Performance Analysis of Speculative Parallel Adaptive Local Timestepping for Conservation Laws

MAXIMILIAN BREMER, JOHN BACHAN, and CY CHAN, Lawrence Berkeley National Laboratory
CLINT DAWSON, The University of Texas at Austin

Stable simulation of conservation laws, such as those used to model fluid dynamics and plasma physics applications, requires the satisfaction of the so-called Courant-Friedrichs-Lewy condition. By allowing regions of the mesh to advance with different timesteps that locally satisfy this stability constraint, significant work reduction can be attained when compared to a time integration scheme using a single timestep size. However, parallelizing this algorithm presents considerable difficulty. Since the stability condition depends on the state of the system, dependencies become dynamic and potentially non-local. In this article, we present an adaptive local timestepping algorithm using an optimistic (Timewarp-based) parallel discrete event simulation. We introduce waiting heuristics to limit misspeculation and a semi-static load balancing scheme to eliminate load imbalance as parts of the mesh require finer or coarser timesteps. Last, we outline an interface for separating the physics of the specific conservation law from the temporal integration allowing for productive adoption of our proposed algorithm. We present a misspeculation study for three conservation laws, demonstrating both the productivity of the local timestepping API, for which 74% of the lines of code are reused across different conservation laws, and the robustness of the waiting heuristics—at most 1.5% of element updates are rolled back. Our performance studies demonstrate up to a 2.8× speedup versus a baseline unoptimized local timestepping approach, a 4x improvement in per-node throughput compared to an MPI parallelization of synchronous timestepping, and scalability up to 3,072 cores on NERSC's Cori Haswell partition.

CCS Concepts: • **Mathematics of computing** → **Partial differential equations**; • **Computing methodologies** → *Discrete-event simulation*; *Massively parallel and high-performance simulations*; • **Applied computing** → Earth and atmospheric sciences;

Additional Key Words and Phrases: Local timestepping, parallel discrete event simulation, Timewarp, shallow water equations, conservation laws

This article has been authored by an author at Lawrence Berkeley National Laboratory under contract number DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes. In addition, this work was supported by the National Science Foundation under grant NSF 1854986.
Authors' addresses: M. Bremer, J. Bachan, and C. Chan, Lawrence Berkeley National Laboratory 1 Cyclotron Rd Berkeley, CA 94720 USA; emails: {mb2010, jdbachan, cychan}@lbl.gov; C. Dawson, The University of Texas at Austin Austin, TX 78712 USA; email: clint.dawson@austin.utexas.edu.

1049-3301/2022/11-ART26
https://doi.org/10.1145/3545996

ACM Reference format:
Maximilian Bremer, John Bachan, Cy Chan, and Clint Dawson. 2022. Performance Analysis of Speculative Parallel Adaptive Local Timestepping for Conservation Laws. *ACM Trans. Model. Comput. Simul.* 32, 4, Article 26 (November 2022), 30 pages.
https://doi.org/10.1145/3545996

1 INTRODUCTION

Simulation of systems of hyperbolic conservation laws remains a core area of interest to the scientific computing community, providing insights to fields ranging from computational fluid dynamics to plasma physics. These methods describe the local exchange of conserved quantities such as mass, momentum, or energy between regions of space. The well-known **Courant-Friedrichs-Lewy (CFL)** condition has formed the basis of numerous popular explicit timestepping schemes for modeling convection-dominated flow [22, 39, 49, 53, 64]. The CFL condition for conservation laws is defined to be

$$\Delta t \leq C_{CFL} \frac{\Delta x}{|\Lambda|},$$

where Δt is the proposed timestep, Δx is the length scale, $|\Lambda|$ is the local wave speed, and C_{CFL} is a discretization-dependent constant. By bounding the allowable timestep as a function of element size and local wave speed, one can limit the difference in element-wise averages of conserved quantities between neighboring elements, preventing spurious oscillation that may amplify indefinitely, making the scheme unstable.

Traditional synchronous timestepping schemes use a single timestep that must guarantee the satisfaction of the CFL condition for all elements. In the presence of large variations of both Δx and $|\Lambda|$, high percentages of elements will update with smaller than necessary timesteps. By allowing local regions of the mesh to take different timesteps, the computational work required by the simulation can be significantly reduced.

This article is a conference extension of our previous work [9] and belongs to a larger effort to make the **Runge-Kutta discontinuous Galerkin (RKDG)** finite element method a viable alternative to current storm surge simulation codes such as ADCIRC [15, 27, 28]. The RKDG finite element method has advantageous numerical characteristics that improve the fidelity and stability of generated solutions but currently requires too much computational work to be usable in an operational setting. In a forecasting scenario, the simulation must be able to produce multiple surge forecasts within a 2-hour time budget using up to hundreds of cores after the hurricane wind forecasts are released to be useful for emergency managers [4]. Since elements only require information from face-connected neighboring elements to update, the RKDG finite element method can be parallelized to satisfy the time-to-solution constraint, but current implementations require too many cores to be competitive when running on shared resources such as university clusters. Work reduction through local timestepping reduces computational resource requirements, potentially making RKDG storm surge codes competitive for real-time forecasting.

Parallelizing **adaptive local timestepping (ALTS)** is particularly difficult when the propagation speed Λ depends on the solution itself. Proposed timestep sizes may change as updates from neighbors are processed. A reduction of an element's timestep may cause a reduction in the neighbor's timestep and a reduction of the neighbor's neighbor's timestep and so on, leading to uncertainty in the number of messages from neighbors that must be processed before a submesh can update. In a distributed computing environment, we need to incorporate possibly non-local information when determining if a region can be updated.

The aim of this work is to optimize the parallel performance of ALTS over core counts relevant for operational surge forecasting. In this article, we present results for the Burgers', Euler, and shallow water equations, demonstrating the robustness of the method for different conservation laws and initial conditions. Performance results shown here are limited to the shallow water equations, as these results most directly emulate storm surge simulation. However, with properly managed task granularity, these results should generalize to other conservation laws as well. Dawson et al. [28] examined local parallel timestepping for storm surge. However, without the capability to dynamically adjust timesteps, the implementation required frequent restarts due to instabilities caused by changing wave speeds, undermining the stability argument for using discontinuous Galerkin finite elements and limiting performance gains. Following the discrete event-based formulation of Bremer [8], we implement ALTS as a discrete event simulation and parallelize it using an optimistic parallel discrete event simulator based on Jefferson's Timewarp algorithm [44]. The contributions of this article are as follows: (1) a set of optimizations to identify and reduce misspeculation; (2) a semi-static load balancer to resolve dynamic load imbalance; (3) a proposed API to separate the physics from the timestepping logic to facilitate productive adoption of the algorithm; and (4) a detailed characterization of the performance of ALTS, including descriptions of event overheads, misspeculation, strong and weak scalability, optimization impact, and comparison versus a state-of-the-art synchronous approach. The parallel efficiency improvements achieved in this work are well suited to enable the use of the discontinuous Galerkin finite element method for operational hurricane storm surge forecasting and other large-scale flow simulations that are otherwise intractable due to required computational work.

1.1 Previous Work

The idea of allowing different subsystems to advance with different timesteps has been extensively examined for Runge-Kutta methods [38, 60]. Extending these methods to conservation laws has seen considerable work [16, 23, 26, 42, 48, 58, 61, 66]. In addition, other fields have arrived at related time integration schemes. Quantized state systems [19, 31, 47] have been used to model systems of ordinary differential equations (particularly for power electronics modeling) by approximating state values with time-dependent polynomials and updating estimates when they deviate more than a given threshold. Those systems focus on enforcing bounds on state variable deviations, as opposed to enforcing particular mathematical constraints, such as a CFL condition. Asynchronous variational integration is another locally timestepped method designed for linear elastodynamics [50]. Whereas local timestepping for conservation laws is focused on maintaining non-linear stability, asynchronous variational integration focuses on preserving the total energy of the system.

Beyond the various applications, parallelizations share common motifs based on the characterization of the algorithm. By assuming constant timesteps or variations in timestepping sizes, local timestepping methods—also known as multi-rate integrators—can be parallelized using task-graph-like approaches (e.g., [13, 43, 67]). However, the inability to account for variations in the wave speed limits the practicality of these approaches for non-linear problems. Methods such as quantized state systems and the solver in the work of Omelchenko and Karimabadi [55–57] rely on a discrete event simulation framework to account for changes in timestep and correctly resolve dependencies in a parallel context. Our ALTS implementation differs from these other discrete event simulation based approaches in that (1) ALTS strictly satisfies the total variation diminishing CFL condition leading to a different control flow and (2) previous approaches have not utilized and examined the performance of ALTS using optimistic **parallel discrete event simulation (PDES)**.

To arrive at a performant implementation of ALTS, we employ two types of optimizations: throttling of optimistic execution and dynamic load balancing. The limiting of the progress logical

processes are able to make to improve the performance of optimistic PDES simulators is an extensively studied subject. Many previous approaches throttle optimistic PDES by inspecting either simulator state (e.g., [30, 65]) or hardware resource utilization [25]. We refer the reader to the review of Das [24] and therein cited works for a comprehensive overview. A common limitation of these methods is that they fail to fully exploit application specific knowledge but rather can only infer it through statistical methods. In the work of Bauer et al. [3] and Lindén et al. **??**, application-specific knowledge is used to provide upper bounds by which neighboring communication must be received. These *dynamic local time windows* block logical process execution once further progress is guaranteed to result in rollback. This technique is used to improve the performance of PDES-based simulation of the reaction-diffusion master equation. Simulation of conservation laws shares two important characteristics with reaction-diffusion master equation simulation: (1) it is not possible to place a lower bound on event arrival times (precluding the use of lookahead), and (2) it is possible to easily predict scheduling times of future events. Our optimizations rely on a similar mechanism to identify and eliminate misspeculation for the local timestepping algorithm.

Dynamic load balancing for PDES has also been examined extensively in previous work. Earlier work focused mostly on measured compute load of the cores to determine when and how to rebalance. For example, in the work of Boukerche and Das [7], a process migration strategy was presented that balanced workload based on CPU-queue length. Glazer and Tropper [35] present a method to dynamically load balance by migrating processes based on measurements of local clock progress and a bin packing algorithm. The approach presented by Avril and Tropper [2] uses the number of events processed but added estimates of the change in communication as an imbalance metric within the context of digital circuit simulation. In the work of Lindén et al. [52], a voxel-based, load balancing protocol was introduced using an imbalance metric based on rollbacks. More recently, Mikida et al. **??** examined the use of load balancing capabilities in the Charm++ [45] runtime to provide dynamic load balancing capabilities to PDES simulation models in ROSS [17]. They examined the efficacy of both a centralized greedy heuristic as well as a distributed strategy that utilizes a gossip protocol to avoid the synchronization cost of more centralized methods [54]. Our technique has some similarities to these previous methods but offers a more tailored approach given the specific characteristics of our application (e.g., handling wet and dry cells separately) and takes advantage of the 1D spatial domain topology to compute partitions extremely efficiently.

A key outstanding issue has been the adoption of adaptive locally timestepped methods by the scientific computing community. PDES as well as the mathematical foundations of ALTS have been around for more than 30 years, yet neither have managed to gain substantial traction in the scientific computing community [33]. Key barriers to entry remain the productive and maintainable introduction of ALTS into an existing code base while still meeting the performance needs of the higher-performance computing community [40]. Adaptive mesh refinement packages such as Clawpack [21] and ExaHyPE [59] solve conservation laws by requiring the user only to specify key characteristics of the conservation law. Other packages try to provide isolated functionality that can be integrated into a larger software stack—for example, SUNDIALS [41] provides optimized routines for timestep integration of initial value problems. Our proposed API most closely follows the second approach. The objective being to rapidly introduce ALTS into existing applications to demonstrate advantageous speedup and to contain the complexity of ALTS within a timestepping layer.

2 OPTIMIZING ALTS

The ALTS method for conservation laws is designed to ensure that the solution remains free of spurious oscillations by producing a total variation diminishing solution [49]. A detailed discussion on the justification for the ALTS algorithm can be found in the work of Bremer et al. [10]. In

this section, we will present a high-level overview of discretizing conservation laws, the ALTS algorithm, and performance optimizations for running ALTS using an optimistic PDES engine.

2.1 Numerical Solutions to Conservation Laws

Conservation laws model the evolution of conserved quantities between regions of the domain. As a partial differential equation, these laws are represented as

$$\partial_t u + \partial_x f(u) = S, \tag{1}$$

where u is our conserved quantity (e.g., mass or momentum) and f is the flux, which encodes how those quantities are exchanged with the surrounding region, and S is the source term (e.g., friction). These equations are notoriously difficult to stably simulate, as their solutions may form discontinuities. When discretizing these methods, particular care must be taken to faithfully approximate the exchange of conserved quantities between regions of the domain.

In this work, we only consider first-order methods for 1D domains. Consider a grid, $x_{lb} = x_0 < x_1 < \ldots < x_n = x_{ub}$. The first-order finite volume method will approximate the evolution of the average conserved quantity in cell j as

$$U_j(t) - U_j(s) = \Delta x_j \int_s^t S_j(\tau) - \frac{\hat{F}_{j+1/2}(\tau) - \hat{F}_{j-1/2}(\tau)}{\Delta x_j} \, \mathrm{d}\tau. \tag{2}$$

Our discretization will assume that conserved quantities are piecewise constant in time. The use of a continuous integral is meant to save on indexing headaches but should be practically understood as a discrete sum that can be evaluated exactly. The numerical flux $\hat{F}_{j+1/2}$ approximates the exchange of conserved quantities between cells j and $j+1$. The maximum speed at which information can pass from one cell into another is denoted as $|\Lambda|$, which is computed to be the magnitude of the largest eigenvalue of the Jacobian of the flux, $\partial_u f(u)$.

The purpose of local timestepping is to not require each cell to update at each discrete time. To help identify when cells update, let t_j^m denote the simulation time of the m-th update on cell j. Note that the state U_j must satisfy Equation (2) at these update times.

To ensure that the solution is free of oscillations, we require two conditions to be satisfied. These restrictions arise due to mathematical constraints, not limitations of the implementation. First, updates must obey the CFL condition, which limits the allowable time between successive updates [39]. Consider an element j and its neighbors N, and define t^* as the most recent time that every element has updated—that is,

$$t^* = \min_{n \in N} \max_m \{t_n^m\}. \tag{3}$$

A proposed update advancing element j from t_j^m to t_j^{m+1} satisfies the CFL condition if

$$t_j^{m+1} - t^* \leq C_{CFL} \frac{\Delta x_j}{\max_{n \in N} |\Lambda_n|(\tau)} \quad \text{for all } t^* \leq \tau < t_j^{m+1}, \tag{4}$$

where C_{CFL} is a discretization specific constant, $|\Lambda|$ is the largest wave speed over the element's stencil, and Δx_j is the size of the element. This inequality limits the allowable change of the conserved quantities within an element between successive updates by restricting the timestep size relative to flow rates. Second, elements must update in a locally ordered manner. In the ALTS setting, not every element updates at every timestamp. Given two adjacent elements, only one element may update more frequently than the other. To flip which element substeps relative to the other element, the two elements must pass through a *synchronization time* where both elements update at the same timestamp. We refer to element update traces that obey this behavior as *locally ordered*.

```
1  def update(forced_update):
2    #read simulation time and actor from DES context
3    t,submesh = get_state()
4
5    if (submesh.t_prev==t or
6        (not forced_update and sbumesh.t_next != t ) or
7        n_outstanding_msgs(t,submesh) > 0):
8      return
9
10   advance(t, submesh)
11   submesh.t_prev = t
12   submesh.t_next = compute_t_next(submesh)
13
14   for neigh in submesh.get_neighbors():
15     requires_update = inhibits_progress(submesh,neigh)
16     schedule(t+eps, neigh.get_id(),
17              push_flux(make_metadata(submesh, neigh),
18                        requires_update))
19
20     if (submesh.t_next > t and
21         n_outstanding_msgs(submesh.t_next, submesh) == 0):
22       schedule(submesh.t_next, submesh, update(False))
```

Listing 1. Update event.

For performance purposes, we partition elements into *submeshes*. Submeshes update constituent elements using a single timestep, which is limited by the most stringent local CFL condition. Since elements within the submesh step synchronously, local ordering is trivially satisfied within the submesh. By changing the number of elements per submesh, we gain the ability to tune the computational cost of submesh updates to amortize simulator overheads. As we show later, balancing simulator overheads with computational work is essential to obtaining good performance.

As a discrete event simulation, each actor corresponds to a submesh along with information about neighboring submeshes required to update element densities and compute the CFL condition at the boundary of the submesh. Also referred to as *logical processes*, actors contain the state of the system, interacting with one another only through messages managed by the PDES engine. Events are scheduled using a `schedule` routine that accepts a timestamp, actor, and functor as arguments or can be executed within the same event using a `schedule_inline` routine. The algorithm consists of two types of events: `update` shown in Listing 1 and `push_flux` shown in Listing 2. A submesh executes an `update` event to advance the submesh state to the event execution time (Listing 1, line 10). The updated halo state needs to be shared with neighboring submeshes, which is done via `push_flux` events (Listing 1, line 16). Note that updates can be scheduled as part of the `push_flux` event (Listing 2, line 9), leading to a potential cascade of updates. Showing that this interleaved execution of `update` and `push_flux` events utilizes the correct state and terminates has been carefully examined in the work of Bremer et al. [10]. The `eps` delay in scheduling the event `push_flux` is a small amount of subtime used to correctly order updates and push fluxes. The `advance`, `accumulate`, and `make_meta_data` functions encapsulate the mathematical discretization of the conservation law. The halo regions computed as part of `make_meta_data` for discontinuous Galerkin-based approaches consist of the state variables evaluated at shared interfaces. Higher-order finite volume methods would require sending larger halo regions. As presented here, ALTS does assume that the halo region is entirely contained in a single neighboring submesh. The next timestep is computed in `compute_t_next` to be the largest timestep that

```
def push_flux(metadata, forced_update):
  #read simulation time and actor from DES context
  t,submesh = get_state()
  accumulate(t, submesh, metadata)
  if n_outstanding_msgs(t,submesh) > 0: return

  submesh.t_next = compute_t_next(submesh)
  if ( forced_update ):
    schedule_inline(submesh, update(True))
  elif (submesh.t_next > t and
        n_outstanding_msgs(submesh.t_next, submesh) == 0):
    schedule(submesh.t_next, submesh, update(False))
```

Listing 2. Push flux event.

satisfies (4). Since the timestep depends on the boundary metadata, the next update time may change as `push_flux` events are processed and the `update` event must be rescheduled. We know that the most recent actor state will have access to all relevant metadata, and thus we only execute the update consistent with the current `submesh.t_next`. All other updates become no-ops, either because their scheduled execution time does not match `submesh.t_next` or the update has already been executed.

The two functions `n_outstanding_msgs` and `inhibits_progress` examine the local representation of the distributed state of the system to determine what events should be scheduled next. The `n_outstanding_msgs` function provides a *lower* bound on the number of outstanding messages to be processed on the actor by time t. By knowing that a message will be processed before the next update time, we defer scheduling updates until the waited upon messages are received, as any event executed after `n_outstanding_msgs` that returns a number greater than zero is *guaranteed* to be rolled back. The `inhibits_progress` function checks to see whether the actor is unable to make progress without forcing its neighbor to update. Forcing a neighbor to update may arise due to a local ordering violation, an inability to propose a CFL-stable next timestamp, or performance benefits of having the neighbor prematurely update.

2.2 The Devastator Runtime

To measure the performance of the ALTS algorithm, we implemented it using the Devastator PDES runtime (first described and used in the work of Chan et al. [20]). Devastator is a general-purpose, optimistic PDES engine that coordinates event execution and rollback required by Jefferson's Time-warp protocol. Devastator is built using the GASNet-EX runtime [6] and utilizes light-weight active messages (RPCs) to invoke events on remote logical processes. Since GASNet-EX active messages are tailored to the specific interconnect hardware, they are efficient at delivering the small asynchronous events required of PDES. It is implemented with modern C++ to improve programmer productivity, uses parallel threads and processes to enable distributed memory execution, and includes several optimizations to improve communication and synchronization efficiency. The modern C++ interface allows clean specification of event classes, their serialization, and their `execute` and `rollback` functions. Since conservation laws are not reversible in general, portions of the actor state are logged during `execute`, and the actor state can be restored to the state prior to the `execute` call by calling the `rollback` function.

To illustrate the parallel efficiency of the runtime, Figure 1 shows weak scaling performance of Devastator running a modified version of the well-known PHOLD benchmark [32]. PHOLD consists of a number of rays that bounce between actors. The ray arrival times are distributed exponentially, and a ray terminates after a fixed number of events. To determine a ray's next

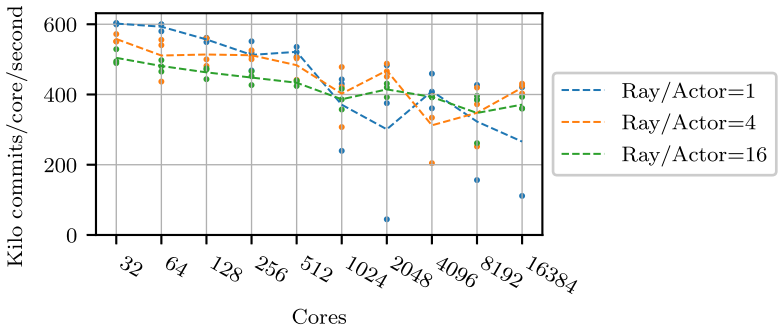


Fig. 1. Weak scaling performance for the modified PHOLD benchmark on NERSC's Cori-Haswell partition.

destination, the actors are lexicographically sorted based on rank ID and then local actor ID, and the target rank is selected by using a wrapped normal distribution with standard deviation of 16 and a mean of the sending actor ID. This modification allows us to tune the communication pattern of the benchmark to more closely resemble applications with locality, such as the ALTS simulations studied here. In the figure, we show the *per core* event commit rate as the number of cores is varied (with the number of logical processes per core fixed at 1,000). The different series in the figure vary the number of active events (rays) per logical process in the system (corresponding to the average event queue depth for each logical process). Each configuration is run three times, with each trial represented as a point and the line representing the average commit rates of the trials. As can be seen in Figure 1, parallel efficiency (performance per core) decreases to approximately 55% to 75% as the number of cores increases from 32 cores to 16,384 cores, corresponding to a 270× to 3800× overall speedup in aggregate event commit rate. In certain cases, significant variance is observed. Determining whether the jagged nature of the scaling results is due to specific paths taken by the rays or due to system noise will be addressed in a more thorough future treatment of Devastator.

2.3 Performance Optimizations

Although satisfying the CFL condition and local ordering leads to stable computation, careful management of PDES overheads is required to achieve expected speedups. In this section, we introduce a set of performance optimizations focused on eliminating bad speculation (Sections 2.3.1—2.3.4) and a dynamic load balancing scheme (Section 2.3.5).

2.3.1 Binning Timestep Sizes. The need for local ordering of events leads to potentially increased forced synchronizations. For example, given submeshes stepping with timesteps 17 and 18, the submesh stepping with size 17 would step to time 17 and then have to update again at time 18 to maintain local ordering. To address this, we bin future timesteps into powers of 2 times a fixed user-supplied Δt_{min}. In this example, both submeshes would take binned timesteps of size 16. To simplify computation all timestamps are kept as integers with units of Δt_{min}. The Pythonic pseudocode for timestep binning is

```
dt_unbinned = t_next - t_prev
dt = power(floor(log2(dt_unbinned)), 2)
t_next = dt * (t_next//dt).
```

In practice, we implement this efficiently using bit shifts. The impact of this optimization is that we reduce forced updates by having submeshes with roughly the same timestep step to the same timestamps. This also concentrates updates to certain multiples of Δt_{min} leading to improved simulator parallelism and reduces the depth of the algorithm. The downside of this approach is

that we are unable to take the largest feasible timestep, potentially increasing the total number of updates.

2.3.2 Waiting on Forced Updates. When a submesh sends a `push_flux` with the `requires_update` argument as true, we know that the submesh will receive a `push_flux` from the neighboring submesh at the current time. Since any updates speculatively executed *before the neighbor's* `push_flux` *event has arrived* would be rolled back, the function `n_outstanding_msgs` examines the submesh's state and delays scheduling further updates until after the waited upon messages have arrived.

2.3.3 Forcing Neighbors to Update. Since the CFL condition is taken relative to the last update time of the submesh or its neighbor, once a submesh updates it to its binned timestamp, it can still make further progress. As an example, consider two submeshes both with an unbinned timestep of 14 starting at time $t = t^* = 0$. If one submesh is delayed, the other can execute binned updates at times 8 (`dt` = 8), 12 (`dt` = 4), and 14 (`dt` = 2), before being unable to schedule another update. Once the delayed submesh updates to time 8, the resulting `push_flux` will lead to rollback of two updates on the other submesh. To address this, we examine the difference in the proposed next timestep based on the current t^* as defined in Equation (3) and if t^* were equal to the submesh's current simulation time (i.e., the submeshes are synchronized). If the ratio of the synchronized timestep size versus the unsynchronized timestep size exceeds 2, we force the submeshes causing the timestep reduction to update. In the example, the not-delayed submesh would update at time 8, and the proposed next update has a timestamp `dt` = 4 using the current $t^* = 0$. However, if the submeshes were synchronized (i.e., $t^* = 8$), the next timestep would be $\texttt{dt} = 8 \geq 2 \cdot 4$. Hence, the submesh that updated to $t = 8$ would force its neighbor to update. By waiting on submeshes that have been issued push fluxes that require updating as part of Section 2.3.2, forcing the neighbor to update leads to waiting on the neighboring submesh rather than incorrectly speculatively executing updates at times 12 and 14.

2.3.4 Establishing Upper Bounds on Neighboring Updates. When Λ and Δx_j are constant in time but vary significantly spatially, message arrival times are independent of the dynamic state of the neighboring submeshes, and it should be possible to recover an efficient task-graph-like execution. To recover this behavior in our more general discrete event simulation setting, we note that the next update time for a submesh must be less than the next unbinned update time required by the internal elements of that submesh. This upper bound is shared with neighboring submeshes as part of the `make_metadata` message. By incrementing `n_outstanding_msgs` if the time argument exceeds the upper bound, the neighboring submesh pauses scheduling of further events until the submesh's `push_flux` event is processed.

2.3.5 Dynamic Load Balancing. Last, as the simulation advances and simulated regions are flooding or draining, the load will need to be rebalanced. Dynamic load balancing for hurricane storm surge for synchronous timestepping has been explored in the work of Bremer et al. [12]. Findings indicate that although the compute load changes substantially, it does so in a slow gradual manner, and a semi-static-based approach performs roughly as well as a fully asynchronous load balancing strategy. Here we use a semi-static approach due to its relative simplicity compared to the fully asynchronous load balancer.

At a high level, the simulation will be broken into epochs of a fixed size. At the end of each epoch, we approximate the computational work of a submesh as $n_{\text{elem}}(1/\Delta t - 1/\Delta t_{\text{epoch}})$, where Δt is the next proposed timestep size for that submesh (i.e., `submesh.t_next` minus the epoch's timestamp). All load information will be aggregated on rank 0. There the submesh to rank assignment will be recalculated to minimize load imbalance, and all submesh relocations will be issued. By fixing the

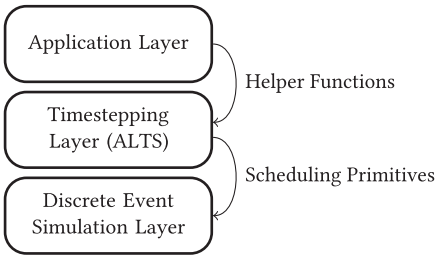


Fig. 2. Stack diagram for ALTS.

element to submesh mapping and only having to assign submeshes to ranks, the complexity of the load balancing problem is considerably reduced, enabling more frequent load balancing. We emphasize that we *oversubscribe* compute resources—that is, assign multiple actors to each rank to hide message latencies and ensure consistent utilization of compute resources. The load balancing problem must satisfy two constraints: (1) compute loads must be balanced between ranks, and (2) the total number of elements assigned to a rank must not exceed the local memory capacity. The second constraint is required since idle regions of the mesh do not update, so they could potentially be assigned to a single rank.

For 2D simulations such as those performed by Bremer et al. [12], standard multi-constraint graph partitioning solutions such as Metis [46] are able to rebalance the problem. However, for the 1D shallow water problems considered in this work, we found that Metis struggled to generate partitions with low load imbalance. Instead, we used a custom partitioner for this work. Submeshes are classified as either wet or dry. A submesh is dry if it has zero work (i.e., only updates once per epoch) and is otherwise wet. Wet submeshes are partitioned by balancing the work across each rank. In addition, each rank is then assigned roughly the same number of dry submeshes (some ranks may receive an extra submesh due to discrete effects). For 1D problems, wet and dry partitions can be efficiently represented with the use of splitters of the global submesh graph, where rank r is assigned the submeshes between the r-th and $(r + 1)$-th splitter. By assigning each rank two contiguous sets of submeshes, we reduce inter-rank communication as each rank contains no more than four submeshes that send their `push_flux` events to another rank. Upon computing a new submesh-to-rank mapping, we greedily permute the new map's rank IDs to minimize the number of submeshes that need to be moved between ranks.

3 SEPARATING PHYSICS FROM TIMESTEPPING

In this section, we outline implementation details to help the adoption of ALTS. Both PDES systems as well as asynchronous runtimes require substantially more thought to correctly program than the MPI+X programming model commonly used in scientific computing. Often, debugging is difficult due to the presence of bugs arising from strange event orderings that only show up during optimized parallel runs. This presents a considerable speed bump to the adoption of PDES-based local timestepping as application developers will be loath to maintain this timestepping code themselves. What we propose is a strict interface to separate the application physics from the timestepping logic, as shown in the stack diagram in Figure 2. The proposed interface in this section abstracts the physics via a `Problem` class with certain required type definitions and member functions. We have prioritized minimizing the intrusiveness of the API into the application layer. Often, applications have complicated application-specific physics terms that represent the result of substantial investment of resources. One of the goals of this interface is as follows: given a flat MPI implementation with a "nice" data layout, the porting of the application to an ALTS

```
class Problem {
  typename bdry_t
  typename msg_t
  typename dynamic_t
  typename dynamic_bdry_t

  dynamic_t& get_dynamic_state()
  void initialize(uint64_t id)
  void build_u_next(double t, double dt, dynamic_t& u_old)
  msg_t make_message(uint id_me, uint id_to)
  void lift_bdry(uint id_me, uint id_from, const bdry_t& bdry)

  double compute_K_internal(double t)
  double compute_K_bdry(uint id_me, uint id_from,
                        const msg_t& bdry, double bdry_dx)
};
```

Listing 3. Problem class interface.

integrator should merely require removing existing timestepping logic and implementing new functions associated with determining allowable timestep sizes. This would allow domain experts to continue developing code for a programming model that looks like flat MPI while reaping the benefits of the more efficient ALTS time integration. A side effect of this objective is that we are also able to implement **synchronously timestepped (STS)** MPI and serial versions without modifying the problem class. In this section, we will provide a detailed discussion of the problem class and the functionality of the member methods and type definitions, discuss how the state is made roll-backable, and discuss performance trade-offs with cleaner abstractions.

3.1 Computation Required by ALTS

The purpose of the `Problem` class is to specify the physics required to update the conserved variables and compute local wave speeds required to determine timestep restrictions. We present the `Problem` class interface in Listing 3.

We begin by focusing on the physics information required during a `push_flux` event. When a submesh updates, the numerical fluxes on neighboring submeshes must be updated to ensure that Equation (2) continues to hold. To achieve this, the updated submesh calls the `make_message` function to construct a message of type `msg_t`, which encapsulates the information needed by its neighbors to update their fluxes accordingly. Note that often there is an opportunity to reduce the required message sizes by computing derived quantities before sending them to neighboring submeshes. For example, in the case of the discontinuous Galerkin method, where conserved quantities are approximated by polynomials, the numerical flux only depends on the value of the conserved quantity at the shared interface. In this case, the `make_message` function computes the necessary derived quantities and sends only those to the submesh's neighbors as part of the `push_flux` event.

When the neighboring submesh executes the `push_flux` event, it accumulates the flux exchanged based on the previous state up until that point in time and computes a new flux as part of the `process_message` routine. The representation of this partially computed right-hand side is encapsulated with the `bdry_t` type. Since the integration of this state is done in the local timestepping application layer, the `bdry_t` type must support standard vector space operations (multiplication by a scalar and addition of two `bdry_t` objects). Finally, when the neighboring submesh is updated, the boundary flux values are applied to all dependent degrees of freedom as part of the `lift_bdry`

function. To make the preceding processes concrete, for the methods considered in this work, the `make_message` routine would simply identify the average conserved quantity associated with the cell neighboring the submesh and send that state as a `msg_t`. When the neighbor executes the associated `push_flux` event, the previous flux at the shared interface is updated by scaling it based on the elapsed time, and a new current flux is computed in `process_message`.

Next, we turn to the physics required by the `update` event. The `build_u_next` function takes the state's current time and the desired timestep as an argument and then returns the *internal* state at time `t + dt`. In this case, the implementation is left to the domain scientists. We implement this as a simple forward Euler step. In addition, `build_u_next` is responsible for integrating boundary conditions. The `build_u_next` function in conjunction with the `lift_bdry` function discussed in the previous paragraph computes all of the necessary terms required to satisfy Equation (2) for a particular submesh.

One critique of the `build_u_next` abstraction is that it requires the application to understand how to perform a simple sequential integration, undermining the physics/timestepping separation. However, we believe that existing applications (e.g., flat MPI code) would already have implemented this functionality, therefore not imposing a significant burden on domain scientists. The advantage of this approach is that it allows for loop fusion. Low-order methods such as the ones considered in this article are memory bound. So requiring the entire computation of the right-hand side followed by scaling the result by the timestep and adding it to the previous state would incur additional memory traffic. Given the emphasis on performance, we have chosen the less clean but more performant option for the abstraction.

The second functionality that the problem layer must supply are the CFL-related constraints. Similar to the state updates, we partition the computation into two calls based on whether the CFL condition depends on solely local or distributed information. We refer to K as the wave speed $|\Lambda|/\Delta x$. The local CFL bound is computed as part of `compute_K_internal`. From an implementation perspective, the computation of the CFL condition tends to be quite expensive. Thus, we opt to fuse the computation of the internal K with the `build_u_next` call and cache the result. When computing the internal K value, the only additional information that requires consideration are the domain's boundary conditions that may depend on the time argument, `t`. Note that these evaluations are only at snapshots in time, the control flow associated with properly evaluating the CFL condition, Equation (4) is delegated to the timestepping layer.

When a neighbor updates and the associated `push_flux` is processed, we need to recheck the CFL condition. Since the neighboring state only affects a limited number of cells, we introduce an additional call `compute_K_bdry`, which will recompute $|\Lambda|/\Delta$, based on the neighbor's updated wave speed. Note that we assume that the relevant neighbor's $|\Lambda|$ can be computed based on the received `msg_t`. In our experiments, we have always found this to be the case.

We conclude with a remark on the extension of this API to higher-order timestepping methods. One approach is based on ADER methods [29]. These are single-step methods that rely on a polynomial space-time representation of the state. To implement ADER timestepping, we would only need to implement a single ADER timestep along with the relevant boundary types at the application layer. The helper function implementation could then be passed to the unmodified timestepping layer.

Alternatively, high-order strong stability preserving Runge-Kutta methods [23, 36] are also commonly used due to their non-linear stability for conservation laws. The key property of these methods is that they can be represented as convex combinations of Euler steps. Implementing this class of Runge-Kutta methods would require modifications to the timestepping layer. Advancing a submesh would require a second small subtime quantity ϵ^2, which would be used to identify which stage is currently being processed. Each update would cause several halo exchanges to occur at

Table 1. Lines of Code Associated with Different Modules in the ALTS Code

Problem	
Burgers'	618
Shallow Water	773
Euler	652
Timestepping Layer	
ALTS-deva	2,934
STS-MPI	566
STS-serial	466
Common	245

time t with the final state being updated at the update time `t + dt`. All of the control flow details are managed in the timestepping layer, and the forward Euler implementation of `build_u_next` function is sufficient for supporting this class of Runge-Kutta methods. Introduction of multi-stage Runge-Kutta methods introduces considerable complexity, but the hope is by sequestering the implementation details from users, we hope to promote adoption of these techniques.

3.2 Rolling Back the Application Layer State

The discussion up until this point has been agnostic to PDES. However, one important aspect of the API is how the application deals with rollback. The API has been designed to allow the user to optimize the amount of state required to be captured by separating data members that vary between updates (e.g., state variables) from those that remain constant (e.g., mesh geometry). Since the computations associated with `build_u_next` are not reversible, Devastator is required to store the entire dynamic state until **global virtual time (GVT)** passes the event time and the memory can be reclaimed. For this reason, we return the next state by value, which we will swap with the old state. Afterward, the current dynamic state will be stored in the `Problem` class and the old dynamic state will be managed by the Devastator runtime, which will deallocate it once it is safe to do so. Similarly, for the push flux event, we split apart the dynamic state of the boundary from things that remain unchanged during the simulation. By explicitly requiring these types to be defined, we enable Devastator to ensure that events remain reversible. With these dynamic types, rollback can simply be performed by swapping old dynamic types into the problem class, restoring the state prior to an event.

Last, we require that the `msg_t` and `Problem` types be serializable. Since push fluxes are sent between nodes and submeshes are migrated during load rebalancing phases, we need to ensure that the state can be serialized into a buffer and sent across the wire.

To understand the practical impact of this interface, we have collected lines of code for the various modules. Using the `Problem` API, we have implemented three different sets of conservation laws, which will be outlined in more detail in the next section. In addition, we have also used the code to implement a serial synchronously stepped implementation as well as an MPI-based implementation. In Table 1, we compare the lines of code for each module. Note that problem classes are specified by the equation they solve (Burgers', Shallow Water, Euler), and the timestepping layer is labeled according to the algorithm (STS or ALTS) and how it is run (Devastator, which we abbreviate as deva, MPI, or serial). We additionally note a block of code as *common*. This code is used across all denoted modules and thus cannot be assigned merely to a single group. From the data, we remark that the ALTS implementation is significantly larger than the STS implementation. This is in part due to more features including performance instrumentation and load balancing

Table 2. Conservation Laws and Their Configurations

Name	State	Flux	Source	Wave Speed	Numerical Flux
Burgers'	u	$u^2/2$	0	$\lvert u\rvert$	Godunov
Shallow Water	$\begin{pmatrix} h \\ q \end{pmatrix}$	$\begin{pmatrix} q \\ \frac{q^2}{h} + \frac{gh^2}{2} \end{pmatrix}$	$\begin{pmatrix} 0 \\ -gh\partial_x z \end{pmatrix}$	$\sqrt{gh} + \lvert q/h\rvert$	Local Lax-Friedrichs
Euler	$\begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix}$	$\begin{pmatrix} \rho u \\ \rho u^2 + P \\ (E+P)u \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$	$\sqrt{\gamma \frac{P}{\rho}} + \lvert u\rvert$	Local Lax-Friedrichs

capabilities as well as more complexity when compared with the STS-based implementations. However, we observe that using this abstraction layer, we achieve significant code reuse with a single problem class consisting of approximately 1,000 lines of code being able to leverage three different runtimes. In other words, across the three different conservation laws, no less than 74% of the code is reused.

4 NUMERICAL EXPERIMENTS

To demonstrate the robustness of our timestepping algorithm and the proposed performance optimizations, we present results for three different 1D non-linear conservation laws. We present a high-level overview of the equations considered in this section in Table 2. The state column corresponds to u and the flux to $f(u)$ in Equation (1).

Burgers' equation may be thought of as the simplest non-linear conservation law with the local wave speed being proportional to the density itself. For Burgers' equation, we consider two initial conditions: a shockwave problem $u_0^{sh}(x) = 1 - \text{sign}(x)/2$ and a rarefaction problem $u_0^{rf}(x) = \text{sign}(x)$.

The second set of conservation laws are the shallow water equations. These are depth-averaged equations where h models the water column height and q models the horizontal momentum. When the seafloor bathymetry (given as z) is non-flat, we rely on the hydrostatic reconstruction of Audusse et al. [1]. The acceleration due to gravity, g, is set to 1. We consider three initial conditions. The first is the lake at rest (lar) problem for which the solution is a constant. This problem is notable because $|\Lambda(x,t)|$ is a constant for all space and time. Hence, any local timestepping benefit is derived solely from variation in element sizes, Δx_j. The other problem we consider is the Carrier-Greenspan (cg) problem [18] as configured in the work of Bokhove [5]. This is a π-periodic solution to the shallow water equations that simulates water flowing up and down a sloped beach. For clarity, we have included three snapshots in Figure 3. The Carrier-Greenspan problem serves as a good proxy for hurricane storm surge. In both cases, the large variations in wave speed Λ are being driven by coastal flooding. Last, we consider the dam break problem, wherein a discontinuity in water column height on a flat bathymetry ($z = 0$) is simulated, specifically,

$$h_0(x) = \begin{cases} 1 & \text{if } x < 0 \\ 1/16.1 & \text{if } x > 0 \end{cases} \quad \text{and} \quad q_0(x) = 0.$$

The final set of equations we consider are the Euler equations, used to simulate compressible gas dynamics. The conserved quantities correspond to fluid density (ρ), momentum (ρu), and internal energy per unit volume (E). We relate the internal energy to pressure via $P = (\gamma - 1)\left(E - \rho u^2/2\right)$, where $\gamma = 1.4$ is the heat capacity ratio. For this problem, we consider two sets of initial conditions, the Sod shock tube problem [63] and the interacting blast wave problem [68]. For completeness,

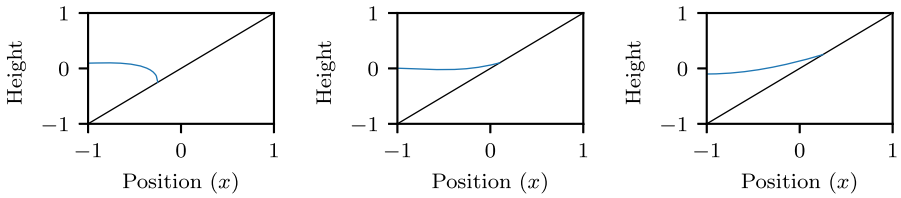


Fig. 3. Water surface height (in blue) and seafloor (in black) relative to mean geoid for the Carrier-Greenspan problem at various time points.

the initial conditions for the Sod problem are

$$\rho_0(x) = \begin{cases} 0.125 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}, \quad \rho u_0(x) = 0, \quad E_0 = \frac{1}{\gamma - 1} \begin{cases} 1 & \text{if } x < 0 \\ 0.1 & \text{if } x > 0 \end{cases}, \tag{5}$$

and for the blast wave problem are

$$\rho_0(x) = 1, \quad \rho u_0(x) = 0, \quad E_0(x) = \frac{1}{\gamma - 1} \begin{cases} 1000 & \text{if } x < -0.4 \\ 0.1 & \text{if } |x| < 0.4 \\ 100 & \text{if } x > 0.4 \end{cases}. \tag{6}$$

Although the real-world applicability of these discretizations is limited, we expect performance trends to be largely the same for 2D problems with more sophisticated discretizations. Efficient execution of ALTS fundamentally relies on selecting an appropriate task granularity to amortize runtime overheads. As the number of push fluxes sent per update changes (as we go to higher dimensions, each neighbor receives one message) and local updates become more expensive due to more sophisticated discretizations, the key parameters that determine task granularity (e.g., minimum number of elements required per submesh) may change, but the performance trends will ultimately remain similar. Section 4.1 details the process of selecting the task granularity to balance the impact of event overheads with other factors.

Boundary conditions are weakly enforced by setting the boundary values on the outside of the numerical flux to the analytic value for all problems with the exception of the blast wave problem. For the blast wave problem, we set reflective boundary conditions—that is, $\rho_{\partial\Omega} = \rho$, $\rho u_{\partial\Omega} = -\rho u$, and $E_{\partial\Omega} = E$.

Meshes are generated by warping uniformly distributed points on the interval $[-1, 1]$. The only exception to this is the blastwave problem for which we scale the domain to $[-0.5, 0.5]$. We consider a *uniform* mesh for which the warp function is the identity map, and a *polynomial* warp function, $p(x) = C(x^3/3 + \varepsilon x)$, where $\varepsilon = 0.02$ and C is chosen so that $p(1) = 1$ and $p(-1) = -1$. The ratio of largest element to smallest element is approximately ε^{-1}. These kinds of locally refined meshes are common for fluid simulations where high accuracy is only needed near regions of interest. We specify submeshes via an iterative procedure designed to balance the work performed by each submesh during the simulation. For stencil-like kernels, the cost of performing an update is $O(n_{\text{elem}})$. The total work per submesh is approximated as the number of elements assigned to a given submesh times the inverse binned timestep of the submesh, assuming $|\Lambda| = 1$. The iterative submesh partitioning process updates the weight associated with a given element and then repartitions the submeshes. This process is repeated until a fixed point is achieved or a preset number of iterations has been computed.

Beyond the mesh, submeshes, and initial conditions, we need to specify a few inputs. The smallest timestep size $\Delta t_{\min}$ is set through the parameter $(|\Lambda|/\Delta x)_{\max}$, which specifies the largest expected value of $|\Lambda|/\Delta x_j$ during the simulation. Note that setting this parameter is not very onerous, as the simulation will still perform well if this value is set significantly below the true $(|\Lambda|/\Delta x)_{\max}$. This value determines the smallest timestep size, by quantizing the simulation in $2\lceil t_{end}(|\Lambda|/\Delta x)_{\max}\rceil$ steps. The extra factor of 2 is the inverse of the C_{CFL} factor required for total variation diminishing timestepping. The minimum timestep size is set to t^{end}/n_{steps}. In addition, we specify the epoch time as Δt_{epoch}. To avoid issues with timestep binning, Δt_{epoch} is rounded down to the nearest power of 2 times $\Delta t_{\min}$.

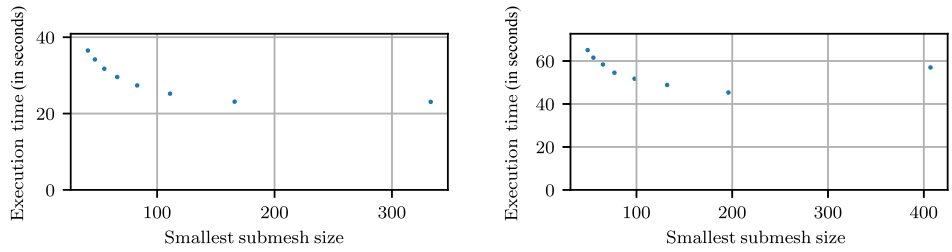
All numerical results here are run on NERSC's Cori-Haswell partition. Each node consists of two Intel Xeon Processors E5-2698 v3. Each Xeon processor has 16 cores. When running Devastator, there are two parallel settings: (world=threads) runs with a single process and uses shared memory parallelization where each thread participates in executing simulation events, and (world=gasnet) is used for distributed simulation with multi-threaded processes. In this case, one thread per process is dedicated to sending and receiving messages, whereas the remaining *worker* threads execute simulation events. To avoid NUMA effects, we assign a separate process to each processor on a node.

4.1 Event Overheads

What makes ALTS different from traditional PDES applications is that we can freely manage event granularities by tuning the number of elements assigned to each submesh. We solve the lake at rest problem on a fixed mesh with 20,000 elements for the uniform mesh and 75,000 elements for the polynomial mesh. We selected the simulation time to be sufficiently long so that reliable measurements could be made. We set $t^{\text{end}} = 64$, $(|\Lambda|/\Delta x)_{\max} = 20{,}200$ for the uniform mesh, and $t^{\text{end}} = 1$, $(|\Lambda|/\Delta x)_{\max} = 1{,}115{,}208$ for the polynomial mesh. The simulation is run varying the number of submeshes, ranging from one to eight submeshes per worker thread. As the number of submeshes increases, the concurrency available to the discrete event simulator increases (i.e., overdecomposition), allowing useful work to occur while waiting for messages to arrive. Furthermore, each submesh must satisfy the CFL condition for fewer elements, leading to a more optimal local CFL condition. This is an important performance tuning knob for irregular algorithms, as we are able to weigh the benefits of increased irregularity versus reduced PDES overheads.

We ran the following experiments on 64 cores on Cori. Each run consists of four processes—one per NUMA domain—with 15 worker threads per process. Enumerating the submeshes from left to right, we assign submeshes in contiguous chunks—that is, rank k is assigned submeshes $C_F k$ through $C_F(k+1)$, where C_F is the number of submeshes per rank. The critical path will lie on the rank with the submeshes with the fewest elements. Since the work performed per submesh is balanced for the lake at rest problem, the ALTS-related work completed on this rank is comparable to that of other ranks. However, this rank must execute more updates and therefore incurs more event overheads.

In Figure 4, we plot execution time versus the lowest number of elements assigned to a submesh. Each configuration was run five times, and the standard deviation of each set of runs was within 0.99% of the respective means. For either configuration, we see that once the element size per submesh drops below 70 elements, event overheads begin to dominate the execution time. Although there is a slight benefit to oversubscribing compute resources for the uniform mesh, for the polynomial mesh we observe a speedup of 1.26 going from an oversubscription factor of 1 (407 minimum elements per submesh) to 2 (196 minimum elements per submesh). For the uniform mesh, dependencies are approximately satisfied at the same time, but for the polynomial mesh, a neighboring submesh that is substepping must first make progress before the submesh with the



(a) Uniform (b) Polynomial

Fig. 4. Execution time versus smallest number of elements assigned to a submesh.

larger timestep is able to progress causing the performance degradation for the largest minimum submesh in Figure 4(b). Oversubscription allows us to update a different submesh while waiting for the substepped submesh to satisfy the stalled submesh's dependencies.

Last, we build a cost model that approximates the time elapsed updating a submesh of n_{elem}. Since the cost of the update kernel is $O(n_{\text{elem}})$, we assume a linear model of the form $cost(n_{\text{elem}}) = C_2 n_{\text{elem}} + C_1$, where C_2 corresponds to the cost to update one element and C_1 to roughly be the event overhead. Using the data for the polynomial mesh runs and omitting the data point for an oversubscription factor of 1 due to the aforementioned discrete effects impeding performance, we estimate the average task execution time by dividing the overall execution time by the largest number of updates committed on any rank, then use the average number of elements per submesh on that rank as the n_{elem} input. Due to the smooth nature of the mesh refinement, we assume that all submeshes on the selected rank update the same number of times. Via linear regression, we find $C_2 = 0.0438$ µs/element and $C_1 = 1.474$ µs with $R^2 = 0.999$. We note that we do not expect this cost model to generalize to larger problems. The selected problem sizes fit in the core's local L1 cache. As elements spill out of the local caches, we expect the cost to update an element, C_2, to increase, reflecting the reduced memory bandwidth. Using this cost model as a lower bound, we approximate that roughly 100 elements are needed to achieve a good resource utilization. At this task granularity, 74.8% of the time is spent doing useful work. This corresponds to an event size of 5.9 µs.

If we think of the lake at rest problem as an ordinary 1D stencil dependency graph, we can compare Devastator's event execution overheads with the large number of existing task-based runtimes using the analysis described in the work of Slaughter et al. [62]. The figure of merit in this analysis is the **minimum effective task granularity (METG)**. Specifically, METG(50%) is the smallest task size, in our case cells per submesh, for which half the time is spent doing useful work. In our case, the METG(50%) is extrapolated to 3 µs (i.e., $2C_1$), making it competitive with the fastest reported implementations in the work of Slaughter et al. [62]. It is in some sense unsurprising that Devastator would do well in this metric since, like many other PDES engines, it trades off some task scheduling flexibility (e.g., local work stealing) for the ability to efficiently execute small events, where execution time is largely driven by C_1. This poses interesting performance trade-offs for the PDES and task-graph runtime communities. If the irregularity of applications is determined to be limited, optimizing for small event execution may lead to better strong scaling properties. The other interesting question is how this efficiency changes as we scale out. As shown in Figure 1, the commit rate drops as we scale out to larger node counts. Quantifying the impact of the GVT reduction on METG(50%) is an important research question we will pursue in future work.

4.2 Description of Misspeculation

To explore when and where misspeculation is occurring, we consider a single-node (world=threads) problem with a 300,000 element mesh with a 6x oversubscription factor. We use

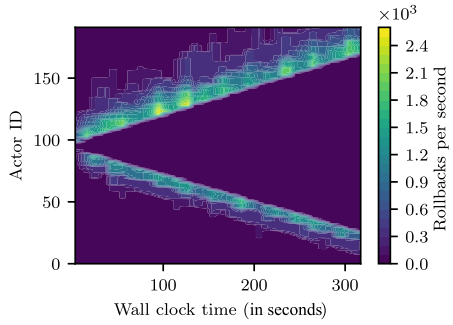
Table 3. Total Number of Updated Elements Committed and Rolled Back for Different ALTS Configurations

Configuration					Element Updates		
Problem	Initial Conditions	Warp	t_{end}	$(\lvert\Lambda\rvert/\Delta x)_{\max}$	Committed	Rollback	Rollback/Commit
Burgers	Shockwave	Uniform	1	6.060e+05	5.705e+10	2.094e+05	3.670e-06
Burgers	Shockwave	Polynomial	1	1.070e+07	4.126e+11	9.729e+04	2.358e-07
Burgers	Rarefaction	Uniform	0.75	6.060e+05	5.992e+10	2.253e+07	3.760e-04
Burgers	Rarefaction	Polynomial	0.75	1.070e+07	5.745e+10	1.510e+07	2.629e-04
Euler	Sod	Uniform	0.5	6.060e+05	1.008e+11	1.174e+07	1.165e-04
Euler	Sod	Polynomial	0.5	1.070e+07	4.077e+11	9.763e+06	2.395e-05
Euler	Blast wave	Uniform	0.038	2.240e+07	1.767e+11	2.601e+09	1.472e-02
Euler	Blast wave	Polynomial	0.038	4.000e+08	5.388e+11	1.137e+09	2.110e-03
SWE	Dam break	Uniform	0.75	6.060e+05	7.864e+10	1.276e+07	1.622e-04
SWE	Dam break	Polynomial	0.75	1.070e+07	4.701e+11	2.536e+07	5.394e-05
SWE	Carrier-Greenspan	Uniform	2π	1.599e+05	3.086e+11	1.795e+07	5.816e-05
SWE	Carrier-Greenspan	Polynomial	2π	1.584e+06	1.137e+12	7.277e+07	6.399e-05
SWE	Constant	Uniform	2	6.060e+05	1.818e+11	0.000e+00	0.000e+00
SWE	Constant	Polynomial	0.2	1.070e+07	8.696e+10	0.000e+00	0.000e+00

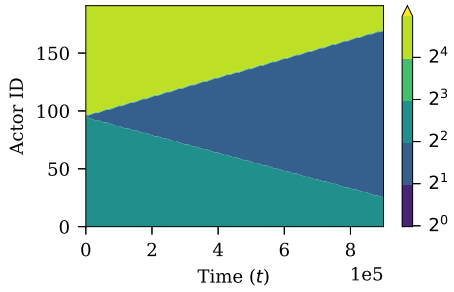
a trivial actor partitioning, leading to imbalanced work-to-rank assignments. However, as a study of speculation, this serves as a stress test since underworked ranks have the possibility to do more work that could be rolled back while waiting on messages from overworked neighbors. We run multiple configurations of the Burgers', Euler, and shallow water equations with several initial conditions and mesh types. We report the number of element updates committed and rolled back for each configuration as well as the ratio of the two statistics in Table 3. At a high level, we remark that the waiting heuristics outlined in Section 2.3 work remarkably well. For the SWE-lake at rest problem, no rollback of updates occurred during the simulation. For this configuration, the only source of variation in timestep sizes is due to variation in element sizes, not the local wave speed, which equals 1 throughout the entire domain. Insofar, the information sent via the upper bound messages anticipates when messages will arrive and allows the submesh to avoid performing updates that would be rolled back. Otherwise, the highest observed rate of rollback was for the Euler-blast wave problem with 1.5% and 0.2% of element updates rolled back for the uniform and polynomial meshes, respectively. Beyond the blast wave problem, all other configurations had a rollback rate of less than 0.05%.

To help understand where and why rollback is occurring, Figure 5 shows the number of rollbacks per second occurring on each submesh (actor) versus wall clock time compared to the timestep sizes taken by the submeshes versus simulated time for three of the problem configurations. Since we are considering problems where the critical path goes through a single actor, the simulated and wallclock times are roughly proportional. We claim that the cause for misspeculation arises from the unanticipated changes in timestep size and the resulting rollback cascade. When a submesh is stepping with a larger timestep than its neighbor, it will try to speculate ahead of its neighbors. While the neighbor's timestep remains stable, the submesh can use its knowledge of the neighbor's timestep along with the upper bound mechanism (see Section 2.3.4) to avoid misspeculation by waiting on an appropriate number of messages. However, due to the evolution of the equations, that neighbor may ultimately require a smaller or larger timestep, leading to unanticipated push flux events and corresponding rollbacks.

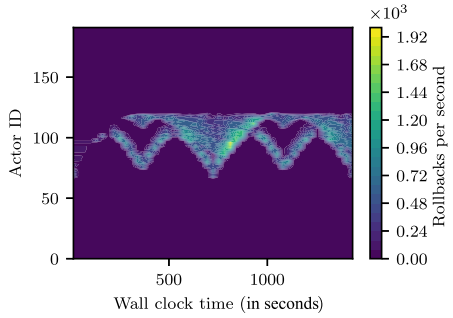
Consider the case of a timestep restriction where two neighbors are updating alongside one another, and one suddenly requires a smaller timestep. This is the case for the SWE dam break problem in Figure 5(a) and (b). In this problem, there are two waves emanating from the initial discontinuity for which the emergent interior region requires a more strict timestep than either



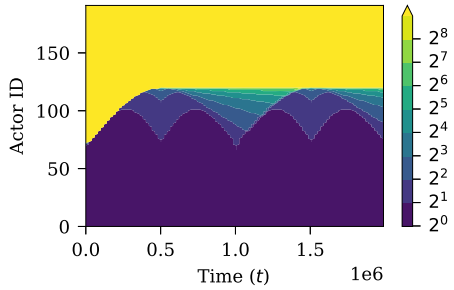
(a) Rollback for the dam break problem



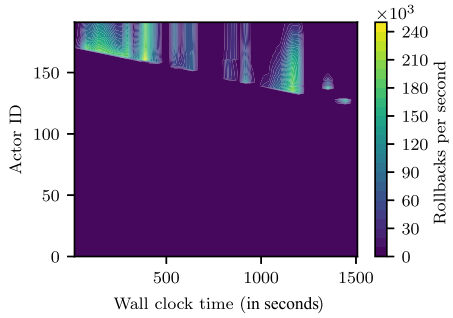
(b) Timestep size for the dam break problem



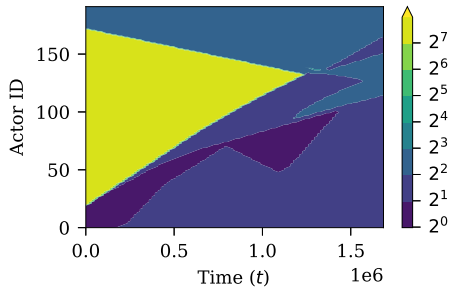
(c) Rollback for the Carrier-Greenspan problem



(d) Timestep size for the Carrier-Greenspan problem



(e) Rollback for the blastwave problem



(f) Timestep size taken for the blastwave problem

Fig. 5. Visual comparison for three different problems on a uniform mesh. The left column shows the rate of element updates being rolled back at a given actor and elapsed time during the simulation. The right column shows the timestep size taken at a given actor and simulation time. For these configurations, actor ID is linearly related to x-position in the domain.

side of the initial conditions. As seen in Figure 5(a), rollback occurs in elements *in front* of the shockwave. With the more strict timestep moving through the domain, coarser timesteps need to be rolled back and re-executed with finer timesteps. Since all cells in a submesh are updated as a unit, these rollbacks only occur when the timestep change lines up precisely with a submesh boundary.

At the coarse-fine timestep boundary, when the finely timestepped region updates, it relaxes its neighbor's upper bound, eventually permitting a coarse timestep update in the neighboring submesh. The process of each submesh updating and relaxing its neighbor's upper bound then propagates away from the boundary, creating a *work cascade*, whereby a set of tasks along the coarsely stepped light cone become executable. The speed of the work cascade propagation v_w is determined by the wall clock time required to execute update events and send push-flux events to neighboring submeshes. So long as the submeshes' timesteps do not change, the work done in each work cascade is useful work that does not need to be rolled back. However, when a submesh's timestep must be refined due to the simulated shockwave's propagation, it induces a *rollback cascade* along another light cone with speed v_{rb}. Since executing an update is more computationally expensive than processing a rollback, $v_w < v_{rb}$, allowing the rollback cascade to catch up with and terminate the work cascade. We can observe this effect in Figure 5(a), where rollbacks emanate from the timestep differences into the less frequently stepped regions.

But the interesting question is this: why does the cascade emanate further from the shock front in the more coarsely stepped region of the mesh (i.e., the actors with higher IDs)? The work associated with either work cascade is the same, although the cascade with the larger timestep will advance further in simulation time. Letting δ_{rb} be the delay between the misspeculated work cascade and the subsequent rollback cascade, the rollback cascade would terminate at a distance from the shock of $(v_{rb}v_w\delta_{rb})/(v_{rb} - v_w)$. We hypothesize that this delay varies based on the ratio of timestepping groups. When there is a larger difference in timesteps, there is more work to be performed at the shockwave interface that must now advance GVT at a slower rate. This leads to a longer delay before the rollback is issued, causing more rollback for the high ID actors than the low ID actors in Figure 5(a).

This speculation distance is particularly important for the Euler-blast wave problem, shown in Figure 5(e) and (f). This problem consists of two shockwaves meeting in the middle of the domain and interacting. The high actor IDs are simulating the lower energy blast, whereas the low actor IDs are simulating the higher energy blast. The two blasts are separated by a low pressure region, which is able to take a large timestep. In this case, the timestepping ratio is so large that the work cascade is able to make it across the entire vacuum state and cause erroneous events to be scheduled for the lower energy blast. Although not visible in Figure 5(e), the vacuum region does experience a rollback ($O(10^{-2}) - O(10^2)$ rollbacks per second). Since the work associated with this lower energy blast is non-trivial, most rollback is observable in the lower energy blast. We also see substantially more rollback when compared to the other two configurations in Figure 5(a) and (c). We determine that there is relatively limited benefit to speculating ahead of the high energy blast (i.e., independently advancing the lower energy blast). Rollback cascades from the misspeculation of the more computationally intensive higher energy blast undo speculative work of the lower energy blast. We also observe that the misspeculation substantially reduces once the two shocks meet and the discrepancy in timestep reduces. From the mathematical perspective, it is interesting to note that these rollback cascades are unphysical. The work done for the lower energy blast should be independent of what is happening far away. We believe that for most of the rollback cascade, the re-executed push flux has an identical state to the unexecuted push flux. One area of future work is implementing lazy cancellation [34] in Devastator to disrupt these rollback cascades.

These experiments showcase the latent irregularity in the application. We see that timestep changes are relatively infrequent and localized, although they may move throughout the domain. The benefits of speculation are less about doing useful work while being uncertain if it will need to be rolled back, but rather avoiding paying the expensive synchronization overheads required to *prove* that no causality violations occur.

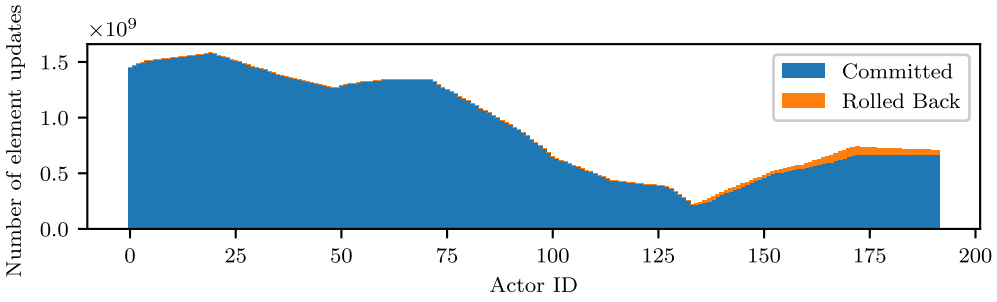


Fig. 6. The cumulative element updates and rollbacks for the Euler-blast wave problem on the uniform mesh.

The occurrence of speculation is fairly limited in all of these configurations. When the local prediction of incoming messages is accurate, we are able to prevent bad speculation by waiting rather than executing updates that will inevitably be rolled back. The execution of the local timestepping algorithm is equivalent to that of the task-graph-based implementations of local timestepping for linear problems for which $|\Lambda|$ is independent of the submesh state. Furthermore, in this section, we have artificially aggravated misspeculation by running an imbalanced configuration. However, most often the misspeculation happens off the critical path. In Figure 6, we show the cumulative element updates and rollbacks for each actor for the uniform Euler-blast wave problem. Rolled back element updates are concentrated on actors that are idly waiting for updates. If we identify the rank with the most work (consisting of actors 18 through 23), we note that only $3.1 \cdot 10^{-3}\%$ of element updates are rolled back.

4.3 Strong Scaling Comparison to STS-MPI

We turn to distributed performance with a strong scaling study of the Carrier-Greenspan problem on a polynomial mesh with 1.2 million elements. We compare the performance of the ALTS Devastator (ALTS-deva) to a synchronously stepped MPI implementation (STS-MPI). At 1.2 million elements, the smallest submesh size is 99 elements, limiting Devastator overheads per Section 4.1. For the MPI version, each submesh steps at each timestep (i.e., synchronously). To ensure a fair comparison over the existing baseline parallelization, STS-MPI implementation utilizes point-to-point non-blocking messages and overlaps communication with available local work, consistent with previous work [11, 14]. Furthermore, both implementations use the same actor data structures and application layer helper functions.

Since ALTS improves performance via work reduction, we use a throughput measure that approximates the throughput as work done for a synchronous problem divided by the execution time—that is,

$$TP = \frac{t^{\text{end}} n_{\text{elem}}}{\Delta t^{\min} T_X^n},$$

where $t^{\text{end}} = 2\pi$ is the simulation time, $n_{\text{elem}} = 1.2 \cdot 10^6$ is the number of elements in the mesh, $\Delta t_{\min} = 7.8125 \cdot 10^{-8}$ ($(|\Lambda|/\Delta x)_{\max} = 6.4 \cdot 10^6$) is the smallest timestep taken, and T_X^n is the execution time measured in seconds with n cores using parallelization X—either Devastator or MPI. Figure 7 compares the output of the MPI-synchronous timestepping (STS-MPI) implementation to the Devastator-ALTS (ALTS-deva) implementation. For the MPI implementation, the 32 core run has been omitted since it was unable to finish within queue time constraints. The core counts are chosen to highlight the scaling behavior of the ALTS-deva implementation. At 1,024 cores, the oversubscription factor is three submeshes per rank. Scaling out further without modifying

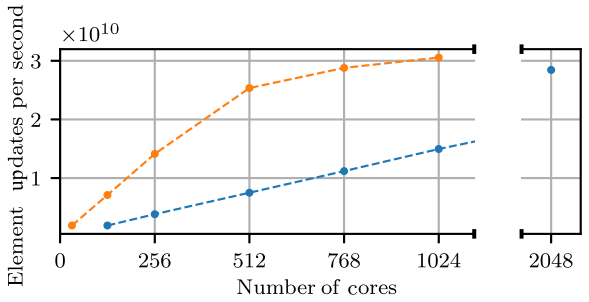


Fig. 7. Comparison of throughput (TP) versus the number of cores for the ALTS-deva implementation ● versus the STS-MPI implementation ●.

submesh sizes would lead to discrete load balancing issues, degrading performance. Since ALTS achieves its speedup through work reduction, we observe an improvement in *per core throughput* when run on 128 to 1,024 cores rather than an improvement in the maximum number of cores to which we are able to scale out. At 128 cores, the ALTS-deva implementation has a speedup of 3.75x over the STS-MPI implementation. This speedup degrades to 2.57x at 768 cores. Nonetheless, at 768 cores, the ALTS-deva implementation achieves a time to solution comparable to the 2,048-core STS-MPI configuration.

This strong scaling study is designed to show tangible benefits of ALTS to accelerate effective per core throughput. However, the more regular workload of the STS-MPI application remains more scalable. We have shown that the 2,048-core MPI run matches the best throughput of the ALTS-deva implementation. At 6,144 cores, the STS-MPI implementation has a parallel efficiency of 80% and a speedup over the best ALTS-deva time to solution of 2.42. This result is important to emphasize. If fastest time to solution remains the primary objective, synchronous timestepping remains the best option. However, if the problem exhibits large spatial variance in the wave speed $|\Lambda|$, can be solved within provided time constraints (e.g., supercomputer queue time limits), and can efficiently utilize compute resources—by breaking the mesh into sufficiently large submeshes while retaining sufficiently many submeshes per core—we expect ALTS to substantially reduce the requisite computational effort to simulate the given configuration.

The speedups obtained are ultimately determined by the distribution of element sizes $\{\Delta x_j\}$ and problem dynamics. Taking the ratio of element updates for the synchronous timestepping method divided by the number of element updates for ALTS, we find a theoretical speedup based on work estimates of $S^{work} = 5.38$. The remainder of this section will focus on explaining the discrepancy between the observed speedups between the STS-MPI implementation and the ALTS-deva implementation. There are five contributors worth examining when comparing the two implementations:

(1) *Time spent load balancing*: Due to Amdahl's law as well as the growing cost of the load balancing problem as the number of ranks increases, increasing amounts of time may be spent inside of the load balancing phase rather than advancing the state of the system. Between the 32-core and 512-core configuration, time spent load balancing went from 15 seconds to 17.9 seconds, a modest increase but less than 1% of the total execution time.
(2) *ALTS overhead*: The cost of computing the CFL condition for the shallow water equations requires evaluation of an additional square root, which causes divider-related stalls in the CPU. These computations are omitted for the STS-MPI implementation, which steps with a fixed step size. Since $|\Lambda|$ is constant for the lake at rest problem, we run the problem with and without the CFL computation on one node. To avoid running into issues with Devastator

overhead, we partition the 1.2 million element mesh into 180 actors. Comparing the execution times, we determine that 25% of the execution time is spent on the CFL computation. We remark that core utilization could be improved by leveraging approximate inverse square root algorithms or vectorizing the code, which we leave as topics for future investigation. Last, more complicated discretizations, such as high-order spatial discretizations, addition of slope limiters, and more complex stencils, would all increase the cost of the baseline's local compute and lower the percentage of time spent on CFL computation.

(3) *Rollback*: When bad speculation is detected, the updates performed will have to be undone, and the time spent executing those updates is essentially wasted. Similar to Section 4.2, only very few updates are rolled back. At 32 cores, 0.005% of element updates were rolled back, and at 512 cores, 0.009% of element updates were rolled back, suggesting that our performance optimization heuristics do a good job of identifying and preventing mis-speculation.

(4) *Event scheduling overhead*: The minimum mesh size has been chosen to limit the impact of Devastator runtime event scheduling overheads. These overheads are small but non-trivial and may degrade performance if many small submeshes are on the critical path.

(5) *Imbalance*: Given the fact that the load balancer must assign submeshes (rather than elements) to ranks as well as the dynamic changes in the work associated with each submesh, the load balance will not be perfect after each load balancing phase. Using the imbalance of the problem going into and out of the load balancing phase, we can approximate the time averaged imbalance during the simulation—that is, $I = (w_{a^*} - \overline{w_a})/\overline{w_a}$, where w_{a^*} is the maximum work per rank and $\overline{w_a}$ is the average work. At 32 cores, the problem is well balanced with $I = 0.02$. The load imbalance for the 512-core run worsens to 0.43. This is partially to be expected because at 512 cores, there are only six submeshes per rank. With only half of the mesh being wet on average, the number of submeshes available for load balancing is effectively halved. However, this imbalance is more pessimistic than the observed performance (i.e., a parallel efficiency of 83.7%). This discrepancy arises due to a mismatch between the actual work associated with a submesh and our cost model outlined in Section 2.3.5. First, Devastator overheads such as performance costs that are fixed per submesh are ignored in the load balancing problem. Second, as we strong scale, cache effects may be causing the code to run faster than expected. Nonetheless, the imbalance metric indicates that the load balancer begins to struggle as we strong scale to fewer submeshes per thread. Although there exist discrepancies between the modeled imbalance and the actual imbalance, the fact that the 32-core configuration has low imbalance and performs well suggests that our approach is able to effectively balance compute loads across ranks.

(6) *Worker thread mismatch*: Another loss of performance is caused by the fact that Devastator dedicates one thread per socket for communication management. Thus, per node on Cori, ALTS-deva only uses 30 worker threads to execute events, whereas STS-MPI uses all 32 cores.

Based on these analyses, we determine that the primary drivers of performance loss are computing the CFL condition and worsening load balance. Nonetheless, at low core counts, the problem can be sufficiently balanced, and the ALTS-deva implementation achieves throughput improvements of up to 4× over the STS-MPI implementation, which is 74.5% of the peak theoretical speedup.

4.4 Weak Scaling Performance

To show that ALTS can be scaled to larger distributed problems, we perform a weak scaling study using the Carrier-Greenspan problem on the uniform mesh. Since the performance of the ALTS method strongly depends on the distribution of the wave speed, $|\Lambda|$, it is important to simulate the full 2π time to compare performance across different problem sizes. Due to the fact that the

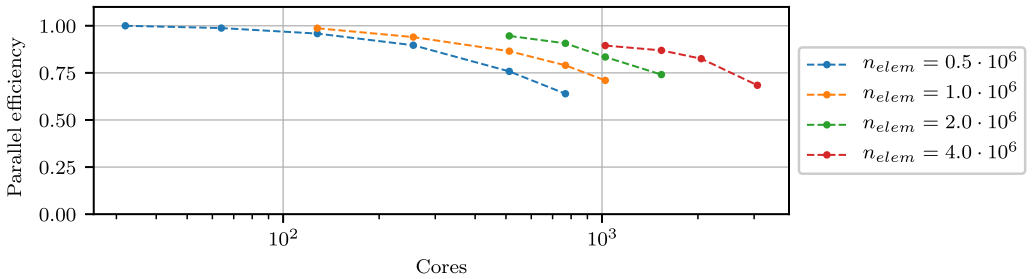


Fig. 8. Weak scaling performance for the Carrier-Greenspan problem on the uniform mesh.

amount of computational work scales quadratically with the number of elements, and the fact that the polynomial mesh is computationally too expensive, using the uniform mesh allows us to consider a larger range of problem sizes.

We consider four submeshes with 500,000, 1 million, 2 million, and 4 million elements. The problem is partitioned into 2,880 actors per 500,000 elements. For reference, storm surge simulation meshes have $O(10^6)$ elements [27]. This leads to a minimum submesh size of 174 elements, ensuring that simulator overheads are reasonably amortized. The rebalance frequency is taken to be once every 8,192 Δt_{min}. As the number of elements doubles, Δt_{min} is halved, and the total number of load balancing phases doubles. With the work per core per epoch remaining constant across weak scaling configurations, we observed that this rebalance frequency provided the optimal performance.

Each configuration is run over a few core counts to demonstrate strong scaling characteristics. The parallel efficiency versus core count is shown in Figure 8. The simulation weak scales well with the 4 million element mesh obtaining 90% of ideal per core throughput at 1,024 cores and 68% of ideal per core throughput at 3,072 cores. Looking at the imbalance across the runs, we see that the average imbalance for the 32-core 500,000 element mesh was 0.03. For the 4 million element mesh, the imbalance increases to 0.14 for the 1,024-core configuration to 0.33 for the 3,072-core configuration, leading to a slight degradation in scalability. The choice of a uniform mesh will lead to higher fractions of elements updating each timestep than a polynomial mesh. Thus, the average number of elements per core may not be a good indicator of work required to achieve good scaling behavior, and nonetheless we conclude that if the problem is load balanced and sufficiently large, the ALTS method can be scaled out.

4.5 Impact of Performance Optimizations

To help quantify the impact of the performance optimization strategies described in Section 2.3, we consider a few problems and then enable the strategies to observe their impact. This section is split into two sections highlighting the benefits of waiting on misspeculation and load balancing.

4.5.1 Waiting Heuristics. To determine the impact of the performance optimizations outlined in Sections 2.3.1 through 2.3.4, we incrementally enable them and report the impact on speedups relative to a local timestepping baseline with no waiting heuristics enabled. We consider a 512-core lake at rest problem ($n_{\text{elem}} = 6 \cdot 10^5$, $(|\Lambda|/\Delta x)_{\text{max}} = 1{,}370{,}368{,}000$, $t^{\text{end}} = 0.1$, $n_{\text{actors}} = 1{,}440$) and 768-core lake at rest problem ($n_{\text{elem}} = 1.2 \cdot 10^6$, $(|\Lambda|/\Delta x)_{\text{max}} = 2{,}740{,}736{,}000$, $t^{\text{end}} = 0.05$, $n_{\text{actors}} = 1{,}440$). The $(|\Lambda|/\Delta x)_{\text{max}}$ parameters have been chosen to be intentionally large—that is, $(|\Lambda|/\Delta x)_{\text{max}} \approx 256/\Delta x_{\text{min}}$. To avoid exaggerating the impact of the waiting heuristics, these problems have been sized to be sufficiently large to sustain a reasonable parallel efficiency. The lake at rest configurations achieve 90% of the throughput rate of lake at rest problems with twice the

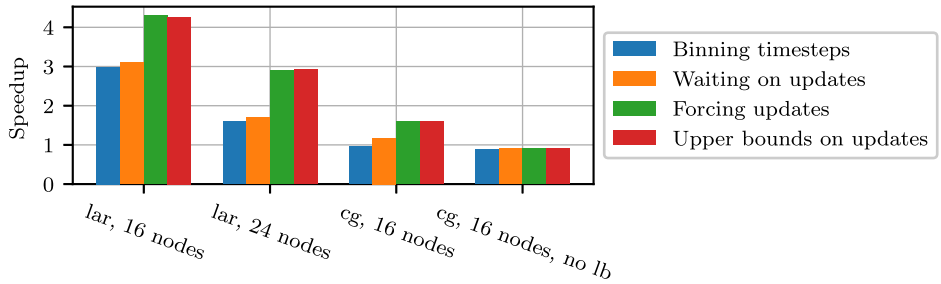


Fig. 9. Speedup obtained by incrementally enabling optimizations: binning timesteps (Section 2.3.1), waiting on updates (Section 2.3.2), forcing updates (Section 2.3.3), and upper bounds on updates (Section 2.3.4).

number of elements. We also consider two 512-core Carrier-Greenspan configurations: one that exactly mimics the strong scaling configuration and the second the same as the first except load balancing is disabled. For the Carrier-Greenspan configurations, $(|\Lambda|/\Delta x)_{\max}$ is chosen to be close to the optimal value. The cumulative speedups for each configuration relative to a baseline without any optimizations enabled are presented in Figure 9. For these bar graphs, cumulative implies that the preceding optimizations remain enabled (e.g., the speedup for Section 2.3.3 has also enabled Sections 2.3.1 and 2.3.2).

For the lake at rest problems, we observe the largest benefits occur when binning timesteps to powers of 2 (Section 2.3.1) and forcing neighbors to update (Section 2.3.3). The large $(|\Lambda|/\Delta x)_{\max}$ values cause a substantial number of local ordering updates to occur as described in Section 2.3.1 and impede the submesh partitioner, as the expected number of element updates is not well approximated by the local CFL condition. Introducing timestep binning (Section 2.3.1) elides local ordering-related updates with CFL-condition-related updates and causes the number of element updates committed per rank to become roughly equal. For the 512-core lake at rest problem, this corresponds to the most overworked rank committing 1.67x the average number of element updates before timestep binning is introduced and 1.01x the average number of element updates after timestep binning is introduced. Forcing neighbors to update (Section 2.3.2) and waiting on those updates (Section 2.3.2) eliminates most misspeculation, causing a second increase in improvement. For the Carrier-Greenspan problem, the close-to-optimal $(|\Lambda|/\Delta x)_{\max}$ reduces the extra updates associated with enforcing local ordering. In that case without Section 2.3.1, timesteps are binned into multiples of $\Delta t_{\min}$ (rather than $2^8 \Delta t_{\min}$). In conjunction with a more optimal timestep taken at the critical path, we observe that the time to solution slightly increases after binning timesteps into powers of 2. Similar to the lake at rest problem, a large performance improvement is only observed after the first three optimizations have been enabled.

Another takeaway from these results is that misspeculation impacts time to solution only when it delays the critical path. Since rollback is triggered by arrival of delayed push fluxes from neighboring submeshes, when the simulation is very imbalanced, the rank which is on the critical path has the slowest rate of advancing the simulation time, and as this rank makes progress, messages that would incur rollback have most likely already arrived. The unbalanced Carrier-Greenspan configuration with no performance optimizations enabled executed 1.3x more element updates than the configuration with all optimizations enabled yet ran 10% faster. Only when considering load balanced problems, either through construction (lake at rest) or via load balancing (Carrier-Greenspan), do we observe performance improvements by reducing misspeculation. In these cases, the rollback-causing messages have not necessarily arrived, and anticipating and waiting on these messages, when feasible, leads to improved time to solution. For the 768-core lake at rest problem,

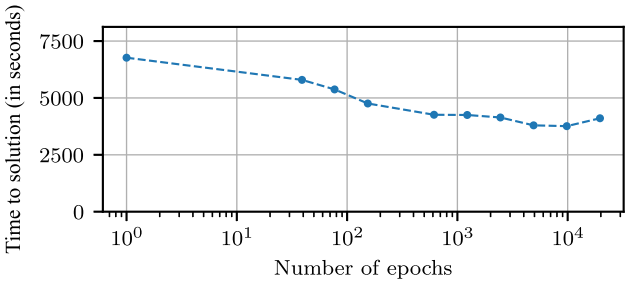


Fig. 10. Impact of load balancing frequency on time to solution.

the percentage of the cell updates rolled back on the critical path goes down from 46% down for the Section 2.3.1 configuration to zero for the fully optimized case.

4.5.2 Load Balancing. For the 512-core strong scaling problem, load balancing plays a significant role, delivering a speedup of 1.8x over a run without dynamic load balancing. In Figure 10, we compare the impact of rebalancing frequency on time to solution. We note that the problem has a relatively wide minimum. With only 614 epochs, the time to solution is only 13% larger than the optimal time to solution, which occurs with 9,818 epochs. To understand this behavior, note that the worst case imbalance corresponds to when all submeshes on a rank are wet. In that sense, once the rebalance frequency begins to significantly exceed the rate of flooding, the problem becomes effectively balanced.

Thus, it is important to emphasize that while the algorithm is and must be stable under very rapidly changing values of $|\Lambda|$, these scenarios are rarely seen in practice. Rather, we see gradual shifts in timestepping group membership, which if not accounted for will lead to instabilities but are nevertheless gradual enough that these simulations can be load balanced effectively. Although we have only considered a 1D problem, the simulated dynamics—local refinement in the middle of the domain and flooding moving into the refined region—represent a worst case scenario for hurricane storm surge modeling. Rarely would all inundation be concentrated in a single submesh; rather, inundation would be spread out over a large front, lowering the overall rate of imbalance generation.

5 CONCLUSION

In this article, we analyze the parallel performance of ALTS using a Timewarp-based optimistic PDES. We introduce several performance optimizations, including load balancing and waiting heuristics, for the ALTS implementation that provide a 1.6× to 2.8× speedup over a baseline PDES implementation. With these optimizations, ALTS achieves up to 4× speedup (74.5% of the theoretical speedup) in time to solution over the state-of-the-art MPI parallelization and weak scales over core counts of interest to hurricane storm surge forecasting. Furthermore, analysis of misspeculation shows that for fluid simulations with gradually locally refined meshes and sufficiently large event sizes, rollback has a minor impact on performance.

We conclude with some remarks about the generalizability of the method to higher dimensions and the presented results. The timestepping algorithm as presented is designed to hold for higher dimensions. The boundary types would become conserved quantities evaluated at shared submesh faces or edges. The CFL condition has been presented to support multiple neighbors as the wave speeds continue to be evaluated over the adjacent neighbors. Similarly, the local ordering constraint is a pairwise relation between submeshes. Thus, the local ordering constraint naturally extends to support more than two neighboring submeshes.

Moving to higher dimensions introduces considerable complexity in the application layer. Spatial derivatives become gradients, and geometric information is required to properly compute integrals. However, all of these details are well established (e.g., [22]) and restricted to the application layer. The ALTS software stack has been specifically designed to facilitate reuse of existing software.

From a task-graph topology standpoint, the timestepping layer continues to update a submesh and then share coupled values with neighboring submeshes. Thus, the extension to higher dimensions has limited impact on the other layers of the ALTS stack. This underpins why we expect performance results here to generalize as well. In previous work characterizing the performance of task-based implementations of 2D discontinuous shallow water solvers [11], METGs were found to be consistent with the METGs for a 1D finite difference scheme [37]. The important discovery in this work is establishing that changing of submeshes' timesteps is limited and gradual. With this limited irregularity and heuristics to identify the majority of misspeculation, the presented parallelization is able to load balance and efficiently utilize hardware resources.

REFERENCES

[1] Emmanuel Audusse, François Bouchut, Marie-Odile Bristeau, Rupert Klein, and Benoît Perthame. 2004. A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows. *SIAM Journal on Scientific Computing* 25, 6 (2004), 2050–2065.

[2] Hervé Avril and Carl Tropper. 1996. The dynamic load balancing of clustered time warp for logic simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*. IEEE, Los Alamitos, CA, 20–27. https://doi.org/10.1145/238788.238804

[3] Pavol Bauer, Jonatan Lindén, Stefan Engblom, and Bengt Jonsson. 2015. Efficient inter-process synchronization for parallel discrete event simulation on multicores. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS'15)*. ACM, New York, NY, 183–194.

[4] Brian Blanton, John McGee, Jason Fleming, Carola Kaiser, Hartmut Kaiser, Howard Lander, Rick Luettich, Kendra Dresback, and Randy Kolar. 2012. Urgent computing of storm surge for North Carolina's coast. *Procedia Computer Science* 9 (2012), 1677–1686.

[5] Onno Bokhove. 2005. Flooding and drying in discontinuous Galerkin finite-element discretizations of shallow-water equations. Part 1: One dimension. *Journal of Scientific Computing* 22, 1 (2005), 47–82.

[6] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A high-performance, portable communication library for exascale. In *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, Vol. 11882. Springer, 138–158.

[7] Azzedine Boukerche and Sajal K. Das. 1997. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*. IEEE, Los Alamitos, CA, 20–28.

[8] Maximilian Bremer. 2020. *Task-Based Parallelism for Hurricane Storm Surge Modeling*. Ph.D. Dissertation. The University of Texas at Austin.

[9] Maximilian Bremer, John Bachan, Cy Chan, and Clint Dawson. 2021. Speculative parallel execution for local timestepping. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'21)*. ACM, New York, NY, 83–94. https://doi.org/10.1145/3437959.3459257

[10] Maximilian Bremer, John Bachan, Cy Chan, and Clint Dawson. 2022. Adaptive total variation stable local timestepping for conservation laws. *Journal of Computational Physics* 463 (2022), 111176.

[11] Maximilian Bremer, Kazbek Kazhyken, Hartmut Kaiser, Craig Michoski, and Clint Dawson. 2019. Performance comparison of HPX versus traditional parallelization strategies for the discontinuous Galerkin method. *Journal of Scientific Computing* 80 (2019), 878–902.

[12] Maximilian H. Bremer, John D. Bachan, and Cy P. Chan. 2018. Semi-static and dynamic load balancing for asynchronous hurricane storm surge simulations. In *Proceedings of the Parallel Applications Workshop, Alternatives to MPI (PAW-ATM'18)*. IEEE, Los Alamitos, CA, 44–56.

[13] Alexander Breuer, Alexander Heinecke, and Michael Bader. 2016. Petascale local time stepping for the ADER-DG finite element method. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, Los Alamitos, CA, 854–863.

[14] Steven Brus, Damrongsak Wirasaet, Joannes J. Westerink, and Clint Dawson. 2017. Performance and scalability improvements for discontinuous Galerkin solutions to conservation laws on unstructured grids. *Journal of Scientific Computing* 70 (2017), 210–242.

[15] Steven R. Brus, Damrongsak Wirasaet, Ethan J. Kubatko, Joannes J. Westerink, and Clint Dawson. 2019. High-order discontinuous Galerkin methods for coastal hydrodynamics applications. *Computer Methods in Applied Mechanics and Engineering* 355 (2019), 860–899.
[16] Giacomo Capodaglio and Mark Petersen. 2022. Local time stepping for the shallow water equations in MPAS. *Journal of Computational Physics* 449 (2022), 110818. https://doi.org/10.1016/j.jcp.2021.110818
[17] Christopher D. Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing* 62, 11 (2002), 1648–1669.
[18] George F. Carrier and Harvey P. Greenspan. 1958. Water waves of finite amplitude on a sloping beach. *Journal of Fluid Mechanics* 4, 1 (1958), 97–109.
[19] François E. Cellier, Ernesto Kofman, Gustavo Migoni, and Mario Bortolotto. 2008. Quantized state system simulation. In *Proceedings of Grand Challenges in Modeling and Simulation (GCMS'08)*. 504–510.
[20] Cy Chan, Bin Wang, John Bachan, and Jane Macfarlane. 2018. Mobiliti: Scalable transportation simulation using high-performance parallel computing. In *Proceedings of the 2018 21st International Conference on Intelligent Transportation Systems (ITSC'18)*. IEEE, Los Alamitos, CA, 634–641.
[21] Clawpack Development Team. 2020. Clawpack Version 5.7.1. Retrieved September 3, 2022 from https://doi.org/10.5281/zenodo.4025432
[22] Bernardo Cockburn and Chi-Wang Shu. 2001. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing* 16 (2001), 173–261.
[23] Emil M. Constantinescu and Adrian Sandu. 2007. Multirate timestepping methods for hyperbolic conservation laws. *Journal of Scientific Computing* 33, 3 (Dec. 2007), 239–278.
[24] Samir R. Das. 2000. Adaptive protocols for parallel discrete event simulation. *Journal of the Operational Research Society* 51 (2000), 385–394.
[25] Samir R. Das and Richard M. Fujimoto. 1997. Adaptive memory management and optimism control in time warp. *ACM Transactions on Modeling and Computer Simulation* 7, 2 (April 1997), 239–271.
[26] Clint Dawson and Robert Kirby. 2001. High resolution schemes for conservation laws with locally varying time steps. *SIAM Journal on Scientific Computing* 22, 6 (2001), 2256–2281.
[27] Clint Dawson, Ethan J. Kubatko, Joannes J. Westerink, Corey Trahan, Christopher Mirabito, Craig Michoski, and Nishant Panda. 2011. Discontinuous Galerkin methods for modeling hurricane storm surge. *Advances in Water Resources* 34, 9 (2011), 1165–1176.
[28] Clint Dawson, Corey Jason Trahan, Ethan J. Kubatko, and Joannes J. Westerink. 2013. A parallel local timestepping Runge-Kutta discontinuous Galerkin method with applications to coastal ocean modeling. *Computer Methods in Applied Mechanics and Engineering* 259 (2013), 154–165.
[29] Michael Dumbser, Martin Käser, and Eleuterio F. Toro. 2007. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes—V. Local time stepping and p-adaptivity. *Geophysical Journal International* 171, 2 (2007), 695–717.
[30] Alois Ferscha. 1995. Probabilistic adaptive direct optimism control in time warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS'95)*. IEEE, Los Alamitos, CA, 120–129. https://doi.org/10.1145/214282.214320
[31] Xenofon Floros, Federico Bergero, François E. Cellier, and Ernesto Kofman. 2011. Automated simulation of Modelica models with QSS methods—The discontinuous case. In *Proceedings of the 8th International Modelica Conference*. 657–667.
[32] Richard M. Fujimoto. 1990. Parallel discrete event simulation. *Communications of the ACM* 33, 10 (1990), 30–53.
[33] Richard M. Fujimoto. 1993. Feature article—Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing* 5, 3 (1993), 213–230.
[34] A. Gafni. 1988. Rollback mechanism for optimistic distributed systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*. 61–67.
[35] David W. Glazer and Carl Tropper. 1993. On process migration and load balancing in time warp. *IEEE Transactions on Parallel and Distributed Systems* 4, 3 (1993), 318–327.
[36] Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. 2001. Strong stability-preserving high-order time discretization methods. *SIAM Review* 43, 1 (2001), 89–112.
[37] Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio. 2015. The performance implication of task size for applications on the HPX runtime system. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*. IEEE, Los Alamitos, CA, 682–689.
[38] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. 1993. *Solving Ordinary Differential Equations I.* Springer-Verlag, Berlin, Germany.
[39] Ami Harten. 1983. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics* 49, 3 (1983), 357–393.

[40] Michael A. Heroux and Gabrielle Allen. 2016. *Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges Workshop Report*. Technical Report. NITRD.
[41] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software* 31, 3 (Sept. 2005), 363–396.
[42] Thi-Thao-Phuong Hoang, Wei Leng, Lili Ju, Zhu Wang, and Konstantin Pieper. 2019. Conservative explicit local time-stepping schemes for the shallow water equations. *Journal of Computational Physics* 382 (2019), 152–176.
[43] Jen-Chih Huang, Xiangmin Jiao, Richard M. Fujimoto, and Hongyuan Zha. 2007. DAG-guided parallel asynchronous variational integrators with super-elements. In *Proceedings of the 2007 Summer Computer Simulation Conference (SCSC'07)*. 691–697.
[44] David R. Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (1985), 404–425.
[45] Laxmikant V. Kale and Sanjeev Krishnan. 1993. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. 91–108.
[46] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.
[47] Ernesto Kofman and Sergio Junco. 2001. Quantized-state systems: A DEVS approach for continuous system simulation. *Transactions of the Society for Modeling and Simulation International* 18, 3 (2001), 123–132.
[48] Lilia Krivodonova. 2010. An efficient local time-stepping scheme for solution of nonlinear conservation laws. *Journal of Computational Physics* 229, 22 (2010), 8537–8551.
[49] Randall J. LeVeque. 1992. *Numerical Methods for Conservation Laws*. Birkäuser Verlag, Basel.
[50] A. Lew, J. E. Marsden, M. Ortiz, and M. West. 2003. Asynchronous variational integrators. *Archive for Rational Mechanics and Analysis* 167, 2 (April 2003), 85–146.
[51] Jonatan Lindén, Pavol Bauer, Stefan Engblom, and Bengt Jonsson. 2017. Exposing inter-process information for efficient parallel discrete event simulation of spatial stochastic systems *(SIGSIM-PADS'17)*. ACM, New York, NY, 53–64.
[52] Jonatan Lindén, Pavol Bauer, Stefan Engblom, and Bengt Jonsson. 2018. Fine-grained local dynamic load balancing in PDES. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'18)*. ACM, New York, NY, 201–212. https://doi.org/10.1145/3200921.3200928
[53] Xu-Dong Liu, Stanley Osher, and Tony Chan. 1994. Weighted essentially non-oscillatory schemes. *Journal of Computational Physics* 115, 1 (1994), 200–212.
[54] Eric Mikida and Laxmikant Kale. 2018. Adaptive methods for irregular parallel discrete event simulation workloads. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'18)*. ACM, New York, NY, 189–200. https://doi.org/10.1145/3200921.3200936
[55] Yuri A. Omelchenko and Homa Karimabadi. 2006. Event-driven, hybrid particle-in-cell simulation: A new paradigm for multi-scale plasma modeling. *Journal of Computational Physics* 216, 1 (2006), 153–178.
[56] Yuri A. Omelchenko and Homa Karimabadi. 2007. A time-accurate explicit multi-scale technique for gas dynamics. *Journal of Computational Physics* 226, 1 (2007), 282–300.
[57] Yuri A. Omelchenko and Homa Karimabadi. 2012. HYPERS: A unidimensional asynchronous framework for multiscale hybrid simulations. *Journal of Computational Physics* 231, 4 (2012), 1766–1780.
[58] Stanley Osher and Richard Sanders. 1983. Numerical approximations to nonlinear conservation laws with locally varying time and space grids. *Mathematics of Computation* 41, 164 (1983), 321–336.
[59] Anne Reinarz, Dominic E. Charrier, Michael Bader, Luke Bovard, Michael Dumbser, Kenneth Duru, Francesco Fambri, et al. 2020. ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems. *Computer Physics Communications* 254 (2020), 107251.
[60] John R. Rice. 1960. Split Runge-Kutta method for simultaneous equations. *Journal of Research of the National Bureau of Standards: Mathematics and Mathematical Physics: B* 64B, 3 (1960), 151–170.
[61] Martin Schlegel, Oswald Knoth, Martin Arnold, and Ralf Wolke. 2009. Multirate Runge-Kutta schemes for advection equations. *Journal of Computational and Applied Mathematics* 226, 2 (2009), 345–357.
[62] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, et al. 2020. Task Bench: A parameterized benchmark for evaluating parallel runtime performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'20)*. IEEE, Los Alamitos, CA, Article 62, 15 pages.
[63] Gary A. Sod. 1978. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *Journal of Computational Physics* 27, 1 (1978), 1–31.
[64] Peter K. Sweby. 1984. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM Journal on Numerical Analysis* 21, 5 (1984), 995–1011.

[65] Seng Chuan Tay, Yong Meng Teo, and Siew Theng Kong. 1997. Speculative parallel simulation with an adaptive throttle scheme. *ACM SIGSIM Simululation Digest* 27, 1 (June 1997), 116–123.

[66] Thomas Unfer. 2021. Third order asynchronous time integration for gas dynamics. *Journal of Computational Physics* 440 (2021), 110434. https://doi.org/10.1016/j.jcp.2021.110434

[67] Carsten Uphoff, Sebastian Rettenberger, Michael Bader, Elizabeth H. Madden, Thomas Ulrich, Stephanie Wollherr, and Alice-Agnes Gabriel. 2017. 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, NY, Article 21, 16 pages.

[68] Paul Woodward and Phillip Colella. 1984. The numerical simulation of two-dimensional fluid flow with strong shocks. *Journal of Computational Physics* 54, 1 (1984), 115–173.

Received 2 December 2021; revised 10 May 2022; accepted 6 June 2022