheme.

To perform strong scaling (see Figure 17), we used MAXDEPTH 12 and recorded l_{min} and l_{max} were 3 and 10, respectively. Therefore, 128 global timesteps are equivalent to a single LTS timestep. We keep the total grid size fixed at $262M$ unknowns for strong scaling tests and increase the number of cores from 64 to 8192. The strong scaling plot shows the same cost breakdown described above. The recorded average parallel efficiencies between LTS and GTS schemes were 87% and 74%, respectively.

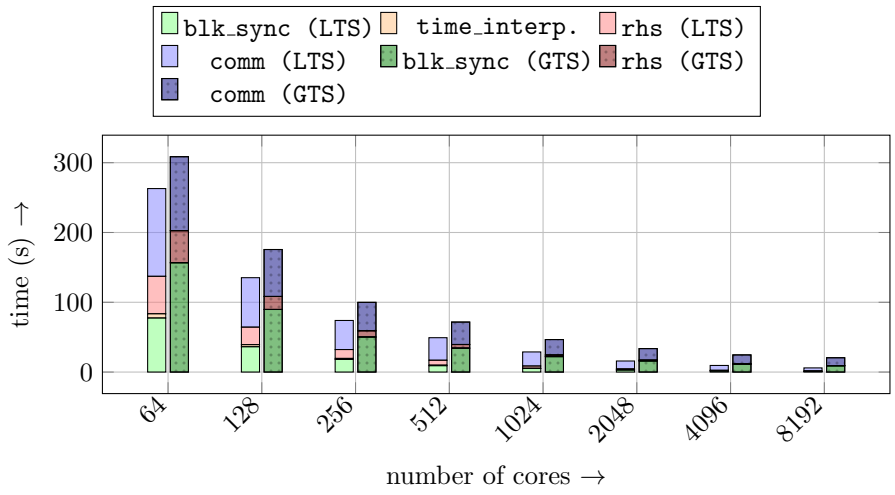


FIG. 17. *Strong scaling results on TACC's Frontera for LTS (left) and GTS (right) timestepping using the RK3 explicit scheme. For this experiment, the refinement levels are* $l_{max} = 10$ *and* $l_{min} = 3$*, hence* $\Delta L = 7$*. Therefore a single LTS is equivalent to* $2^{\Delta L} = 128$ *global timesteps. For LTS, the plot shows the overall cost breakdown between block synchronization (*`blk_sync`*), applying time interpolations between blocks (*`LTS_interp.`*), computing the RHS (*`rhs`*) and the communication costs (*`comm`*). For GTS, we show the cost breakdown between* `blk_sync`*,* `rhs`*, and* `comm` *(time interpolations are not required for GTS). A significant difference of* `blk_sync` *cost between LTS and GTS. For GTS* `blk_sync` *is a global operation, while in LTS* `blk_sync` *is a local operation, where synchronization is performed only on the subset of blocks, which are currently being evolved. Presented strong scaling study performed using a fixed problem size of* $262M$ *unknowns where the number of cores ranges from* 64 *to* 8192 *cores. Note that for strong scaling results remeshing is disabled in order to keep the problem size fixed and unchanged during evolution.*

The low overhead of `blk_sync` operation allows LTS to demonstrate superior weak and strong scalability compared to the GTS scheme.

4.6. Weighted partitioning and mesh generation. The performance of data partitioning and mesh generation is crucial for AMR applications, especially when the computational grid changes frequently. We refer to this process as re-meshing, which requires repartitioning the data (since the refinement change may have caused load imbalance), enforcing 2:1 balancing, and mesh data structure generation. The performance of the remeshing is crucial, but it is not the main focus of this paper. In the current implementation, we trigger refinement in LTS when all the blocks are synchronized in time. Figure 13 shows how the grid changes as the wave propagates radially outward. Figures 18 and 19 show the weak and strong scalability of the operations related to remeshing. The above experiments show that mesh generation has a relatively high computational cost, compared to SFC weighted partitioning and 2:1 balancing of octrees. This is mainly because mesh generation performs a large number of search operations on the octree to build the neighborhood data structures, which are essential to performing numerical computations.

5. Conclusions. This paper presented methods to enable time adaptivity for solving PDEs numerically on spatially adaptive grids. We presented experimental results for the accuracy and scalability of the presented approaches. We show that time adaptivity for highly adaptive octrees with high refinement levels can be essential to reduce overall time to solution. As future work, we are hoping to explore (1)

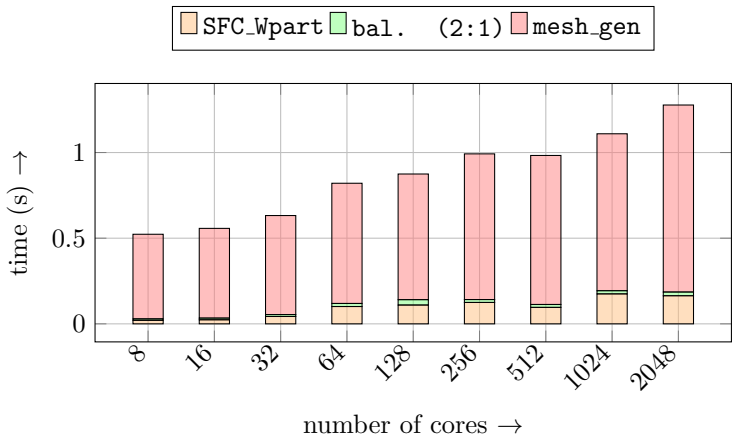


FIG. 18. *Weak scaling results in TACC's Frontera to perform SFC-based weighted partitioning (`SFC_Wpart`) for Gaussian octant distribution centered at* $(0, 0, 0)$ *followed by* 2:1 *balancing (`bal.2:1`) of the octants, which is used as the input for the mesh generation (`mesh_gen`). For this experiment, we used* $1.6M$ *grid points per core, using* 7^3 *points per octant, with the number of cores varying from* 8 *to* 2048*. The largest problem reported had a total of* $3.3B$ *grid points, where the mesh generation completed under* $2s$*.*

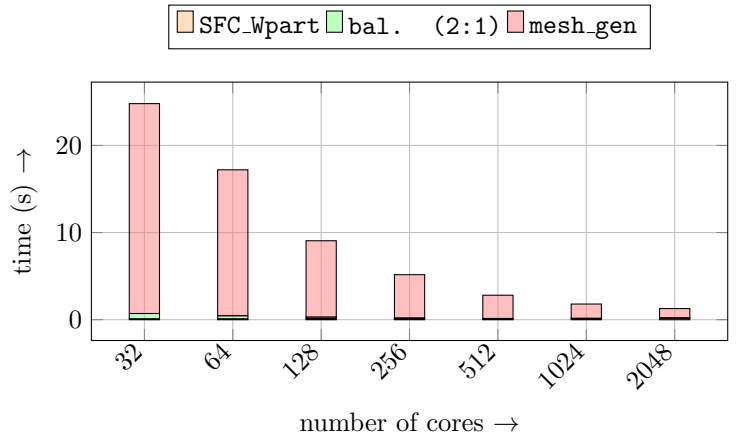


FIG. 19. *Strong scaling results in TACC's Frontera to perform SFC-based weighted partitioning (`SFC_Wpart`) for Gaussian octant distribution centered at* $(0, 0, 0)$ *followed by* 2:1 *balancing (`bal.2:1`) of the octants, which is used as the input for the mesh generation (`mesh_gen`). For the depicted strong scaling, we keep the problem size fixed at* $3.3B$ *grid points with the number of cores increasing from* 32 *to* 2048*.*

LTS approaches for implicit timesteppers, (2) scalable LTS in heterogeneous clusters with GPUs, and (3) efficient localized grid update (remesh) that is well suited for LTS approaches.

Appendix A. Code description and evaluation.

A.1. Experiments. Experiments performed in Frontera are executed in the following module environment.

```
Currently Loaded Modulefiles:
```

```
1) intel/19.0.5        4) python3/3.7.0
2) impi/19.0.5         5) autotools/1.2
3) petsc/3.12          6) cmake/3.16.1
```

A.1.1. NLSigma. The source code used in the paper is open source and available at https://github.com/paralab/Dendro-5.01. All the experiments presented in the paper use the main source files in the `NLSigma` folder. To build the code,

```
mkdir build
cd build
ccmake ../.
make nlsmSolverNUTS -j4
```

More details on the required packages and how to build the code can be found at the repository main page. To enable internal profile counters (for timing the code), `ENABLE_DENDRO_PROFILE_COUNTERS` on, and to switch between linear and nonlinear wave propagation, `NLSM_NONLINEAR` off and on correspondingly. `nlsmSolverNUTS` takes two arguments; the first argument is the path to the parameter file, and the second argument is the timestepping mode, which is an integer value,

- 0: Perform LTS timestepping.
- 1: Perform GTS timestepping.
- 2: Perform single LTS step and required steps by the uniform timestepping, each the same time.
- 3: Perform 2 until we reach the simulation end time specified on the parameter file.

A.1.2. NLSM convergence. The configuration file for convergence tests is provided under the repository folder `NLSigma/par/nlsm_d<8,10>.par.json`. The detailed description for each parameter in the file is provided as a comment in the parameter file. In order to run the cases, please use

```
ibrun -np <mpi tasks> ./NLSigma/nlsmSolverNUTS  <par file> <ts_mode>
```

Acknowledgments. This work used computing resources of the Extreme Science and Engineering Discovery Environment (XSEDE) allocation TG-PHY180054 and Frontera pathway allocation PHY20033.

REFERENCES

[1] *Einstein at Home*, https://einstein.phys.uwm.edu.
[2] *FLASH Home Page*, https://flash.uchicago.edu/website/home/, 2007.
[3] A. Abramovici et. al., *LIGO: The Laser Interferometer Gravitational-wave Observatory*, Science, 256 (1992), pp. 325–333.
[4] M. Alcubierre, *Introduction to* $3+1$ *Numerical Relativity*, Internat. Ser. Monogr. Phys., Oxford University Press, Oxford, UK, 2008.
[5] M. Almquist and M. Mehlin, *Multilevel local time-stepping methods of Runge–Kutta-type for wave equations*, SIAM J. Sci. Comput., 39 (2017), pp. A2020–A2048, https://doi.org/10.1137/16M1084407.
[6] *Chombo—Infrastructure for Adaptive Mesh Refinement*, Lawrence Berkeley National Laboratory Applied Numerical Algorithms Group, https://seesar.lbl.gov/anag/chombo/, 2006.
[7] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.*II*—A general-purpose object-oriented finite element library*, ACM Trans. Math. Software, 33 (2007).
[8] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu, *Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers*, Comput. Methods Appl. Mech. Engrg., 152 (1998), pp. 85–102.
[9] P. Bastian, *Load balancing for adaptive multigrid methods*, SIAM J. Sci. Comput., 19 (1998), pp. 1303–1321.

[10] J. Bédorf, E. Gaburov, and S. Portegies Zwart, *Bonsai: A GPU tree-code*, in Advances in Computational Astrophysics: Methods, Tools, and Outcome, R. Capuzzo-Dolcetta, M. Limongi, and A. Tornambè, eds., Astronomical Society of the Pacific Conference Series 453, 2012, p. 325.

[11] M. A. Bender, *Compute process allocator (CPA)*, Urbana, 51 (2006), 61801.

[12] M. J. Berger, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, Tech. report, Department of Computer Science, Stanford University, 1982.

[13] M. J. Berger, *Stability of interfaces with mesh refinement*, Math. Comput., 45 (1985), pp. 301–318.

[14] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512, https://doi.org/10.1016/0021-9991(84)90073-1.

[15] M. Bern, D. Eppstein, and S.-H. Teng, *Parallel construction of quadtrees and quality triangulations*, Internat. J. Comput. Geom. Appl., 9 (1999), pp. 517–532.

[16] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J. hoon Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, and Y. Li, *ENZO: An adaptive mesh refinement code for astrophysics*, Astrophys. J. Suppl. Ser., 211 (2014), 19, https://doi.org/10.1088/0067-0049/211/2/19.

[17] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM J. Sci. Comput., 33 (2011), pp. 1103–1133, https://doi.org/10.1137/100791634.

[18] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dynamic Octree Load Balancing Using Space-Filling Curves*, Technical Report CS-03, Williams College, Department of Computer Science, 2003.

[19] J. Carrier, L. Greengard, and V. Rokhlin, *A fast adaptive multipole algorithm for particle simulations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 669–686.

[20] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in Proceedings of the 21st International Parallel and Distributed Processing Symposium, IEEE, 2007.

[21] F. Collino, T. Fouquet, and P. Joly, *Conservative space-time mesh refinement methods for the FDTD solution of Maxwell's equations*, J. Comput. Phys., 211 (2006), pp. 9–35.

[22] R. Courant, K. Friedrichs, and H. Lewy, *On the partial difference equations of mathematical physics*, IBM J. Research Development, 11 (1967), pp. 215–234, https://doi.org/10.1147/rd.112.0215.

[23] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, *New challenges in dynamic load balancing*, Appl. Numer. Math., 52 (2005), pp. 133–152.

[24] Einstein Toolkit, https://einsteintoolkit.org.

[25] M. Fernando, D. Duplyakin, and H. Sundar, *Machine and application aware partitioning for adaptive mesh refinement applications*, in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, New York, 2017, ACM, pp. 231–242, https://doi.org/10.1145/3078597.3078610.

[26] M. Fernando, D. Neilsen, H. Lim, E. Hirschmann, and H. Sundar, *Massively parallel simulations of binary black hole intermediate-mass-ratio inspirals*, SIAM J. Sci. Comput., 41 (2019), pp. C97–C138, https://doi.org/10.1137/18M1196972.

[27] M. S. Fernando and H. Sundar, *paralab/Dendro-5.01: Local Timestepping on Octree Grids*, https://doi.org/10.5281/zenodo.3879315, 2020.

[28] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Parallel structures and dynamic load balancing for adaptive finite element computation*, Appl. Numer. Math., 26 (1998), pp. 241–263.

[29] D. Fuster, A. Bagué, T. Boeck, L. Le Moyne, A. Leboissetier, S. Popinet, P. Ray, R. Scardovelli, and S. Zaleski, *Simulation of primary atomization with an octree adaptive mesh refinement and vof method*, Int. J. Multiph. Flow, 35 (2009), pp. 550–565.

[30] M. J. Gander and L. Halpern, *Techniques for locally adaptive time stepping developed over the last two decades*, in Domain Decomposition Methods in Science and Engineering XX, Springer, New York, 2013, pp. 377–385.

[31] C. Gear and D. Wells, *Multirate linear multistep methods*, BIT, 24 (1984), pp. 484–502.

[32] S. Gottlieb and C.-W. Shu, *Total variation diminishing runge-kutta schemes*, Math. Comput., 67 (1998), pp. 73–85.

[33] R. Grauer, C. Marliani, and K. Germaschewski, *Adaptive mesh refinement for singular solutions of the incompressible euler equations*, Phys. Rev. Lett., 80 (1998), 4177.

[34] M. J. Grote, M. Mehlin, and T. Mitkova, *Runge–Kutta-based explicit local time-stepping methods for wave propagation*, SIAM J. Sci. Comput., 37 (2015), pp. A747–A775, https://doi.org/10.1137/140958293.
[35] *HAD Home Page*, had.liu.edu, 2009.
[36] T. Isaac, C. Burstedde, and O. Ghattas, *Low-cost parallel algorithms for* 2:1 *octree balance*, in Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium, 2012, pp. 426–437, https://doi.org/10.1109/IPDPS.2012.47.
[37] M. Ishii, M. Fernando, K. Saurabh, B. Khara, B. Ganapathysubramanian, and H. Sundar, *Solving PDEs in space-time:* 4*D tree-based adaptivity, mesh-free and matrix-free approaches*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–61.
[38] P. Joly and J. Rodriguez, *An error analysis of conservative space-time mesh refinement methods for the one-dimensional wave equation*, SIAM J. Numer. Anal., 43 (2005), pp. 825–859.
[39] A. Kanevsky, M. H. Carpenter, D. Gottlieb, and J. S. Hesthaven, *Application of implicit–explicit high order Runge–Kutta methods to discontinuous-Galerkin schemes*, J. Comput. Phys., 225 (2007), pp. 1753–1781.
[40] L. Liu, X. Li, and F. Q. Hu, *Nonuniform time-step Runge–Kutta discontinuous Galerkin method for computational aeroacoustics*, J. Comput. Phys., 229 (2010), pp. 6874–6897.
[41] L. Liu, X. Li, and F. Q. Hu, *Nonuniform-time-step explicit Runge–Kutta scheme for high-order finite difference method*, Comput. & Fluids, 105 (2014), pp. 166–178.
[42] G. Marcus and M. Teodora, *High-order explicit local time-stepping methods for damped wave equations*, J. Comput. Appl. Math., 239 (2013), pp. 270–289, https://doi.org/10.1016/j.cam.2012.09.046.
[43] K. D. Nikitin, M. A. Olshanskii, K. M. Terekhov, and Y. V. Vassilevski, *A numerical method for the simulation of free surface flows of viscoplastic fluid in* 3*D*, J. Comput. Math., (2011), pp. 605–622.
[44] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1992.
[45] J. R. Rice, *Split Runge-Kutta method for simultaneous*, J. Res. National Bureau of Standards Math. Math. Phys. B, 64 (1960), p. 151.
[46] M. Rietmann, D. Peter, O. Schenk, B. Uçar, and M. Grote, *Load-balanced local time stepping for large-scale wave propagation*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 925–935.
[47] J. Rudi, A. C. I. Malossi, T. Isaac, G. Stadler, M. Gurnis, P. W. J. Staar, Y. Ineichen, C. Bekas, A. Curioni, and O. Ghattas, *An extreme-scale implicit solver for complex pdes: Highly heterogeneous flow in earth's mantle*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, 2015, ACM, pp. 5:1–5:12, https://doi.org/10.1145/2807591.2807675.
[48] R. Sampath, H. Sundar, S. S. Adavani, I. Lashuk, and G. Biros, `Dendro` *Home Page*, https://www.seas.upenn.edu/csela/dendro, 2008.
[49] A. Sandu and E. M. Constantinescu, *Multirate explicit adams methods for time integration of conservation laws*, J. Sci. Comput., 38 (2009), pp. 229–249.
[50] M. Schreiber, T. Weinzierl, and H.-J. Bungartz, *SFC-based communication metadata encoding for adaptive mesh*, in Proceedings of PARCO, 2013.
[51] M. Schreibera, T. Weinzierla, and H.-J. Bungartza, *SFC-based communication metadata encoding for adaptive mesh refinement*, Parallel Computing Accelerating Computational Science Engineering, 25 (2014), p. 233.
[52] L. Sliwko and V. Getov, *Workload schedulers-genesis, algorithms and comparisons*, Internat. J. Comput. Sci. Software Engrg., 4 (2015), pp. 141–155.
[53] H. Suh and T. Isaac, *Evaluation of a minimally synchronous algorithm for* 2:1 *octree balance*, in Proceedings of SC'20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 307–318.
[54] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler, *Parallel geometric-algebraic multigrid on unstructured forests of octrees*, in Proceedings of SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–11.
[55] H. Sundar, R. 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.
[56] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros, *Low-constant parallel algorithms for finite element simulations using linear octrees*, in Proceedings of

SC'07: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2007.

[57] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros, *Low-constant parallel algorithms for finite element simulations using linear octrees*, in Proceedings of the ACM/IEEE Conference on Supercomputing, New York, 2007, ACM, pp. 25:1–25:12, https://doi.org/10.1145/1362622.1362656.

[58] T. Weinzierl et al., *Peano—A Framework for PDE Solvers on Spacetree Grids*, 201x, http://www.peano-framework.org.

[59] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, et al., *AMReX: A framework for block-structured adaptive mesh refinement*, J. Open Source Software, 4 (2019), pp. 1370–1370.